JPA-SCPI Parser

User Manual

JPA Consulting

JPA Consulting

www.jpacsoft.com

Contacts

Technical support: support@jpacsoft.com

Sales enquires: sales@jpacsoft.com

JPA-SCPI Parser V1.3.1 User Manual, release 1

Contents

1 Licence Agreement

1.1 This document is a legally binding Licence Agreement (the "Agreement") made between "the Licensee" and the “Manufacturer”

1.2 By purchasing or otherwise using this “Product”, including computer source code, computer software, associated media, any printed materials, and any "online" or electronic documentation the Licensee agrees to be bound by the terms of this Agreement

2. GENERAL TERMS

2.1 **Definitions:** the following expressions shall have the following meanings:

- *"Manufacturer" – “JPA Consulting”*
- *"JPA Consulting"* – JPA Consulting Limited, whose registered office is at Suite 2, Garrad House, 2 – 6 Homesdale Road, Bromley, Kent, BR2 9LZ, UK
- "*Licensee*" - the person, firm or company that has placed an Order
- *"Product"* – JPA-SCPI Parser provided by JPA Consulting, including software in which JPA Consulting has sub-licensing rights, in executable, machine readable, object, printed or interpreted form, including any documentation, modifications, improvements, or updates supplied to the Licensee under any Order
- *"Licence Term"* – duration of Agreement subject always to Clause 6 hereof
- "*Proprietary Information*" - all intellectual property rights including but not limited thereto all copyrights, design rights (registered and unregistered), patents, trademarks, designs, formula, code and other similar data relating to the Product
- "*Order*" - any purchase order issued by the Licensee for the Product from JPA Consulting
- "*Quotation*" - any quotation for the supply of the Product issued by JPA Consulting
- "*Site*" - the location for which the Product may be used identified in the Order
- “*Computer Program*” - any form of set of instructions that is able to run on any form of microprocessor, including a micro-controller within a piece of electronic equipment or any form of personal computer
- "*Specification*" - the written specification of the Product maintained during development and contained in documentation
- "*Support Agreement*" - the Software Maintenance and Technical Support facility provided by JPA Consulting

2.2 **Incorporation of Terms:** these Terms shall apply to the Product(s) supplied by JPA Consulting under any Order placed by the Licensee

2.2.1 In the event of any ambiguity to any provision of this Agreement or ruling by a court of competent jurisdiction to be illegal, invalid or unenforceable, the remaining provisions shall remain in full force and effect

3. GRANT OF LICENSE

3.1 The Manufacturer hereby grants, and Licensee hereby accepts, subject to the terms and conditions of this Agreement a non-exclusive, non-transferable and non-assignable license to use the Product

3.2 The Product is protected by copyright laws and international copyright treaties, as well as other intellectual property laws and treaties

3.3 The Product is licensed, not sold - any rights not explicitly granted under this Agreement are hereby reserved

3.4 The Licensee may not resell, rent, lease, or distribute any part of the Product, except the source code component of the Product, and only then as a compiled component of a Computer Program

3.5 The Licensee may only incorporate the Product into applications and equipment produced or manufactured by Licensee's organisation and sold under the Licensee's organisation name

3.6 The Licensee is entitled to make sufficient copies of the Product or parts of the Product for use by the software developers of Licensee's organisation, subject to the terms and conditions of this Agreement

3.7 If the Licensee acting as consultant wishes to use the Product in development of applications or equipment for more than one organisation then the Licensee requires a multi-brand licence to provide the Product to those organisations

3.8 Any source code component of the Product used as a compiled component of a Computer Program that is distributed or accessible outside the Licensee's organisation (including use from the Internet) must be protected to the extent that it cannot be easily extracted or decompiled

3.9 The application the Licensee distributes shall not be a software development tool intended for distribution to other software developers or programmers

3.10 The Licensee may not resell, rent, lease, or distribute products created from the Product in any form that could compete with the Manufacturer

3.11 Failure to comply with and adhere to the terms and conditions of this Licence could subject the Licensee to legal action by JPA Consulting and/or the termination of this licence

4. LEGAL JURISDICTION

4.1 This contract is governed by the law of England & Wales

4.2 The Licensee acknowledges having read this licence and having understood all its terms, to agree to respect them in whole

5. COPYRIGHT

5.1 The Manufacturers' Product including source code and all documentation in whatever physical form is copyrighted and contains proprietary information

5.2 The Licensee shall not distribute or reveal any parts of the Product to anyone other than the software developers of Licensee's organisation

5.3 The Licensee could be legally responsible for any infringement of intellectual property rights that caused or encouraged by Licensee's failure to abide by the terms of this Agreement

5.4 The Manufacturer reserves all rights not specifically granted to the Licensee

6. MODIFICATIONS

6.1 **Modifications:** the Manufacturer will provide the Licensee with error corrections, bug fixes, patches, and or updates to the Product licensed hereunder to the extent available in accordance with the Manufacturer's release schedule for a period of one (1) year from the date of despatch

6.2 **Updates:** if this copy of the Product is an upgrade from an earlier version of the Product, it is provided on a licence exchange basis

6.2.1 The Licensee agrees by installation and use of this copy of the Product to:

(i) Voluntarily terminate any earlier end-user licence, and

(ii) To not continue to use the earlier version of the Product nor transfer it to another

6.3 **Title:** all such error corrections, bug fixes, patches, updates, or other modifications shall remain the sole property of the Manufacturer

7. LICENCE TERM

7.1 This Agreement provides a long term licence of 25 years unless a quotation has specifically stated a shorter period and such shorter period has been specifically ordered by the Licensee

8. WARRANTY & RISKS

8.1 Although the Manufacturer has thoroughly tested the Product and reviewed the documentation, the Manufacturer cannot guarantee that the Product will suit the Licensees' needs, nor that it will function correctly in every hardware or software environment, nor that its operation will be uninterrupted or infallible

8.2 Efforts have been made to assure that all parts of the Product, including the source code, are correct, reliable, and technically accurate, however the Product is licensed to the Licensee as is and without warranties as to performance of merchantability, fitness for a particular purpose or use, or any other warranties whether expressed or implied. Licensee's organisation and all users of the Product assume all risks when using it

8.3 The Manufacturer, distributors, and resellers of the Product shall not be liable for any consequential, incidental, punitive or special damages arising out of the use of or inability to use the Product or the provision of or failure to provide support services, even if advised of the possibility of such damages

8.4 In any case, the entire liability under any provision of this agreement shall be limited to the amount actually paid by the Licensee for the Product

8.5 Should the Licensee discover a material defect on the media upon which the Product is furnished (not applicable if the program is downloaded from a server or if copied from other media) within ninety (90) days following the date of purchase, the media will be replaced free of charge

8.6 Except insofar as it has been stated in paragraph one (above), the Manufacturer grants no guarantee and acknowledges no express or tacit guarantee regarding the Product, its quality, its description, its retail value, or its appropriateness for any specific function

8.7 In no case will the Manufacturer assume any responsibility for any direct or indirect damages, losses, loss of revenues, or for any loss of recorded data concerning the use or the unsuitability of the Product

8.8 No distributor, dealer, agent, or intermediary is authorised to modify or otherwise amend this declaration of guarantee and of limited liability

9. HIGH RISK ACTIVITIES

9.1 The Product is not fault-tolerant and is not designed, or intended for use or resale as on-line control equipment in hazardous environments requiring fail-safe performance, in which the failure of the Product could lead directly to death, personal injury, or severe physical or environmental damage ("High Risk Activities")

9.2 "High Risk Activities" could include but are not limited exclusively to the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, direct life support machines, or weapons systems

9.3 The Manufacturer and its suppliers specifically disclaim any express or implied warranty of fitness for High Risk Activities

10. CONFIDENTIALITY

10.1 **Acknowledgement:** the Licensee hereby acknowledges and agrees that the Product, including source code and documentation in whatever physical form constitutes and contains valuable proprietary products and trade secrets of the Manufacturer and/or its suppliers, embodying substantial creative efforts and confidential information, ideas, and expressions. Accordingly, Licensee agrees to treat (and take precautions to ensure that its employees treat) all components of the Product as confidential in accordance with the confidentiality requirements and conditions set forth below:

10.1.1 Each party agrees to keep confidential all confidential information disclosed to it by the other party in accordance herewith

10.1.2 Each party agrees to protect the confidentiality thereof in the same manner it protects the confidentiality of similar information and data of its own (at all times exercising at least a reasonable degree of care in the protection of confidential information)

Provided, however, that neither party shall have any such obligation with respect to use of disclosure to others not parties to this Agreement of such confidential information as can be established to have:

(a) Been known publicly

(b) Been known generally in the industry before communication by the disclosing party to the recipient

(c) Become know publicly, without fault on the part of the recipient, subsequent to disclosure by the disclosing party

(d) Been known otherwise by the recipient before communication by the disclosing party; or

(e) Been received by the recipient without any obligation of confidentiality from a source (other than the disclosing party) lawfully having possession of such information

10.2 **Injunctive Relief:** the Licensee acknowledges that the unauthorised use, transfer, or disclosure of any part of the Product or copies thereof will:

(i) Diminish substantially the value to the Manufacturer of the trade secrets and other proprietary interests that are the subject of this Agreement

(ii) Render the Manufacturer's remedy at law for such unauthorised use, disclosure, or transfer inadequate; and

(iii) Cause irreparable injury in a short time period

If the Licensee breaches any of its obligations with respect to the use or confidentiality of the Product, the Manufacturer shall be entitled to equitable relief to protect its interests therein, including, but not limited to, preliminary and permanent injunctive relief

10.3 **Survival:** Licensee's obligations under this Article will survive the termination of this Agreement or of any licence granted under this Agreement for whatever reason

11. TERMINATION

11.1 Without prejudice to any other rights, JPA Consulting may terminate this agreement if the Licensee fails to comply with its terms and conditions

11.2 In any such event the Licensee must permanently uninstall all copies of the Product

2 Introduction

2.1 What is JPA-SCPI Parser?

JPA-SCPI Parser gives your in-house software development team the tools to quickly and easily create SCPI command parsers for your programmable instruments. No longer do you need to employ an outside consultancy house, or spend a large amount of time creating a SCPI parser from scratch.

Written in ANSI/ISO C and designed to be efficient in use of ROM and RAM, the JPA-SCPI Parser library of source code is suitable for use on almost any embedded processor/hardware platform. Comprehensive instructions in this manual give a step-by-step guide to creating all types of command syntax. In addition, source code templates are provided to get you going even faster.

JPA-SCPI Parser supports all the common requirements of any programmable instrument. However, if you need to expand JPA-SCPI Parser in any way for your own needs, then you can. To assist you the source code is fully commented, and a separate *Design Notes* document is also supplied, describing the design of the functions and the structures used.

2.2 The SCPI Standard

SCPI is the standard of choice for defining command sets of programmable instruments.

Supported by all the major manufacturers, SCPI provides customers with faster and cheaper installation, development and support by giving their technicians and programmers a consistent command language across all different types of programmable instrument.

By making your instrumentation SCPI-compatible, you could gain both from increased sales and also reduced support costs, since many customers will already be familiar with SCPI and require less basic technical support.

2.2.1 A Note on SCPI Compliance

When implementing your command interface for your instrument there are two approaches:

- Full SCPI compliancy
- SCPI “look and feel” commands

Full SCPI compliancy requires you to follow the standards set out in the SCPI Standard documentation. JPA-SCPI Parser handles much of this for you. In addition, the standard specifies other requirements you will need to meet in order to claim SCPI compliancy, such as what certain commands should do, what commands to include for certain instrument classes, and user documentation requirements. JPA-SCPI Parser includes 10 templates for designing command sets for the most popular SCPI instrument classes. A base template is also included that contains all the commands required by any instrument claiming SCPI compliancy.

Often, full SCPI compliancy is not required. Instead, by giving the user the “look and feel” of SCPI, the user will be immediately at home with their new piece of equipment. This approach is extremely common amongst instrument manufacturers, and JPA-SCPI Parser can be used for this too. For instance, you may wish to leave out some of the more obscure commands defined in the SCPI Standard. You may also decide to add command keywords

that better suit your equipment. Whatever you choose you can implement it with JPA-SCPI Parser.

> If SCPI compliance is a required goal then you must read the relevant sections of the SCPI Standard document. SCPI compliancy not only requires a parser able to interpret SCPI-style commands, it also requires other standards to be met such as the types of commands included and the responses to commands returned. This information is included in the SCPI Standard document, available free-of-charge by download (see *"5.1 SCPI Standard"*).

2.3 Using JPA-SCPI Parser

JPA-SCPI Parser allows your own software development team to concentrate on the specifics of your instrument without having to spend valuable time implementing a SCPI command parser.

A simple-to-use pair of files *cmds.c* and *cmds.h* is used to specify your command set and templates of these files are provided to speed up your development. Access Functions allow your code to use JPA-SCPI Parser easily. Full documentation provided in this manual give you all the information you require to get your instrument *"talking SCPI today"*.

2.4 Aims of JPA-SCPI Parser

While developing JPA-SCPI Parser, we always had these aims in mind, and we believe we have satisfied them:

- Support all the common requirements of SCPI-compatible instruments
- Run on almost any processor/hardware platform:
    - Written in ANSI/ISO C
    - Minimal ROM and, particularly, RAM requirements
- Fast and easy to deploy

2.5 Contacting Us

For technical support, please email us at: support@jpacsoft.com, or visit our website: http://www.jpacsoft.com where a technical support request can be submitted.

We are always glad to hear from our customers with their experiences of JPA-SCPI Parser, whether good or bad, and how they would like to see JPA-SCPI Parser improved or expanded. Please contact us at: feedback@jpacsoft.com.

3 What's Included?

3.1 Documentation

- Readme.txt – Release notes for this version
- User Manual – this document
- Design Notes – description of the JPA-SCPI Parser design and structure

3.2 Source Code

- JPA-SCPI Parser modules – *scpi.c* and *scpi.h*
- Sample SCPI command set – *cmds.c* and *cmds.h*
- 10 x SCPI instrument class templates
- SCPI base class template

3.3 Important Note – Text Formats

Different types of computer terminate lines of ASCII text in different ways:

- PCs running Windows or DOS use a Carriage Return & Linefeed combination
- Unix-based computers (including PCs running Linux) use a Linefeed
- Macintoshes use a Carriage Return

We therefore supply 3 variants of each piece of source code. Each variant is stored in a sub-folder to identify it, one of:

- **pc**
- **mac**
- **unix**

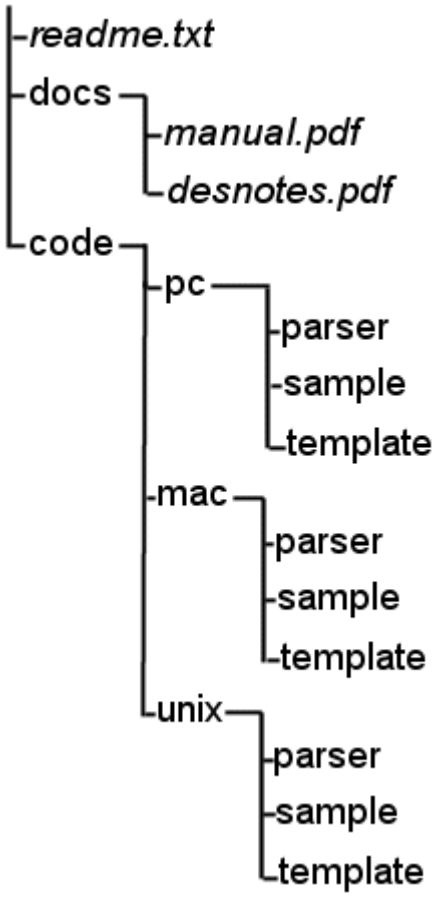
When you are accessing the source code it will save you time if you select the files in the sub-folder most appropriate to your computer.

Apart from these line-termination differences, the source code files are identical across the 3 variants.

3.4 Organization of Supplied Files

The files supplied are organized in a hierarchy of folders (also known as directories).

The folder structure is shown here:



readme.txt contains the latest release notes relating to this version of JPA-SCPI Parser.

The **docs** folder contains JPA-SCPI Parser documentation: ***manual.pdf*** is this *User Manual*. ***desnotes.pdf*** is the *Design Notes* document.

The **code** folder contains all the source code supplied. Immediately beneath this are 3 folders: **pc**, **mac** and **unix**. These folders contain the source code files in the 3 text formats, as discussed on page 17. Wherever you see folder **{format}** in this manual, substitute **pc**, **mac** or **unix**, according to which text file format you require for your development system.

Under folder **{format}**, is folder **parser**, containing the *scpi.c* and *scpi.h* modules that make up the parser code itself. Also under **{format}** is folder **sample**. This contains sample code illustrating some of the command specifications you may require in your own code.

The last folder under **{format}** is **template**. This folder contains the 10 SCPI Instrument Class templates, and the SCPI Base Class template. Each template has its own folder. The names of the template folders are given in Appendix C.

3.5 Notes on the Source Code

All the source code files supplied are intended for reading with a tab spacing of **2**. Use this tab spacing when viewing or editing the files. This is particularly important while you are viewing/editing any of the *cmds.c* files. *cmds.c* is structured as a series of tables. Using the correct tab spacing for this file will mean that columns of the tables line up correctly. This makes understanding the code a lot easier.

4 Overview of JPA-SCPI Parser

The JPA-SCPI Parser source code is divided into two parts:

- The SCPI parser itself
- The command specifications

4.1 SCPI Parser

The actual SCPI parser is contained within the *scpi.c* and *scpi.h* files. These comprise the functions, structures, variables and constants used to parse the SCPI commands of your instrument.

You will not normally need to modify the source code in the *scpi.c* and *scpi.h* modules, apart from any minor changes that might be required specific to your C compiler.

The source code includes extensive comments, including descriptions of every parameter and return value for every function. Documentation on the code's structure and design are also given in the *Design Notes* document.

If you decide that you do need to modify the *scpi.c* and *scpi.h* modules in order to meet your requirements, then this is fine. Of course, we are unable to provide support for any non-standard parts of JPA-SCPI Parser.

4.2 Command Specifications

The command specifications define the set of SCPI commands that your instrument supports. The specifications are contained within *cmds.c* and *cmds.h*. It is these files that you will be modifying in order to specify your command set.

Within the command specifications are defined the command keywords, the number and types of parameters accepted by each command, and various attributes of the parameters, such as allowed types of units and default values.

Templates are provided for 10 of the most popular types of SCPI instrument. These provide a base set of command specifications for your instrument. Whether you decide to use one or more of the templates or start your command specifications from scratch, this manual describes how to implement the types of SCPI command you may require.

JPA-SCPI Parser uses constant C structures and arrays to hold the command specifications. By doing this, no initialization function is required, simplifying the use of JPA-SCPI Parser. In addition, RAM space requirements are kept to a minimum, which is often a significant consideration when coding for some embedded platforms.

5 Before You Start

5.1 SCPI Standard

If you are aiming for SCPI compliancy then you will need a copy of the current *SCPI Standard*. Even if you only intend to provide SCPI *look-and-feel* you will find it useful to have a copy of the SCPI Standard to hand. At the time of writing, the standard is available free-of-charge by download from http://www.scpiconsortium.org.

The SCPI Standard includes information on all the most common SCPI commands your instrument may require. It also includes information on SCPI Instrument Classes that will be valuable during the initial stages of your SCPI implementation.

5.1.1 IEEE488.2 Standard

The SCPI Standard bases some of its design on the *IEEE488.2 Standard*. Much of the IEEE488.2 Standard describes low-level specifications such as the representation of numbers. JPA-SCPI Parser deals with these issues itself and so you probably will not need to know the details.

In addition, SCPI includes some *IEEE488.2 Common Commands* such as **`*RST`** and **`*ESE`**. If you wish to implement full SCPI compliancy, you will need to support these and react to the commands in the ways required by IEEE488.2. If this is the case, then you will need a copy of the IEEE488.2 Standard. At the time of writing, the IEEE488.2 standard is available for download or in printed format, both at a cost, from http://www.ansi.org.

5.2 Where Now?

If you are unfamiliar with SCPI, or you'd like to refresh your memory, please spend a few minutes reading Appendix A.

Once you are up-to-speed with SCPI just turn the page and continue reading...

6 Choose your SCPI Instrument Class(es)

The SCPI Standard includes definitions of 13 SCPI Instrument Classes. These classes correspond to different types of instruments including DC Voltmeters, Ohmmeters, power supplies, and RF sources.

Each class specifies a minimum set of commands that instruments compliant to the Instrument Class must support. The aim is to give instruments of the same type a common set of commands and responses, even if the instruments are made by different manufacturers.

You need to decide which SCPI Instrument Class(es), if any, that you want your instrument to belong to. For instance, an instrument such as a DMM (Digital Multimeter) may belong to the Voltmeter (AC & DC), the Ohmmeter (2 and 4-wire), and the Ammeter (AC & DC) Instrument Classes. On the other-hand, a programmable switch may only belong to the Signal Switcher Instrument Class. Other types of instrument may not belong to any of the SCPI Instrument Classes.

SCPI Compliance Needs

An instrument does not need to comply with any SCPI Instrument Class in order to be SCPI-compliant. However, if you want your instrument to comply with a particular SCPI Instrument Class (or Classes), it must support all the commands required by that Instrument Class.

Even if you do not need or want to claim compliance to any SCPI Instrument Class, it is a useful starting point to select which Instrument Classes embody the main functionality of your instrument. This will give you a head start in deciding what commands you need to implement. You may also be able to make use of the SCPI Instrument Class templates supplied with JPA-SCPI Parser, even if you do not need all the functionality of the Instrument Classes.

6.1 SCPI Instrument Classes Introduced

This section describes 10 of the most common SCPI Instrument Classes. A SCPI Instrument Class template is supplied for each of these. Each template includes the commands required for compliance with the Instrument Class.

Those SCPI Instrument Classes are:

- DC Voltmeter
- AC RMS Voltmeter
- Ohmmeter
- 4-wire Ohmmeter
- DC Ammeter
- AC RMS Ammeter
- Power Supply
- Digitizer (e.g. oscilloscope)
- Signal Switcher (e.g. programmable switch)
- RF and Microwave Source

In addition there are 3 other SCPI Instrument Classes defined in the SCPI Standard that are not described here: *Chassis Dynamometer*, *Emissions Cell*, and *Emissions Bench*. If you

think your instrument may comply with any of these Instrument Classes, refer to the SCPI Standard for more information and for the commands supported by those classes.

Read the brief descriptions of the SCPI Instrument Classes below. Refer to the SCPI Standard for more information if you wish. Take note of any Instrument Classes that you want to comply with or will be useful for your instrument.

6.1.1 DC Voltmeter SCPI Instrument Class

A *DC Voltmeter* is defined by the SCPI Standard as an instrument that measures the average voltage level across its inputs at a point in time.

Command subsystems supported by this SCPI Instrument Class include:

SENSe	*Selects measuring function, range and resolution*
INITiate	*Initiates the taking of measurements*
FETCh?	*Retrieves the measurements taken by* **INITiate**
READ?	*Performs the combined action of* **INITiate** *and* **FETCh?** *sequence*
MEASure?	*Performs the combined action of* **SENSe** *and* **READ?** *sequence*
TRIGger	*Configures trigger conditions for initiating the taking of measurements*

Wherever the function of a DC Voltmeter Instrument is used in a SCPI command, it uses the keywords: **VOLTage[:DC]**

For a full list of the commands supported by the template for this SCPI Instrument Class, see Appendix C.

Note: An instrument wishing to comply with the DC Voltmeter SCPI Instrument Class must support the SCPI command **SYSTem:CAPability?** and return an *instrument specifier* that includes **DCVOLTMETER** (see *"7.3.4 Do I Need the Command "SYSTem:CAPability?"* for more information).

6.1.2 AC RMS Voltmeter SCPI Instrument Class

An *AC RMS Voltmeter* is defined by the SCPI Standard as an instrument that measures the root mean square of the voltage level across its inputs at a point in time.

Command subsystems supported by this SCPI Instrument Class include:

SENSe	*Selects measuring function, range and resolution*
INITiate	*Initiates the taking of measurements*
FETCh?	*Retrieves the measurements taken by* **INITiate**
READ?	*Performs the combined action of* **INITiate** *and* **FETCh?** *sequence*
MEASure?	*Performs the combined action of* **SENSe** *and* **READ?** *sequence*
TRIGger	*Configures trigger conditions for initiating the taking of measurements*

Wherever the function of an AC RMS Voltmeter Instrument is used in a SCPI command, it always uses the keywords: **VOLTage:AC**

For a full list of the commands supported by the template for this SCPI Instrument Class, see Appendix C.

Note: An instrument wishing to comply with the AC RMS Voltmeter SCPI Instrument Class must support the SCPI command **SYSTem:CAPability?** and return an *instrument specifier* that includes **ACVOLTMETER** (see *"7.3.4 Do I Need the Command "SYSTem:CAPability?"* for more information).

6.1.3 Ohmmeter SCPI Instrument Class

An *Ohmmeter* is defined by the SCPI Standard as an instrument that measures the resistance across its input terminals at a point in time.

Command subsystems supported by this SCPI Instrument Class include:

SENSe	*Selects measuring function, range and resolution*
INITiate	*Initiates the taking of measurements*
FETCh?	*Retrieves the measurements taken by* **INITiate**
READ?	*Performs the combined action of* **INITiate** *and* **FETCh?** *sequence*
MEASure?	*Performs the combined action of* **SENSe** *and* **READ?** *sequence*
TRIGger	*Configures trigger conditions for initiating the taking of measurements*

Wherever the function of an Ohmmeter Instrument is used in a SCPI command, it always uses the keyword: **RESistance**

For a full list of the commands supported by the template for this SCPI Instrument Class, see Appendix C.

Note: An instrument wishing to comply to the Ohmmeter SCPI Instrument Class must support the SCPI command **SYSTem:CAPability?** and return an *instrument specifier* that includes **OHMMETER** (see *"7.3.4 Do I Need the Command "SYSTem:CAPability?"* for more information).

6.1.4 4-wire Ohmmeter SCPI Instrument Class

A *4-wire Ohmmeter* is defined by the SCPI Standard as an instrument that measures the resistance across two of its input terminals with the assistance of 2 additional wires, at a point in time.

Command subsystems supported by this SCPI Instrument Class include:

SENSe	*Selects measuring function, range and resolution*
INITiate	*Initiates the taking of measurements*
FETCh?	*Retrieves the measurements taken by* **INITiate**
READ?	*Performs the combined action of* **INITiate** *and* **FETCh?** *sequence*
MEASure?	*Performs the combined action of* **SENSe** *and* **READ?** *sequence*
TRIGger	*Configures trigger conditions for initiating the taking of measurements*

Wherever the function of a 4-wire Ohmmeter Instrument is used in a SCPI command, it always uses the keyword: **FRESistance**

For a full list of the commands supported by the template for this SCPI Instrument Class, see Appendix C.

Note: An instrument wishing to comply to the 4-wire Ohmmeter SCPI Instrument Class must support the SCPI command **SYSTem:CAPability?** and return an *instrument specifier* that includes **FOHMMETER** (see *"7.3.4 Do I Need the Command "SYSTem:CAPability?"* for more information).

6.1.5 DC Ammeter SCPI Instrument Class

A *DC Ammeter* is defined by the SCPI Standard as an instrument that measures the average current through its terminals at a point in time.

Command subsystems supported by this SCPI Instrument Class include:

SENSe	*Selects measuring function, range and resolution*
INITiate	*Initiates the taking of measurements*
FETCh?	*Retrieves the measurements taken by* **INITiate**
READ?	*Performs the combined action of* **INITiate** *and* **FETCh?** *sequence*
MEASure?	*Performs the combined action of* **SENSe** *and* **READ?** *sequence*
TRIGger	*Configures trigger conditions for initiating the taking of measurements*

Wherever the function of a DC Ammeter Instrument is used in a SCPI command, it always uses the keyword: **CURRent[:DC]**

For a full list of the commands supported by the template for this SCPI Instrument Class, see Appendix C.

Note: An instrument wishing to comply to the DC Ammeter SCPI Instrument Class must support the SCPI command **SYSTem:CAPability?** and return an *instrument specifier* that includes **DCAMMETER** (see *"7.3.4 Do I Need the Command "SYSTem:CAPability?"* for more information).

6.1.6 AC RMS Ammeter SCPI Instrument Class

An *AC RMS Ammeter* is defined by the SCPI Standard as an instrument that measures the root mean square of the current through its terminals at a point in time.

Command subsystems supported by this SCPI Instrument Class include:

SENSe	*Selects measuring function, range and resolution*
INITiate	*Initiates the taking of measurements*
FETCh?	*Retrieves the measurements taken by* **INITiate**
READ?	*Performs the combined action of* **INITiate** *and* **FETCh?** *sequence*
MEASure?	*Performs the combined action of* **SENSe** *and* **READ?** *sequence*
TRIGger	*Configures trigger conditions for initiating the taking of measurements*

Wherever the function of an AC RMS Ammeter Instrument is used in a SCPI command, it always uses the keyword: **CURRent:AC**

For a full list of the commands supported by the template for this SCPI Instrument Class, see Appendix C.

Note: An instrument wishing to comply to the AC RMS Ammeter SCPI Instrument Class must support the SCPI command **SYSTem:CAPability?** and return an *instrument specifier* that includes **ACAMMETER** (see *"7.3.4 Do I Need the Command "SYSTem:CAPability?"* for more information).

6.1.7 Power Supply SCPI Instrument Class

A *Power Supply* is defined by the SCPI Standard as a basic sourcing instrument. Normally a power supply provides a constant current or voltage to an electrical circuit.

Command subsystems supported by this SCPI Instrument Class include:

OUTPut	*Sets the state of the power supply's output*
[SOURce]	*Sets the output current and/or voltage level. The* **SOURce** *node is the default root node for Power Supply class compliant instruments*

For a full list of the commands supported by the template for this SCPI Instrument Class, see Appendix C.

Note: An instrument wishing to comply to the Power Supply SCPI Instrument Class must support the SCPI command **SYSTem:CAPability?** and return an *instrument specifier* that includes **DCPSUPPLY** (see *"7.3.4 Do I Need the Command "SYSTem:CAPability?"* for more information).

6.1.8 Digitizer SCPI Instrument Class

A *Digitizer* is defined by the SCPI Standard as an instrument that primarily measures voltage waveforms against time, such as an oscilloscope.

Command subsystems supported by this SCPI Instrument Class include:

INPut	*Configures the coupling of the inputs and other settings*
[SENSe]	*Selects the measurement function, range and other parameters. The* **SENSe** *node is the default root node of Digitizer class compliant instruments.*
INITiate	*Initiates the taking of measurements*
TRIGger	*Configures trigger conditions for the taking of measurements*

For a full list of the commands supported by the template for this SCPI Instrument Class, see Appendix C.

Note: An instrument wishing to comply to the Digitizer SCPI Instrument Class must support the SCPI command **SYSTem:CAPability?** and return an *instrument specifier* that includes **DIGITIZER** (see *"7.3.4 Do I Need the Command "SYSTem:CAPability?"* for more information).

6.1.9 Signal Switcher SCPI Instrument Class

A *Signal Switcher* is defined by the SCPI Standard as a basic signal routing instrument. It can make and break signal connections. More advanced instruments can perform signal routing bases on events or pre-programmed sequences.

Command subsystems supported by this SCPI Instrument Class include:

[ROUTe]	*Used to make and break signal connections. The* **ROUTe** *node is the default root node for Signal Switcher class compliant instruments.*

For a full list of the commands supported by the template for this SCPI Instrument Class, see Appendix C.

Note: An instrument wishing to comply to the Signal Switcher SCPI Instrument Class must support the SCPI command **SYSTem:CAPability?** and return an *instrument specifier* that includes **SWITCHER** (see *"7.3.4 Do I Need the Command "SYSTem:CAPability?"* for more information).

6.1.10 RF and Microwave Source SCPI Instrument Class

An *RF or Microwave Source* is defined by the SCPI Standard as a sourcing instrument. It normally produces a sinusoidal output at a constant output level. This class includes signal generators and sweepers. However, function generators, waveform generators, pulse generators and optical sources are not covered by this class.

Command subsystems supported by this SCPI Instrument Class include:

[SOURce]	*Selects the frequency and output level of the signal. The* **SOURce** *node is the default root node for RF and Microwave Source class compliant instruments.*
OUTPut	*Turns output on and off*
UNIT	*Selects the unit of power used to set the output level*

For a full list of the commands supported by the template for this SCPI Instrument Class, see Appendix C.

Note: An instrument wishing to comply to the RF and Microwave Source SCPI Instrument Class must support the SCPI command **SYSTem:CAPability?** and return an *instrument specifier* that includes **RFSOURCE** (see *"7.3.4 Do I Need the Command "SYSTem:CAPability?"* for more information).

7 Define Your Command Set

Before any coding, you need to create a list of command specifications, one for each command to be supported by your instrument. The start of this chapter describes a form of notation that you can use for your list of command specifications. The remaining sections tell you about using commands from SCPI Instrument Classes and adding commands of your own.

7.1 Command Notation

When specifying commands, it is useful to use a standard form of notation. As well as a clear way of defining your commands for your own use, you can include this list in the instrument's user documentation. The SCPI Standard (*Syntax And Style, Section 5 "Notation"*) defines a form of command notation to be used. The form of notation we describe here varies slightly from that form, but is in common use. It can make for more readable command sets, in our opinion. Use whatever form of notation you prefer.

Before we set out the notation conventions, here are some example command specifications written using the notation:

```
CONFigure
  [:SCALar]:RESistance  [<range>|MIN|MAX [,<resolution>|MIN|MAX]]
  [:SCALar]:FRESistance [<range>|MIN|MAX [,<resolution>|MIN|MAX]]
  [:SCALar]:VOLTage:DC  [<range>|MIN|MAX [,<resolution>|MIN|MAX]]

CONFigure?

[SENSe:]
  FUNCtion[:ON] {"VOLTage[:DC]"}
  FUNCtion[:ON]?
  VOLTage:DC:RANGe[:UPPer] {<range>|MIN|MAX}
  VOLTage:DC:RANGe[:UPPer]?
  VOLTage:DC:RANGe:AUTO# {ON|OFF}

TRIGger
  [:SEQuence]:SOURce {BUS|IMMediate|EXTernal#}
  [:SEQuence]:SOURce?
```

7.1.1 Command Keywords

Group each command within the same subsystem together, i.e. each command that has the same root node.

List the root node once, at the start of the subsystem. It should be in the leftmost position. Commands within the subsystem should be listed below this, indented slightly. The root node should not be repeated.

Separate keywords with a colon (`:`). Enclose optional keywords in square brackets (`[,]`). Remember to enclose one of the adjacent colons if necessary.

Indicate the short form of a keyword using uppercase characters and using lowercase for the remaining characters of the keyword.

7.1.2 Numeric Suffices

You will notice that a couple of the command specifications above include the **#** symbol. This is not a literal character, but instead represents the position where a numeric suffix may be entered by the user. A numeric suffix allows the user to specify the number of the channel, trigger source etc. where there is more than one choice. The entry of the numeric suffix by the user is optional – if it is not entered then the value 1 is used[1].

Note: The **#** symbol can be used in the specification in either the command keywords or in a Character Data parameter entry.

7.1.3 Parameters

Following the command keywords, list the parameters. The parameters are separated from the command keywords by one of more spaces. Each parameter is separated from the other parameters by a comma (**,**).

Boolean parameters should be represented as **{OFF|ON}**. Indicate the default value, if any, using bold type or underline.

For parameters that allow Character Data entries (i.e. mnemonics such as *MAXimum*, *DEFault*, etc.), use curly brackets (**{**,**}**) to enclose the various choices of mnemonics. Separate each choice using a pipe character (**|**). Represent the long and short form of the mnemonics using upper and lower case characters (as for command keywords). Indicate the default value, if any, by using bold type or underline. Optional characters within the Character Data entry should be enclosed inside square (**[**,**]**) brackets.

Numeric Value parameters are indicated by their name enclosed in angled brackets (**<**,**>**) The name should inform the user what the value represents, e.g. **<range>**.

String parameters are again represented by a name within angled brackets. Make it clear that the parameter is a string, e.g. **<message string>**.

Indicate optional parameters by enclosing them in square brackets (**[**,**]**). You can nest square brackets if you wish.

7.2 Base Command Set

Instruments that require SCPI compliancy must support a set of *base SCPI commands*. JPA-SCPI Parser provides a SCPI Base Class template that includes all these base commands.

If you are seeking SCPI compliancy then begin by copying all the commands listed in the SCPI Base Class template (see *"C.11 SCPI Base Class"*) into your list of command specifications.

Even if you do not require SCPI compliancy, you may want to look through the Base Class template and copy any commands you wish to use. For instance, the Base Class template includes commands for querying errors that are useful to most units.

[1] A default value of 1 is the SCPI standard. If you would like to use a different default value then see section *12.3.3 Default Numeric Suffix*.

SCPI Compliance Needs – Base Commands

To claim SCPI-compliance you must support all the base SCPI commands. These commands are included in the SCPI Base Class template. If you remove any of these commands from your command set, your instrument is not SCPI-compliant.

7.3 SCPI Instrument Class Commands

Having looked through the SCPI Instrument Classes in the previous chapter, you will know which SCPI Instrument Classes that you either want to comply with or just want to use when defining your instrument's command set.

If you do not wish to use any SCPI Instrument Classes then skip to *"7.4 Adding Your Own Commands"*.

7.3.1 Using One or More SCPI Instrument Classes with a Template

If you are using any of the 10 SCPI Instrument Classes for which there is a template, then the commands included in the template are listed in Appendix C. Add all the commands shown to your command specifications list. Do not include commands that are duplicates if you are using more than one template.

Note, if you are using 2 or more of the *meter* classes, e.g. DC Voltmeter, Ohmmeter, then there is one command that requires special attention:

```
SENSe:FUNCtion[:ON] {<function list>}
```

This command is used to select the measurement function of the meter. The parameter **<function list>** needs to include all possible functions. Each function should be inside double-quotes. For example, if you are combining the DC Voltmeter, AC Voltmeter, DC Ammeter and AC Ammeter templates, then the command specification should be:

```
SENSe:FUNCtion[:ON]
          {"VOLTage:DC"|"VOLTage:AC"|"CURRent:DC"|"CURRent:AC"}
```

Note: the order of the functions in the list does not matter.

7.3.2 Using a SCPI Instrument Class without a Template

If you are using any of the SCPI Instrument Classes that does not have a template (*Chassis Dynamometer*, *Emissions Bench* or *Emissions Cell*), then you will need to refer to the SCPI Standard for a list of commands required by those Instrument Classes. Add these commands to your command specifications list.

SCPI Compliance Needs – Instrument Classes

If you do not support a command required by a particular SCPI Instrument Class then you cannot claim compliancy to that SCPI Instrument Class. However, this is not a requirement for base SCPI compliancy.

7.3.3 Optional Commands

The SCPI Standard lists commands that can be optionally included for some of the SCPI Instrument Classes. Refer to the SCPI Standard to see which optional commands exist for your chosen Instrument Class(es) and add any your require to your command specification list.

7.3.4 Do I Need the Command "SYSTem:CAPability?"

This command is used to query which SCPI Instrument Classes are supported by an instrument. You need to support this command if you want SCPI compliance to any of the SCPI Instrument Classes.

The response to the command is a string of *instrument identifiers*. There is one instrument identifier per SCPI Instrument Class. The previous chapter gave the instrument identifiers required for each SCPI Instrument Class template; instrument identifiers for the other SCPI Instrument Classes are given in the SCPI Standard. There are also instrument identifiers for other attributes of the instrument's capabilities. For more information on the **`SYSTem:CAPability?`** command, refer to the SCPI Standard.

SCPI Compliance Needs – `SYSTem:CAPability`

An instrument that claims compliancy to one or more SCPI Instrument Classes must support **`SYSTem:CAPability?`**. It is not required for base SCPI compliancy.

7.4 Adding Your Own Commands

You may now want to add commands of your own. If you are aiming for SCPI compliance, then refer to the SCPI Standard when deciding the format of the command and the keywords to use – many commands are already defined in the SCPI Standard and should be used wherever possible. Commands that do not already exist in the SCPI Standard should follow the basic guidelines described in the SCPI Standard. This will help to give your instrument a recognizable feel to operators familiar with SCPI.

Whatever commands you decide to include, add each one to your command specifications list.

8 An Overview of the Required Coding

You should now have a complete set of commands that you wish to support, so you can begin coding. The job of coding is made up of two parts:

- Specifying the command set in code
- Integrating calls to JPA-SCPI Parser Access Functions into your own code

Those tasks are dealt with step-by-step in the following chapters. First of all, here is an overview of what is required.

8.1 Command Specifications

The command specifications and related specifications are all contained in the *cmds.c* and *cmds.h* files. These modules contain structures, arrays and enumerated types that define the commands supported and the types of parameter allowed for each command.

Templates are included to give you a head start when defining your own *cmds.c* and *cmds.h* files. The first task is to choose which template (or templates) to use and to use them as a basis for your own files. Once that is done, you will add your own command specifications.

A command specification comprises two parts:

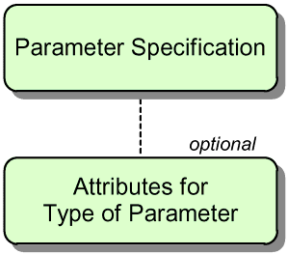
1. Command keywords, e.g. **`SOURce:VOLTage`**
2. Parameter specifications, e.g. **`{<Volts>|MAXimum|MINimum}`**

Command keywords are simply defined as strings – one string is required per command supported. A constant array of strings is used to hold the command keywords for each command specification.

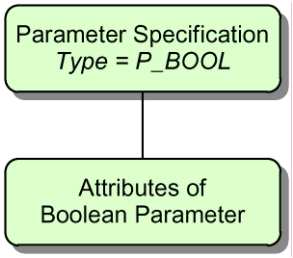
Parameter specifications are more complex, in that they have a basic type, such as Numeric Value, String, etc., and, optionally, attributes associated with that type of parameter. For instance, Numeric Values can have a set of allowed units, Boolean parameters can have a default value, etc.

In addition, some parameter specifications can allow two different types of parameter – in the example **`{<Volts>|MAXimum|MINimum}`**, the parameter may be entered as a number of volts or as a mnemonic (**`MAX`**, **`MAXIMUM`**, **`MIN`** or **`MINIMUM`**).

In order to define a parameter specification, JPA-SCPI Parser uses a hierarchical tree structure. In its general case, a parameter specification takes this form:

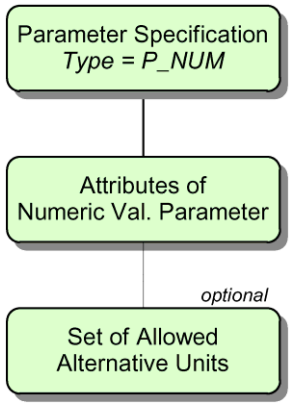


Depending on the type of parameter allowed, there may be an attribute structure present. A Boolean parameter specification has this form:



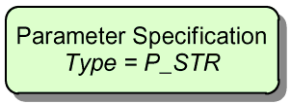
The parameter specification includes the fact that the type of parameter is *P_BOOL* (Boolean). JPA-SCPI Parser knows that this type of parameter has further attributes defined in a Boolean Parameter Attributes structure. A pointer in the parameter specification defines which Boolean Parameter Attributes structure is used.

The parameter specification of a Numeric Value takes this form:

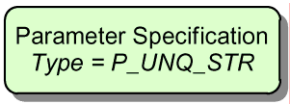


Here, the parameter specification has type *P_NUM* (Numeric Value). This tells the parser that further attributes will be found in a Numeric Value Parameter Attributes structure, pointed to within the parameter specification. In this case, the Numeric Value Parameter Attributes can also optionally include a set of allowed alternative units – this is discussed in more detail in the following chapters.

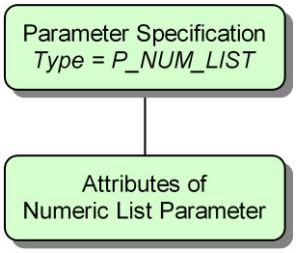
String parameter specifications do not have any other attributes. Their parameter specification is simply:



Similarly, the parameter specification for an Unquoted String is just:

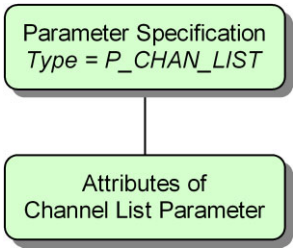


The parameter specification of a Numeric List takes this form:



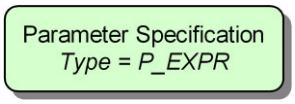
The specification has type *P_NUM_LIST* (Numeric List). From this information, the parser knows that other attributes will be found in a Numeric List Parameter Attributes structure, pointed to within the parameter specification.

Similarly, the parameter specification of a Channel List has this form:



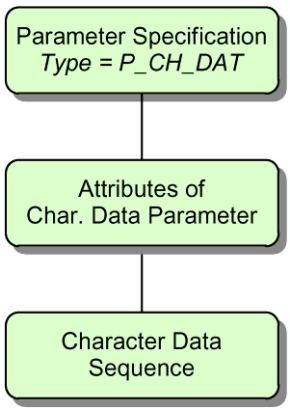
The parser sees that the parameter specification is type *P_CHAN_LIST* (Channel List). It therefore knows that further attributes will be found in a Channel List Parameter Attributes structure pointed to within the parameter specification.

The parameter specification of an Expression takes this simple form:



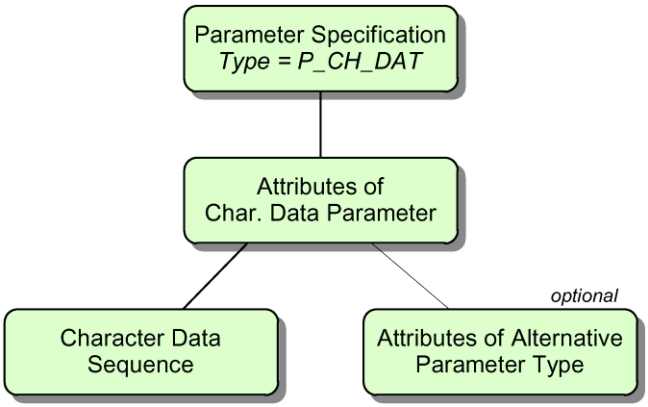
There are no additional attributes of an Expression parameter specification.

The final type of parameter is Character Data – discussed in more detail later. For now, an example of character data is **`MINimum|MAXimum|DEFault`**. The parameter specification for Character Data is:



In addition, a parameter specification can be defined so that it allows either Character Data or another type of parameter, e.g. **{<Volts>|MINimum|MAXimum}** allows a number of volts or a mnemonic to be entered. In this case the parameter specification is the same as Character Data, except that an alternative parameter type is defined. If that parameter type is Boolean or Numeric Value, then it requires attributes.

The form of a parameter specification that allows either Character Data or another parameter type is this:



This has been a very brief overview of how command specifications are defined in JPA-SCPI Parser.

The next few chapters will take you through how to define the structures for your command set. We follow a "top-down" approach. That is, we take each command of your command set in turn and define whatever structures and sub-structures are needed to implement the command's specification. Any command specifications that have the same requirements in terms of parameters simply re-use the existing structures. By following this approach, you will come across all the types of command and parameter specifications required to implement your command set.

8.2 Integrating into Your Own Code

Once you have defined the command specifications, you will want to integrate JPA_SCPI Parser into your own code so that your instrument can understand the commands being sent to it. JPA-SCPI Parser provides a set of Access Functions for use in your own code. These allow you to:

- Parse commands sent to your instrument, returning either the number of the matching command specification, or an error code indicating what was wrong, e.g. invalid parameter, wrong number of parameters, invalid command etc.
- Convert parameters sent in the command into standard C variable types, e.g. integer, unsigned integer, long integer, double, string etc.

Now you have an overview of the tasks ahead it is time to start coding! The following chapters take you step-by-step through everything you need to do.

9 Starting Your Implementation

9.1 Select Your Templates

As discussed previously, JPA-SCPI Parser includes templates for 10 of the SCPI Instrument Classes and also a SCPI Base Class template that provides the commands required for base SCPI compliancy.

The first task is to decide which template, or templates you need – you always start with at least one template. The template(s) you will use depend on which SCPI Instrument Classes you have decided on.

Look at the table below. The left-hand column contains the different possibilities of SCPI Instrument Class(es) you have chosen. Find the row that matches your situation. The right-hand column tells you which templates to use.

SCPI Instrument Class(es) Chosen	**Template(s) to Use**
No SCPI Instrument Classes.	Use: **SCPI Base Class**
One SCPI Instrument Class. It has a template.	Use the template for the SCPI Instrument Class
One SCPI Instrument Class. It does not have a template.	Use: **SCPI Base Class**
Two or more SCPI Instrument Classes. None of them have templates	Use: **SCPI Base Class**
Two or more SCPI Instrument Classes. Only one has a template.	Use the template for the SCPI Instrument Class that has a template
Two or more SCPI Instrument Classes. Two or more of them have templates.	Use all the templates for the SCPI Instrument Classes chosen.

Now that you know which template(s) to use, follow either the instructions in section *"9.2 Using a Single Template"* or section *"9.3 Using Two or More Templates"* as appropriate.

9.2 Using a Single Template

If you are using a single SCPI Instrument Class template supplied with JPA-SCPI Parser, or you are just using the SCPI Base Class template, follow these steps.

9.2.1 Copy the Template into your Project Folder

A template comprises a pair of files: *cmds.c* and *cmds.h*. Find your template in Appendix C. It tells you which folder those files are located in.

Copy the pair of template files into your project folder – leave the originals in their template folder as you may require them in the future for another instrument.

9.2.2 Customize cmds.c

Open the copy of *cmds.c* in your project folder. The first few lines of comments at the top of the file tell you the name of the template and also the revision history of the *cmds.c* module. *cmds.c* will become the main file in your project where your command set is specified and will be bespoke for your instrument. So delete those first lines of comments about the template (and insert your own remarks, if you wish).

Now skip forward to section *"9.4 Tidying Up"*.

9.3 Using Two or More Templates

You may want to use 2 or more of the SCPI Instrument Class templates supplied. Remember that you never need to include the SCPI Base Class template with any other template, since all templates include all the commands in the SCPI Base Class template.

If you do want to use 2 or more templates then follow these steps.

9.3.1 Copy one of the Templates into your Project Folder

It does not matter which of your templates you start with. Look up one of your templates in Appendix C and find out which folder it is in. Locate that folder and copy the *cmds.c* and *cmds.h* files into your project folder.

9.3.2 Customize cmds.c

Open the copy of *cmds.c* in your project folder. The first few lines of comments at the top of the file tell you the name of the template and also the revision history of the *cmds.c* module. *cmds.c* will become the main file in your project where your command set is specified and will be bespoke for your instrument. So delete those first lines of comments about the template (and insert your own remarks, if you wish).

9.3.3 Merge Other Templates into cmds.c

To make things quicker, all the *cmds.h* files are identical for every template. You only need to merge the *cmds.c* files together from your templates.

Taking each one of your other templates in turn, locate the *cmds.c* file for that template (look up it's location in Appendix C) and open it (read-only mode – you do not want to change its contents). We will refer to this file in the instructions below as the *template*.

At the same time, open your own copy of *cmds.c* in your project folder for editing. We will refer to this file in the instructions below simply as *cmds.c*.

We will now copy elements from the template into your copy of *cmds.c*. For the moment we will not worry about what each element of *cmds.c* is used for – this is dealt with in later chapters.

9.3.3.1 Alternative Units

Skip forward in both *cmds.c* and the template until you find the section of code titled *"Alternative Units"*. In this section are located one or two lines starting with **`ALT_UNITS_LIST`**. If the template contains any such lines that are not present in *cmds.c* then copy them into *cmds.c* in this section. The order of the lines does not matter.

For example, if you are copying from the *RF and Microware Source* template, you will need to copy this line from that template into *cmds.c*:

```
  ALT_UNITS_LIST   eAltPower[] = {U_VOLT, U_WATT, U_DB_W, U_END}; /* Volts, Watts,
dbW   */
```

9.3.3.2 Numeric Value Types

Locate the sections of *cmds.c* and the template that are titled *"Numeric Value Types"*. Are there any lines in the template that are not present in *cmds.c* in this section? If so, copy the line from the template into *cmds.c*. The order of the lines does not matter. For instance, if

your template is *RF and Microwave Source*, then you may need to copy this line into *cmds.c*:

```
NUM_TYPE  sPower    = { U_NONE,   eAltPower,      0   }; /* All Power Units    */
```

9.3.3.3 Character Data Sequences

Skip forward in both *cmds.c* and the template to the next section of code: *"Character Data Sequences"*.

Again, compare the lines in both files. If any of the lines in the template are not present in *cmds.c*, then copy the line into *cmds.c*. The order of the lines does not matter.

For example, if you are copying from the template *Digitizer*, you may need to copy these lines into *cmds.c*:

```
CHDAT_SEQ SeqACDCGnd[]          = "AC|DC|GND";
CHDAT_SEQ SeqACDC[]             = "AC|DC";
CHDAT_SEQ SeqXTIMe[]            = "\"XTIMe:VOLTage#[:DC]\"";
CHDAT_SEQ SeqAscii[]            = "ASCii";
CHDAT_SEQ SeqPosNegEit[]        = "POSitive|NEGative|EIther";
CHDAT_SEQ SeqInternal[]         = "INTernal#";
```

Note that, if you are using more than one *meter* template, e.g. *DC Voltmeter*, *AC Ammeter*, *Ohmmeter* etc. then each template will contain an entry in this section called **`SeqFunctions[]`**. This is a special case, since the contents of this entry is different in each template.

To merge two copies of **`SeqFunctions[]`**, append the contents of the template to the existing contents of **`SeqFunctions[]`** in *cmds.c*, separating each entry with a pipe (|) character. For instance, if your *cmds.c* entry is:

```
CHDAT_SEQ SeqFunctions[]        = "\"VOLTage:DC\"";
```

and you are merging with the AC Voltmeter template, you should change the entry in *cmds.c* to be:

```
CHDAT_SEQ SeqFunctions[]        = "\"VOLTage:DC\"|\"VOLTage:AC\"";
```

Of course, if you are merging more than 2 digital meter templates, then this list will contain more entries.

9.3.3.4 Character Data Types

Locate the next section in the template and *cmds.c* entitled *"Character Data Types"*. Once more compare the entries in the two files. If there are any entries in the template that are not present in *cmds.c* then copy those entries into *cmds.c*. The order of the entries does not matter.

As an example, if you are copying from template *Digitizer*, you will need to copy these entries into your *cmds.c* file:

```
CHDAT_TYPE  sACDCGnd    = { SeqACDCGnd,   NO_DEF, ALT_NONE };    /* AC|DC|GND */
CHDAT_TYPE  sACDC       = { SeqACDC,      NO_DEF, ALT_NONE };    /* AC|DC     */
CHDAT_TYPE  sXTIMe      = { SeqXTIMe,     NO_DEF, ALT_NONE };
                                                    /* "XTIMe:VOLTage#[:DC]" */
CHDAT_TYPE  sASCii      = { SeqAscii,     NO_DEF, ALT_NONE };    /* ASCii     */
CHDAT_TYPE  sPosNegEit  = { SeqPosNegEit, NO_DEF, ALT_NONE };
                                                    /* POSitive|NEGative|EIther */
CHDAT_TYPE  sInternal   = { SeqInternal,  NO_DEF, ALT_NONE };
                                                    /* INTernal# */
```

9.3.3.5 Numeric List Types

The next section in the template and *cmds.c* is entitled *"Numeric List Types"*. Compare the entries in the two files. If there are any entries in the template that are not present in *cmds.c* then copy those entries into *cmds.c*. The order of the entries does not matter.

9.3.3.6 Channel List Types

The next section in the template and *cmds.c* is entitled *"Channel List Types"*. Compare the entries in the two files. If there are any entries in the template that are not present in *cmds.c* then copy those entries into *cmds.c*. The order of the entries does not matter.

9.3.3.7 Command Specs – Part 1: Command Keywords

Locate the sections of code in the template and *cmds.c* file titled *"Command Specs – Part 1: Command Keywords"*. Each line corresponds to a command supported.

Carefully compare the lines in the template with the lines in *cmds.c*. If there are any lines in the template that are not present in *cmds.c* **and you wish to support that command**, then copy it into *cmds.c*.

Commands in this section of code are grouped into subsystems, i.e. commands with the same root node. This is only for readability, but it may be best to maintain this organization. When copying entries from the template into *cmds.c*, insert the line so that it is within the group of commands in the same subsystem. For example, if *cmds.c* was originally the *DC Voltmeter* template, and you are copying from the *Ohmmeter* template, these are some of the lines you will need to copy:

```
"SENSe:RESistance:RANGe[:UPPer]",                    /*    34        */
"SENSe:RESistance:RANGe[:UPPer]?",                   /*    35        */
"SENSe:RESistance:RANGe:AUTO",                       /*    36        */
"SENSe:RESistance:RANGe:AUTO?",                      /*    37        */
"SENSe:RESistance:RESolution",                       /*    38        */
"SENSe:RESistance:RESolution?",                      /*    39        */
```

After copying these lines into *cmds.c*, entries for the **SENSe** subsystem would look like this:

```
"SENSe:FUNCtion[:ON]",                               /*    32        */
"SENSe:FUNCtion[:ON]?",                              /*    33        */
"SENSe:VOLTage:DC:RANGe[:UPPer]",                    /*    34        */
"SENSe:VOLTage:DC:RANGe[:UPPer]?",                   /*    35        */
"SENSe:VOLTage:DC:RANGe:AUTO",                       /*    36        */
"SENSe:VOLTage:DC:RANGe:AUTO?",                      /*    37        */
"SENSe:VOLTage:DC:RESolution",                       /*    38        */
"SENSe:VOLTage:DC:RESolution?",                      /*    39        */
"SENSe:RESistance:RANGe[:UPPer]",                    /*    34        */
"SENSe:RESistance:RANGe[:UPPer]?",                   /*    35        */
"SENSe:RESistance:RANGe:AUTO",                       /*    36        */
"SENSe:RESistance:RANGe:AUTO?",                      /*    37        */
"SENSe:RESistance:RESolution",                       /*    38        */
"SENSe:RESistance:RESolution?",                      /*    39        */
```

Allowing Numeric Suffices

The digitizer template, for instance, already allows a numeric suffix in its **SENSe** subsystem commands. You can allow one or more numeric suffices in whichever commands you need. To do so, insert a '**#**' character in the command keywords specification wherever a numeric suffix is to be entered.

Don't worry about the numbering in the comments for now.

If instead you are copying a command from the template that belongs to a subsystem not yet present in *cmds.c* then insert it wherever you prefer. It is a good idea to separate commands from different subsystems with a blank line for readability.

Whenever you add a line to this section of code you must also add a line to the section of code called *"Command Specs – Part 2: Parameters".*

9.3.3.8 Command Specs – Part 2: Parameters

Each line in this section of code corresponds to a line in the section *"Commands Specs – Part 1: Command Keywords".*

- There must be the same number of lines in both sections.
- Each line in *Part 1: Command Keywords* corresponds to the same line number in *Part 2: Parameters* – the order of the entries in the two parts must be the same.

So for each line you have added into *cmds.c*, locate the corresponding line in the template in *Command Specs – Part 2: Parameters*; use the comments beside each line to help you – the command numbers and command syntax will match. Copy this line into *cmds.c* so that the order of the lines in *Part 1* and *Part 2 of Command Specs* in *cmds.c* is the same.

For instance, say you have added the last line (highlighted) of these 3 lines to *"Command Specs – Part 1"* in *cmds.c*:

```
"SENSe:VOLTage:DC:RESolution",                          /*    38        */
"SENSe:VOLTage:DC:RESolution?",                         /*    39        */
"SENSe:RESistance:RANGe[:UPPer]",                       /*    34        */
```

You therefore need to copy the corresponding entry into *"Command Specs – Part 2"* in *cmds.c* from the template, and insert it in the same order:

```
{{  { REQ  CH_DAT  sMinMaxVolts },{ NOP  } }}, /* 38  :VOLTage:DC:RESolution   */
                                               /*       {<resolution>|MIN|MAX} */
{{  NO_PARAMS                              }}, /* 39  :VOLTage:DC:RESolution?  */
{{  { REQ  CH_DAT  sMinMaxOhms  },{ NOP  } }}, /* 34  :RESistance:RANGe[:UPPer]*/
                                               /*       {<range>|MIN|MAX}      */
```

9.4 Tidying Up

You will now have a copy of *cmds.c* that contains all the commands from each of the templates you are using. It may also contain commands that you do not wish to support.

9.4.1 Remove Unwanted Commands

Look through your *cmds.c* file at the section of code *"Command Specs – Part 1: Keywords"*. If there are any commands that you do not wish to support then:

- Remove the line from *"Command Specs – Part 1"*
- Locate the corresponding line in *"Command Specs – Part 2"* and remove it also

9.4.2 Renumber Commands

If you have removed any unwanted commands from *cmds.c* or you used 2 or more templates, then you will need to follow the steps described here.

Go back to the top of *"Command Specs – Part 1: Command Keywords"* in your *cmds.c* file. Each entry has a comment on the right-hand side that gives the command number.

Renumber these command numbers in the comments as required, so that the top command is numbered 0, the command below it is 1, and so on for all the commands. Accurate command numbering is vital for a correct implementation of the parser.

Now go to the section of code in *cmds.c* titled *"Command Specs – Part 2: Parameters"*. Again, each entry here has a comment on the right-hand side contain the command number and also the syntax of the command it represents.

Again, renumber the commands in the comments so that the top entry has command number 0, the next one is 1, etc.

It is vital that the command specifications in Part 1 and Part 2 of *cmds.c* match up. Make sure that:

- The last command number in both parts is the same – this tells you there are the same number of entries
- The order of commands in both parts are the same. Have both sections of code viewable at once and check that the first line in both sections correspond to the same command syntax. Repeat this for each line in turn.

10 Specify Maximum Number of Parameters

What is the maximum number of parameters that any of your commands can accept? By default, the templates are all configured so commands can accept a maximum of 2 parameters per command.

If the maximum number of parameters accepted by any of your commands is 2 (or 1) then you do not need to make any modifications. Skip forward to the next chapter. Otherwise, follow the steps described here.

10.1 Set Maximum Parameters in cmds.h

Open your copy of the *cmds.h* header file. Locate the section of code entitled *"Maximum Parameters"*. There is a single definition within this section. It looks like this:

```
/*****************************************************************************/
/* Maximum Parameters                                                        */
/* ------------------                                                        */
/* USER: Modify this value to be equal to the maximum number of parameter accepted    */
/*       by any of the supported Command Specs                               */
#define MAX_PARAMS              (2)       /* Most params accepted by any command    */
/*****************************************************************************/
```

Change the definition of **MAX_PARAMS** from the value **(2)** to whatever the maximum number of parameters you require. Save and close *cmds.h*.

10.2 Modify cmds.c for Maximum Parameters

10.2.1 NO_PARAMS

Now open your copy of the file *cmds.c*. Find the section of code headed *"More definitions for use in the Command Spec Table"*. There is a definition of **NO_PARAMS**. It looks like this:

```
#define NO_PARAMS    {NOP},{NOP}
```

NO_PARAMS is used as shorthand for commands that do not take any parameters. It replaces all the columns of parameter information with the single symbol **NO_PARAMS**.

Change the definition of **NO_PARAMS** so that it comprises the same number of **{NOP}** items as the maximum number of parameters.

For example, if you have defined **MAX_PARAMS** as **(4)**, then use this definition:

```
#define NO_PARAMS    {NOP},{NOP},{NOP},{NOP}
```

> If a few of your commands can accept many parameters then refer to *"17.4 Commands that allow Many Parameters"* for tips.

10.2.2 Command Specifications

The other modification you need to make to *cmds.c* is in the section of code entitled *"Command Specs - Part 2: Parameters"*. This is where the specifications of each command's parameters are kept. The code looks similar to this:

```
/*****************************************************************************/
/* Command Specs - Part 2: Parameters                                        */
/* ----------------------------------                                        */
/* USER: Include all the Command Specs supported                             */
/* Notes:                                                                    */
/*    a) Each line in this table corresponds to the line in the Command Spec Command  */
/*       Keyword table with the same index. There must be the same number of entries  */
/*       in both tables.                                                     */
/*****************************************************************************/
const struct strSpecCommand sSpecCommand[] =
{
/*                                                                  C o m m a n d     */
/*  Param 1                      Param 2                           Number  Syntax    */
/*  =======                      =======                           ======  ======    */
/*  Opt/Req Type   Attributes    Opt/Req Type    Attributes                          */
/*  ------- ----   ----------    ------- ----    ----------                          */
/*                                                                                   */

/* Commands required by all SCPI-Compliant Instruments                               */

/* Required IEEE488.2 Common Commands (see SCPI Standard V1999.0 ch4.1.1)            */
{{  NO_PARAMS                                                  }}, /*   0  *CLS        */
{{  { REQ   NUM    sNoUnits    },{ NOP                      } }}, /*   1  *ESE <value>*/
{{  NO_PARAMS                                                  }}, /*   2  *ESE?       */
{{  NO_PARAMS                                                  }}, /*   3  *ESR?       */
{{  NO_PARAMS                                                  }}, /*   4  *IDN?       */
{{  NO_PARAMS                                                  }}, /*   5  *OPC        */
{{  NO_PARAMS                                                  }}, /*   6  *OPC?       */
{{  NO_PARAMS                                                  }}, /*   7  *RST        */
{{  { REQ   NUM    sNoUnits    },{ NOP                      } }}, /*   8  *SRE <value>*/
{{  NO_PARAMS                                                  }}, /*   9  *SRE?       */
```

Commands that allow one or more parameters have a row that takes the format:

{{ {*<parameter1>*} , {*<parameter2>*} }},

For example:

{{ { OPT CH_DAT sMinMaxAmps },{ OPT CH_DAT sMinMaxAmps } }},

If the definition of **MAX_PARAMS** has changed from **2** to another value, then this format needs modifying. If for instance **MAX_PARAMS** is **3** then the format required is now:

{{ {*<parameter1>*} , {*<parameter2>*} , {*<parameter3>*} }},

To make this change, add this text between the **{*<parameter2>*}** item and the closing double curly brackets:

,{NOP}

For example, in the case above the row would become:

{{ {OPT CH_DAT sMinMaxAmps},{OPT CH_DAT sMinMaxAmps},{NOP} }},

If **MAX_PARAMS** was defined as **4**, then you would need to insert:

,{NOP},{NOP}

and so on.

Note that rows that use the **NO_PARAMS** definition do not need modifying, i.e.:

{{ NO_PARAMS }},

Something to bear in mind when you make these modifications is readability. The templates are coded so that the columns of this table line up with the headings above. If you increase the maximum number of parameters, then you should create column headings in the comment lines above the structure, i.e. copy the *"Param2 – Opt/Req, Type, Attributes"* headings for parameter 3 etc. When you insert the extra **{NOP}**, line them up with the columns above. This does mean that the table will get wider and you may need to scroll to see the full width of the lines. This is usually preferable to having columns that do not line up.

11 Specify Supported Units

Before we can begin specifying the supported commands, we need to decide what units are to be supported by your instrument.

Numeric Value parameters often allow units to be specified after the value. For instance:

100V

3.5 MOHM

-56UA

Look through all the parameters used by your commands and make a list of all the different types of unit that are supported, e.g.:*Volts, Amps, Ohms, Hertz, Decibel Watts,* etc.

For the purposes of JPA-SCPI Parser, units are split into two parts:

Base Units	*e.g. Volts, Amps, Ohms, Decibel Watts, Hertz*
Unit Multiplier	*e.g. p, n, u, m, k, MA, G*

Only write down the base units supported at this stage, e.g. do not list *milli-volts* and *kilo-volts* separately, but just write down *Volts*.

When a user enters a Numeric Value as a parameter, JPA-SCPI Parser automatically converts the value, whatever unit multiplier was used, into the base units.

For instance, a user might enter:

1.5UA

JPA-SCPI Parser splits this into the value *1.5* and the Unit String *UA*. It then recognizes *UA* as belonging to the Amps unit family and converts the value to the base units, Amps. When your code calls an Access Function of JPA-SCPI Parser, you will be returned with the value 1.5e-6 (amps). This removes the need for your code to perform the scaling itself.

11.1 Specify Base Units in cmds.h

Open your copy of the *cmds.h* file for editing and locate the section of code entitled *"Base Unit Types"*. It will look like this:

```
/*****************************************************************************/
/* Base Unit Types                                                           */
/* ---------------                                                           */
/* USER: Add Base Unit Types supported by your instrument                    */
/*       Optional: Remove Base Unit Types not supported                      */
/*****************************************************************************/
enum enUnits
{
  U_NONE,               /* USER: Do not modify this line                     */

  U_VOLT,               /* User-modifiable list of supported base unit types */
  U_AMP,
  U_OHM,
  U_WATT,
  U_DB_W,
  U_JOULE,
  U_FARAD,
  U_HENRY,
  U_HERTZ,
  U_SEC,
  U_KELVIN,
  U_CELSIUS,
```

```
    U_FAHREN,

    U_END                   /* USER: Do not modify this line                    */
};
```

Each entry between **U_NONE** and **U_END** represents a type of *base unit*. Take a look at your list of units. Are there any in your list that do not have an entry in the source code? If you do require support for a unit not already present then add an entry for each one to this enumeration. Use the format:

U_*<base unit name>*

Use a meaningful name, e.g. **U_BAR** for Bar (unit of pressure) or **U_GRAM** for Gram (unit of mass).

The order of the entries in this enumeration is unimportant, except that **U_NONE** must be the first value, and **U_END** must be the last.

For now, do not remove any of the types of base unit that are not required. The deletion of unused specifications is dealt with in chapter 15.

Parser Limitations: A maximum of 255 base unit types are allowed.

Now save and close *cmds.h*.

11.2 Specify Supported Units in cmds.c

Return to your *cmds.c* file and locate the section headed *"Unit Specs"*. This section contains all of the *unit strings* recognized by the parser and also defines which unit family and what multiplier each unit string represents.

Take a look at the lines defining the Voltage unit strings:

```
/* Volts                                                                    */
  { "NV",     U_VOLT,       -9 },   /* NanoVolt                           */
  { "UV",     U_VOLT,       -6 },   /* MicroVolt                          */
  { "MV",     U_VOLT,       -3 },   /* MilliVolt                          */
  { "V",      U_VOLT,       0  },   /* Volt                               */
  { "KV",     U_VOLT,       3  },   /* KiloVolt                           */
  { "MAV",    U_VOLT,       6  },   /* MegaVolt                           */
```

If you have added a new entry (or entries) to the base units enumeration in *cmds.h* then you will need to add some entries here.

Insert a line before the **END_OF_UNITS** line at the bottom of this section of source code.

Each type of unit has one or more rows in this table. Each row takes the format:

{ "*<unit string>*", *<base units>*, *<units multiplier>* },

Create new rows as you require before the **END_OF_UNITS** line. The elements of each row are described here.

11.2.1 Unit String

The *unit string* is the sequence of characters that the parser will recognize as the units component of a Numeric Value parameter. It comprises an optional *unit multiplier* followed by the base unit string, e.g.:

> The unit string *MV* is made up of *M* the unit multiplier (meaning *milli*, or 1e-3) and *V* (meaning volts) the base unit string.

Some requirements of the unit string:

1. Every unit string must be unique.
2. Do not include spaces within the string
3. Unit strings do not support long and short forms – all characters of the unit string must be entered, so specify unit string using uppercase characters only.
4. When choosing unit multiplier characters, you should use those defined in the IEEE488.2 standard, as used by SCPI:

Unit Multiplier Character(s)	Multiplier
A	1e-18
F	1e-15
P	1e-12
N	1e-9
U	18-6
M	1e-3
K	1e3
MA	1e6
G	1e9
T	1e12
PE	1e15
EX	1e18

   Note that there are a couple of exceptions: when the units are *HZ* or *OHM*, then the unit multiplier characters for 1e6 are *M* or *MA*, e.g. *MHZ* means MegaHertz, and so does MAHZ. If the user needs to specify, say, milliohms, then they can use micro-ohms instead and multiply the value by 1000.

5. When choosing what base unit characters to use, again the IEEE488.2 Standard defines the preferred strings. Use these wherever possible:

Base Unit String	Unit
A	Amp
C	Coulomb
DBW	Decibel Watts[1]
F	Farad
G	Gram
H	Henry
HZ	Hertz
J	Joule
K	Degree Kelvin
CEL	Degree Celsius
FAR	Degree Fahrenheit
L	Liter
LM	Lumen
LX	Lux

[1] *DBUW* represents Decibel MicroWatts, *DBMW* represents Decibel MilliWatts, and so on. Note: *DBM* is also allowed – it is equivalent to DBMW.

M	Meter
N	Newton
OHM	Ohm
PAL	Pascal
RAD	Radian
S	Second
SIE	Siemens
T	Tesla
V	Volt
W	Watt
WB	Weber

6. It is acceptable in JPA-SCPI Parser to define different unit strings that have the same meaning if you wish. For instance, you will see that within the templates that resistance unit strings use *OHM* or *R* as the base unit string. This allows entry of resistance values to be followed by either e.g. *KOHM* or *KR* etc.
7. You may include the following characters within the unit string if required:

/	e.g. *M/S* for Meters per Second
.	e.g. *N.M* for Newton Meters
digits	e.g. *M/S2* for Meters per Second^2

11.2.2 Base Units

The *base units* parameter defines which base units the entry belongs to. This must be one of the base unit types defined in the enumeration in *cmds.h*. For example, if you have added the enumeration type **`U_GRAM`** to the types in *cmds.h*, then you will use **`U_GRAM`** as your base units for entries that represent grams, kilograms, etc.

11.2.3 Units Multiplier

The *units multiplier* specifies how numbers entered with the units string are to be multiplied in order to convert them to the base units.

JPA-SCPI Parser uses this formula:

Stored Value = Number Entered x 1e<*units multiplier*>

For example, the unit string **`UV`** is associated with base units **`U_VOLTS`** and has the units multiplier **`-6`**. If a Numeric Value parameter is entered as *100UV*, JPA-SCPI Parser converts this to base units (volts) like this:

Stored Value = 100 x 1e-6

= **0.0001 Volts**

The units multiplier must be an integer between –43 and +43.

11.2.4 Example Entries

For example, say you have added the base unit type **`U_GRAM`** to the enumeration in *cmds.h*. You now wish to allow entries in micrograms, milligrams, grams, and kilograms. You would add these lines to the table in *cmds.c*:

```
/* Grams                                                    */
  { "UG",     U_GRAM,       -6 },   /* MicroGram          */
  { "MG",     U_GRAM,       -3 },   /* MilliGram          */
  { "G",      U_GRAM,       0  },   /* Gram               */
  { "KG",     U_GRAM,       3  },   /* KiloGram           */
```

11.2.5 Expanding Ranges of Supported Units

The templates support a typical range of unit strings, e.g. the base unit Volts is associated with the unit strings *NV*, *UV*, *MV*, *V*, *KV*, *MAV* (Nanovolts to MegaVolts). But what if you want to allow entries in picovolts (*PV*), for instance? This is easily done.

Locate the *Volts* section of the table. Insert this line before the row containing the definitions for unit string **"NV"**:

```
    { "PV",     U_VOLT,       -12 },   /* PicoVolt           */
```

Do this for whatever unit strings you wish to support that are not already present in *cmds.c*. Note that the order of entries in the table is unimportant, but you may wish to group rows for the same units together as we have done in the templates.

Parser Limitations: A maximum of 255 entries are allowed in this table.

12 Optional Support Features

12.1 Introduction to the Optional Support Features

JPA-SCPI Parser includes support for some features that you may or may not require. By disabling support features that you do not need, you will save ROM and RAM space.

The optional support features are:

- **Numeric Suffix** - *Allows you to specify positions in your command keyword and character data specifications where the user can enter a numeric suffix. This feature can be used for instruments that support multiple channels, triggers, etc., to specify command sets without having to create duplicate commands for each channel.*
- **Numeric List parameter type** - *Additional parameter type to allow entry of Numeric List parameters*
- **Channel List parameter type** - *Additional parameter type to allow entry of Channel List parameters*
- **Expression parameter type** - *Additional parameter type to allow entry of Expression parameters*

If you are unsure as to which of these features you need to support, then take a look at *"Appendix A – An Introduction to SCPI"* where you will find more information in the relevant sections. The SCPI Standard also describes them in detail.

In addition, two further options allow you to (a) increase the number of characters that can be entered in an input command line, and (b) increase the number of command definitions that can be supported. These are described further on in this section.

12.2 Enabling/Disabling the Features You Need

By default, all the optional features are supported. If you wish to disable one or more of them, then open *cmds.h*. Locate the section near the top of the file entitled "*Optional Support Features*". It looks like this:

```
/*****************************************************************************/
/* Optional Support Features                                                 */
/* -------------------------                                                 */
/* USER: #define the features that you require and comment out those not required    */
#define SUPPORT_NUM_SUFFIX        /* Numeric Suffix in keywords                      */
#define SUPPORT_NUM_LIST          /* Numeric List parameter type                     */
#define SUPPORT_CHAN_LIST         /* Channel List parameter type                     */
#define SUPPORT_EXPR              /* Expression parameter type                       */
/*****************************************************************************/
```

Each **`#define`** enables one of the optional support features. By commenting out the **#define** lines for the features you do not require, you will disable that feature.

Each of the support features are individually selectable, so you can disable as many as you want in order to save memory.

12.3 Numeric Suffix Support Settings

If you are using the optional Numeric Suffix support feature, then you should now look at the section of code in *cmds.h* entitled "*Numeric Suffix*". It looks like this:

```
#ifdef SUPPORT_NUM_SUFFIX
/*******************************************************************************/
/* Numeric Suffix                                                              */
/* --------------                                                              */
/* (only used if Numeric Suffix support feature is enabled)                    */
/*                                                                             */
/* USER: Modify these values as required. See User Manual for more information.*/
#define MAX_NUM_SUFFIX        (10)          /* Maximum number of numeric suffices    */
                                            /* possible in a single command          */
#define NUM_SUF_MIN_VAL       (1)           /* Minimum value allowed (0 or greater)  */
#define NUM_SUF_MAX_VAL       (UINT_MAX)    /* Maximum value allowed (<=UINT_MAX)    */
#define NUM_SUF_DEFAULT_VAL   (1)           /* Default value if no suffix present    */
/*******************************************************************************/
#endif
```

As you can see, this code is only compiled if the Numeric Suffix support feature is enabled. There are four attributes you can change relating to how numeric suffices work on your system. These are described below.

12.3.1 Maximum Number of Numeric Suffices

Take a look at the set of command specifications you are implementing. What is the maximum number of numeric suffices that can be entered by a user in a single command? Count the number of '**#**' symbols in each command specification. If you have included any '**#**' symbols in any of your character data parameter specifications then you must include these too. For example, look at this command specification:

```
TRIGger:SOURce# {INTernal|EXTernal#}
```

In this case, the user may enter up to 2 numeric suffices.

Whichever of your commands can accept the most numeric suffices, use this number as the maximum allowed. Change the definition of **MAX_NUM_SUFFIX** to be this number, or leave it at the default value of 10, which will nearly always be sufficient. You will save a few bytes of RAM if you reduce this value.

12.3.2 Range of Allowed Numeric Suffices

By default, the parser allows any number between 1 and **UINT_MAX** (e.g. 65535 if unsigned ints use 16 bits) to be entered as a numeric suffix. This is nearly always acceptable, since you can perform your own range checking on the numeric suffices after the parser returns them.

If you wish though, you can change the range of values that are allowed as numeric suffices. Remember that this range checking affects all numeric suffices in your system. If you use numeric suffices for more than one purpose, e.g. output channels and trigger sources, then you may want different range limits to apply in each case. To do this, make the parser's numeric suffix range limits the superset of your requirements, and perform your own range-checking afterwards.

The range of values allowed for numeric suffices by the parser are #defined as **NUM_SUF_MIN_VAL** and **NUM_SUF_MAX_VAL**. Change their definitions to the values you require. Limitations are:

- Each value must be between 0 and **UINT_MAX**
- **NUM_SUF_MAX_VAL** must be greater than (or equal to) **NUM_SUF_MIN_VAL**

12.3.3 Default Numeric Suffix

The SCPI Standard says that if a numeric suffix is not entered then it is equivalent to being entered as 1 (*SCPI Standard V1999.0, 6.2.5.2 "Multiple Identical Capabilities"*). This is how JPA-SCPI Parser operates by default. However, if you want a different default numeric suffix then you can.

The default value is #defined as **`NUM_SUF_DEFAULT_VAL`**. You can make it whatever value you want, even if it is outside the range of **`NUM_SUF_MIN_VAL`** and **`NUM_SUF_MAX_VAL`**. This can be useful, for instance, if you need to discriminate between the user entering, say:

`FETCH?`

and

`FETCH1?`

By default, both entries will return a numeric suffix of 1. You will not be able to tell which version of the command that the user entered. If you do need to know if a numeric suffix was entered or not then change the #define of **`NUM_SUF_DEFAULT_VAL`** to equal 0. Now if the user enters **`FETCH?`**, the numeric suffix returned will be 0, whereas if the user enters **`FETCH1?`**, the numeric suffix returned will be 1.

12.4 Channel List Support Settings

If you are using the optional support feature Channel List parameter type, then take a look at the section of code in *cmds.h* entitled "*Maximum Dimensions allowed in a Channel List Entry*":

```
#ifdef SUPPORT_CHAN_LIST
/*****************************************************************************/
/* Maximum Dimensions allowed in a Channel List Entry                        */
/* ------------------------------------------------                          */
/* (only used if Channel List support feature is enabled)                    */
/*                                                                           */
/* USER: Modify this value to be equal to the maximum number of dimensions that are   */
/*       allowed in any of the channel list parameters.                      */
/* See User Manual for more information.                                     */
#define MAX_DIMS            (3)             /* Maximum dimensions in a channel list   */
/*****************************************************************************/
#endif
```

Consider each of the Channel List parameter types accepted by your commands. What is the most number of *dimensions* accepted by any of them? On many instruments, just one dimension is used. A few instruments make use of two dimensions. A very few instruments have more than 2. By default, JPA-SCPI Parser allows up to 3 dimensions.

Define **`MAX_DIMS`** to be the maximum number of dimensions supported by your Channel List parameters. You will save a few bytes of RAM if you reduce this number.

12.5 Option to Support More than 255 Characters in an Input Command Line

By default, the library allows up to 255 characters in a command line for parsing. If you need to allow more than 255 characters in an input command line, you can by following these steps:

1) Open your *cmds.h* file
2) Locate the line that begins:

```
#define SCPI_CHAR_IDX    unsigned char
```

3) Modify this definition to use the unsigned type you require, for instance:

```
#define SCPI_CHAR_IDX    unsigned int
```

In this example. The maximum number of characters allowed is now 65535 (assuming the unsigned int type uses 16 bits).

12.6 Option to Support More than 255 Commands

By default, the library allows up to 255 commands to be defined in its *cmds.c* file. Normally this limit is adequate. However if you require more than 255 command definitions then you can do so in this way:

4) Open your *cmds.h* file

5) Locate the line that begins:

```
#define SCPI_CMD_NUM    unsigned char
```

6) Modify this definition to use the unsigned type you require, for instance:

```
#define SCPI_CMD_NUM    unsigned int
```

In this example. The maximum number of commands is now 65535 (assuming the unsigned int type uses 16 bits).

13 Specify Command Keywords

Every command specification has the same fundamental format:

<command keywords> [<parameters>]

Some examples:

Command Keywords	Parameters
*CLS	*none*
*ESE	<value>
SENSe:FUNCtion[:ON?]	*none*
[SOURce:]FREQuency[:CW]	{<freq>|MINimum|MAXimum}
CONFigure[:SCALar]:RESistance	[<range>|MINimum|MAXimum}] , [<resolution>|MINimum|MAXimum]

The command keywords and parameter specifications are defined in two separate tables in the *cmds.c* file. The first task is to list the command keywords for each of your supported commands.

In your *cmds.c* file, locate the section titled *"Command Specs - Part 1: Command Keywords"*. It will look like this:

```
/*****************************************************************************/
/* Command Specs - Part 1: Command Keywords                                  */
/* ----------------------------------------                                  */
/* USER: Create an entry for each sequence of Command Keywords supported.   */
/* Notes:                                                                    */
/*    a) Include full command tree in all entries                            */
/*    a) Enclose optional keywords in square brackets, including any colon   */
/*    b) Enter required characters in Uppercase, optional characters in Lower*/
/*    c) DO NOT include spaces                                               */
/*    d) Duplicate entries are allowed if required in the Command Specs      */
/*****************************************************************************/
const char *SSpecCmdKeywords[] =
{                                          /* Command Number                 */
                                           /* --------------                 */
/* Commands required by all SCPI-Compliant Instruments                       */

/* Required IEEE488.2 Common Commands (see SCPI Standard V1999.0 ch4.1.1)    */
  "*CLS",                                  /*     0                          */
  "*ESE",                                  /*     1                          */
  "*ESE?",                                 /*     2                          */
  "*ESR?",                                 /*     3                          */
  "*IDN?",                                 /*     4                          */
  "*OPC",                                  /*     5                          */
  "*OPC?",                                 /*     6                          */
  "*RST",                                  /*     7                          */
  "*SRE",                                  /*     8                          */
  "*SRE?",                                 /*     9                          */
```

Now look at your list of command specifications that you want to implement. Taking each one in turn, look at it's command keywords and check if there is already an entry for it in this part of the source code. For example, the commands **`*CLS`** and **`SYSTem:Error[:NEXT]?`** will already be present (unless you deleted them).

Any of your command specifications that already have an entry do not require any more work – they are already implemented.

For command specifications that do not yet have an entry in the source code, you need to follow the steps described below for each one.

13.1 Create a Row in Command Specs – Part 1: Command Keywords

Each command specification has a row in this part of *cmds.c*. You need to create a new row for the command specification.

First, decide where to add the row. If the command is in the same subsystem as one that already exists in the source code, then you will add it there. If it is in a subsystem not yet present in the source code, then you will add it after the end of the existing rows, but always before the **`END_OF_COMMANDS`** line.

Each row has this format:

```
"<command keywords>",              /* <command number> */
```

These two elements of the row are described below.

13.1.1 Command Keywords

The *command keywords* parameter is based on the standard SCPI notation:

1. Characters that are required for the long-form of a keyword are lowercase.
2. Optional keywords should be enclosed in square brackets (**[**, **]**). Remember to include the colon as well within the square brackets.
3. Square brackets cannot be nested.
4. Allow a numeric suffix by use of a '**#**' character in the place where the numeric suffix will be entered. A command can have more than one if required.
5. Do not include spaces.

Look at the existing rows in the source code for examples.

13.1.2 Command Number

Although this is just a comment and does not affect how the code compiles, *command numbers* are important to ensuring a successful implementation of your JPA-SCP Parser. Command numbers are used in these ways:

- As a return parameter from the JPA-SCPI Parser function **`SCPI_Parse()`** to tell you which command was matched
- Within *cmds.c* to cross-reference the command keywords with the command parameter specifications

The command number for the first row is always 0, the one below it is 1, and so on, increasing by 1 each row. If you have inserted any rows you will need to adjust the numbering of the rows. You may want to wait until you have added all your new rows before renumbering the command numbers, so they are in the correct order once more.

Parser Limitation: By default, a maximum of 255 rows are allowed in this table, i.e. up to 255 commands are supported. This can be increased as you require. See 12.6 Option to Support More than 255 Commands *for details.*

14 Specify Command Parameters

You should now have a command keywords entry for every command in your command set.

Locate the section of your source code file *cmds.c* that is titled *"Command Specs - Part 2: Parameters"*.

This section requires a corresponding row for each row in the command keywords section. The number of rows and the order of the rows must be the same. In this way, the command keywords and the specifications of the command parameters are associated together to form the complete command specification.

Look at the first row in the parameter specifications. It may be this:

```
{{  NO_PARAMS                                                    }},  /*  0   *CLS */
```

You should find that the first row in the *command keywords* table corresponds to the same command, e.g., in this case it has the same **"*CLS"** keyword.

Check all the other entries in the command keywords section of source code and identify any that do not yet have an entry in the parameters specifications table.

For each of these commands, you need to create a new row in the parameter specifications table. The row must be inserted in the table so that the order of rows in the *command keywords* and *parameter specifications* tables are the same. Follow the instructions in the sections below to create the row.

14.1 Commands without Parameters

If the command takes no parameters then create this row in the parameter specifications:

```
{{ NO_PARAMS        }},                 /*   command syntax     */
```

That completes the specification for this command.

14.2 Commands with Parameters

If the command takes one or more parameters then create this new line in the parameter specifications:

```
{{                  }},                 /*   command syntax     */
```

> You will notice that double opening and closing curly brackets (**{{**) are used to enclose each entry in the array of structures **sSpecCommand[]**. Although only one curly bracket may seem necessary, some compilers (e.g. Borland® C++ Builder™) will not compile this source code without two present.
>
> Most compilers that work with just one bracket also allow two curly brackets. For maximum portability, we have therefore used two curly brackets in our templates. If this causes a problem with your compiler simply remove one of the opening and closing curly brackets from each line in this section of code.

Each parameter that the command allows needs a *parameter specification*. Enter the parameter specifications between the double curly brackets.

Each parameter specification takes this format:

{*<Required>*, *<Type>*, *<Attributes>*}

For each parameter allowed by the command, follow the instructions in the sections below to create the parameter specification. Repeat this process for each of the parameters of the command.

In addition, if the command allows less parameters than the maximum number of parameters specified, use this parameter specification in place of parameters that are not used:

```
{ NOP }
```

For example, a command that only takes one parameter where **MAX_PARAMS** is defined as 3 will use a line of this format:

```
{{ {<Required>, <Type>, <Attributes>} , { NOP } , { NOP } }},
```

14.3 Required and Optional Parameters

A parameter is classed as either *required* or *optional*. Required parameters must be entered by the user in order for the command to be valid. Optional parameters can be entered or left out – the command will still be valid.

In SCPI notation, square brackets are used to surround optional parameters, e.g.:

```
TRIGger:DELay? [MINimum|MAXimum]
```

This command takes one parameter. The parameter is optional as indicated by the square brackets surrounding it.

However, SCPI notation also uses another method to indicate that a parameter is optional: the *default value*.

Consider this command written in SCPI notation:

```
INPut:IMPedance:AUTO {OFF|ON}
```

Here, the command takes one parameter, either *OFF* or *ON*. It also allows the command to be entered without a parameter, in which case it is treated as if the parameter is entered as *OFF*. *OFF* is the default value. The use of bold type or underline indicates the default value for a parameter. If a parameter has a default value then that parameter is *optional*, even though its specification is not surrounded by square brackets.

In summary, a parameter is classed as *optional* in JPA-SCPI Parser if:

- the parameter is within square brackets (nested or not nested)

or:

- the parameter has a default value

If the parameter is *required* then start the parameter specification with this code:

```
{ REQ
```

Alternatively, if the parameter is *optional* then start the specification like this:

```
{ OPT
```

14.4 What Type of Parameter?

JPA-SCPI Parser categorises parameters into these basic types:

- Numeric Value
- Boolean
- Character Data
- String
- Unquoted String
- Numeric List
- Channel List
- Expression
- Character Data with Alternative Parameter Type

You must decide which of these types the parameter is. The sections below should help you to decide. Once you have decided the type, follow the instructions for specifying that type of parameter.

14.4.1 Numeric Value

A *Numeric Value* parameter is used to allow entry of a number. In addition, the number may be followed by units. For instance, a Numeric Value parameter may be used to allow entry of the voltage level to output on a programmable power supply, or the current range to set on an ammeter.

Numeric Value parameters can be entered with or without units, e.g. *100*, *56V*, *1.25e-6H*, *25.7MOHM*. In addition, JPA-SCPI Parser allows entry of Numeric Values in binary, octal and hexadecimal number bases, using the SCPI Standard's syntax (see *"A.3.8.1.1 Number Bases"*).

Numeric Value parameters allow entry of integer values, real values (i.e. decimal point), positive and negative numbers.

14.4.1.1 Examples of Numeric Value Parameters

Command Specification	Example Valid Commands
`*ESE <enable value>`	**ESE 100* **ESE 5* **ESE #B10010100* **ESE #H7F*
`SOURce:VOLTage:DC <voltage>`	*SOUR:VOLT:DC 12.5* *SOUR:VOLTAGE:DC 100MV* *SOURCE:VOLT:DC 12.7e+3 V* *SOUR:VOLT:DC –14.6V*

14.4.2 Boolean

A *Boolean* parameter is used to accept a selection of two states using the mnemonics *ON* and *OFF*. This type of parameter may be used, for instance, in a command that turns auto-ranging on and off, or sets the state of a relay.

In addition to the mnemonics *ON* and *OFF*, a Boolean parameter also accepts a Numeric Value instead. 1 represents *ON* and 0 represents *OFF*. Any other value is rounded to the

nearest integer. If the rounded value is 0 then the Boolean value is *OFF*, otherwise the value is *ON*.

14.4.2.1 Example of a Boolean Parameter

Command Specification	Example Valid Commands
SENSe:RESistance:RANGe:AUTO {ON|OFF}	*SENS:RES:RANG:AUTO ON* *SENS:RES:RANGE:AUTO OFF* *SENSE:RESISTANCE:RANGE:AUTO 15*

14.4.3 Character Data

Some commands allow the entry of mnemonics such as *MINimum*, *MAXimum*, *DEFault*, *ONCE*, *UP*, *DOWN* etc. We call such parameters *Character Data*.

The mnemonics can have a long and a short-form version, just like command keywords. Optional characters (i.e. long-form only) are specified in lowercase.

In some ways you could treat Boolean parameters as a case of Character Data where the allowed mnemonics were *ON* and *OFF*. However, JPA-SCPI Parser also allows entry of Numeric Values for parameters specified as Boolean (as described above). This feature is not available with parameters of type Character Data, so do not specify a parameter as type Character Data when you could use Boolean.

14.4.3.1 Example of a Character Data Parameter

Command Specification	Example Valid Commands
TRIGger[:SEQuence]:SOURce **{BUS|IMMediate|EXTernal}**	*TRIG:SOUR BUS* *TRIG:SEQ:SOUR IMM* *TRIGGER:SOURCE IMMEDIATE* *TRIG:SEQUENCE:SOUR EXT*

14.4.4 String

A *String* parameter is one that accepts a string of ASCII characters delimited by quotes, either double (") or single (') quotes. Such a parameter could be used to allow input of a message to display on an instrument's readout, or a string to store in calibration memory.

A String parameter can accept any ASCII printable character within the quotes, including commas. However the quotes used to delimit the string cannot appear within the string.

14.4.4.1 Example of a String Parameter

Command Specification	Example Valid Commands
SYSTem:DISPlay <message>	*SYST:DISP "Outputting 10 Volts"* *SYSTEM:DISP 'Select "1A" Range'*

14.4.5 Unquoted String

On some occasions you may want to accept a string of characters without the delimiting quotes required by the String parameter type. For instance, to allow entry of a password or pass-code in order to access some restricted functions of your instrument or to access its calibration factors. In this case the parameter type *Unquoted String* may be used. It allows entry of any sequence of ASCII printable characters with the following restriction:

- The string of characters cannot include commas, since a comma is used to separate the parameter from a following parameter

14.4.5.1 Example of an Unquoted String Parameter

Command Specification	Example Valid Commands
`SYSTem:SECure:CODE <code>`	*SYST:SEC:CODE WHJ87RT* *SYSTEM:SECURE:CODE ABC1234*

14.4.6 Numeric List

A *Numeric List* parameter allows a variable number of numeric values to be entered as a single parameter. Each entry in the list can be retrieved by JPA-SCPI Parser for your code to deal with. Numeric Lists can include ranges of values too; these are indicated by a colon (`:`) between the first and last numeric values of the range.

Since a Numeric List is a type of SCPI Expression, it is enclosed by round brackets.

14.4.6.1 Example of a Numeric List Parameter

Command Specification	Example Valid Commands
`SYSTem:ERRor:ENABle[:LIST] <list>`	*SYST:ERR:ENAB (1,5,7:12,15:20,23)* *SYSTEM:ERR:ENAB:LIST (1.7:3.78,5.6)*

NOTE: You must enable optional support for Numeric Lists if any of your commands can accept a Numeric List parameter – see *"12 Optional Support Features"*.

14.4.7 Channel List

A *Channel List* parameter allows a set of one of more channel numbers to be entered as a single parameter. Each entry in the list can be retrieved by JPA-SCPI Parser for your code to deal with.

Entries in Channel Lists can have more than one dimension, e.g. two dimensions can be used to indicate row and column on a switch matrix. Dimensions are indicated using a '`!`' symbol to separate the dimensions.

Channel Lists can also include ranges of channel numbers; these are indicated by a colon (`:`) between the first and last numeric values of the range.

Since a Channel List is a type of SCPI Expression, it is enclosed by round brackets. In addition, it is identified as a Channel List by use of a '`@`' symbol as the first character inside the brackets.

14.4.7.1 Example of a Channel List Parameter

Command Specification	Example Valid Commands
`ROUTe:OPEN <channel list>`	*ROUT:OPEN (@1,2,3:7,4)* *ROUTE:OPEN (@2!3:7!5,8!2)*

NOTE: You must enable optional support for Channel Lists if any of your commands can accept a Channel List parameter – see *"12 Optional Support Features"*.

IMPORTANT NOTE: The Channel List parameter type can be used to parse any type of channel list as long as it only contains numeric values. The SCPI Standard also allows an instrument to accept alphanumeric module specifiers and path names if required (*SCPI Standard V1999.0, 8.3.2 Channel Lists*). If you require this additional support, then you will need to use the Expression parameter type instead, and use your own code to perform the required parsing.

14.4.8 Expression

SCPI defines a set of parameter types that it refers to as *Expressions*. Two of the types are described above, namely Numeric Lists and Channel Lists. SCPI also supports other types of Expression, including numeric expressions (i.e. calculations) and DIF (Data Interchange Format) expressions. If you want to allow one of these types of parameters to be entered then use the *Expression* parameter type. Note that you will need to perform your own parsing of the contents of the Expression.

The *Expression* parameter type allows entry of any string of text that is surrounded by brackets. In addition, it can itself include nested brackets – the nesting level is validated by JPA-SCPI Parser, and incorrect numbers of opening or closing brackets return an error code from the *SCPI_Parse()* function.

14.4.8.1 Example of an Expression Parameter

Command Specification	Example Valid Commands
TRACe:FEED:OCONdition	*TRAC:FEED:OCON (INPUT5=ON)*

NOTE: You must enable optional support for Expressions if any of your commands can accept an Expression parameter – see *"12 Optional Support Features"*.

14.4.9 Character Data with Alternative Parameter Type

Quite often, none of the 8 parameter types above will fit your parameter. For instance, consider the command specification:

SENSe:CURRent:DC:RESolution {<resolution>|MINimum|MAXimum}

This parameter can accept either a Numeric Value (<*resolution*>) or Character Data (the mnemonics *MINimum* and *MAXimum*). This type of parameter specification is very common in SCPI command sets.

We class this parameter in JPA-SCPI Parser as *Character Data with an alternative parameter type (Numeric Value)*.

Any parameter that can accept Character Data may also have an alternative type of parameter. The alternative type of parameter can be any of the 4 remaining types of parameter. Here are examples of each type:

Alternative Type	Parameter Specification	Example Parameter Entries
Numeric Value	**{<seconds>|MINimum|MAXimum}**	*15* *27.5US* *MIN* *MINIMUM* *MAX*
Boolean	**{ON|OFF|ONCE}**	*ON* *ONCE* *0* *15*
String	**{CLEAr|<message>}**	*CLEA* *"Testing…"* *"Reset unit now"*

Unquoted String	**{GUESt|<code>}**	*GUES* *GUEST* *Q4HY78*
Numeric List	**{ALL|NONe|<numeric list>}**	*ALL* *(1:9,12,15)*
Channel List	**{ALL|NONe|<channel list>}**	*NON* *(@2!3:5!7,9!9)*
Expression	**{MINimum|MAXimum|<expr>}**	*MAX* *(15*7.8)*

14.5 Specifying Parameter Type in Code

Once you have decided which of the 9 types of parameter you are specifying, follow the instructions in the relevant section below.

14.6 Specifying a Numeric Value Parameter

If the parameter is type Numeric Value, then enter the second item in the parameter specification as:

```
NUM
```

For example, if the parameter is required and type Numeric Value, then its specification so far will be:

```
{ REQ NUM
```

Note that a comma is not required between the items.

14.6.1 Defining Numeric Values without Units

If the parameter does not allow units, e.g. a parameter that represents a count, then set the third and last item of the parameter specification to this:

```
sNoUnits
```

This is the last item in the parameter specification so close the parameter specification with a closing curly bracket.

For example, if the parameter is optional, then its specification will be:

```
{ OPT NUM sNoUnits }
```

This completes the parameter's specification.

14.6.2 Defining Numeric Value Types

If the parameter does allow units to be entered, then we need to give JPA-SCPI Parser some more information – the *attributes of the Numeric Value*.

Many Numeric Value parameters are used in the same way, e.g. a voltage level. Such parameters have the same attributes, i.e. they allow volts as the units and any values entered without units are taken as volts.

To save time and memory space repeating these attributes for each particular parameter, we instead define a single *Numeric Value Type* and then reference that from the parameter's specification.

A Numeric Value Type has three attributes:

- Default Units
- Alternative Units
- Exponent of Default Units

14.6.2.1 Default Units

Numeric Value parameters that allow units can also specify their *default units*. These are the units used if the parameter entered does not include units. For example, say you want to support this command:

SENSe:CURRent:DC:RANGe <current>

By defining the default units as Amps for this command's parameter, it would allow the following entries:

100UA

0.95MA

1e-6A

1.2

1.5e-9

Notice that entries can contain *A* as the units along with all its derivatives: *UA* (microamps), *MA* (milliamps) etc. In addition, entries without units are allowed. The value is treated as if it was followed by the *A* symbol, since the default units are defined as Amps for this parameter.

14.6.2.2 Alternative Units

On some occasions you may want a command parameter to allow more than one type of units. For instance the same command may allow input of a temperature in degrees Kelvin, Celsius or Fahrenheit. In this case the parameter has *alternative units*.

If the parameter allows alternative units then you need to either create a new *alternative units list* or to use an existing one.

Locate the section in *cmds.c* headed *"Alternative Units"*. It will look similar to this:

```
/*****************************************************************************/
/* Alternative Units                                                         */
/* -----------------                                                         */
/* USER: Create a list for each set of Alternative Units supported (if any)  */
/* Notes:                                                                    */
/*      a) Always include U_END as last member of each list                  */
/*****************************************************************************/
ALT_UNITS_LIST  eAltDegCAndF[] = {U_CELSIUS, U_FAHREN, U_END}; /* Deg C & Deg F*/
```

The example shown here defines an alternative units list comprising degrees Celsius and degrees Fahrenheit. You may have other alternative units lists in your *cmds.c*. If none of the existing lists is suitable for the parameter you need to create a new one.

To do this, add a line similar to the one shown above of this format:

ALT_UNITS_LIST <*name*>[] = {<*alternative units type*>[,...], U_END};

The ***<name>*** does not matter as long as it is unique and is a valid variable name. You may find it best to prefix it with the characters **`eAlt`** (indicating enumerated type, alternative units), and give it a name indicating the types of alternative units it includes.

The list of alternative units within the curly brackets can contain any of the base units types defined in **`enUnits`** in the file *cmds.h*. No type should be repeated and the last type must be **`U_END`**.

Note: If you already have default units defined for the parameter, these are automatically included in the set of allowed units so you do not need to include them in the alternative units list.

Parser Limitations: A maximum of 255 entries are allowed in any array. There is no limit on the number of arrays that can be defined

14.6.2.3 Exponent of Default Units

This feature allows you to specify that values entered without units are automatically scaled according to an exponent.

For instance, if you wish a command to have default units of MilliHenrys, you first specify that the default units are Henrys. You then need to tell JPA-SCPI Parser that values without units are *MilliHenrys* rather than *Henrys*. This is done by setting the *exponent of default units*.

A value entered without units is scaled (multiplied) according to this formula:

Stored Value = Value Entered x 1e<exponent of default units>

For example, if the exponent of default units is –3, and a value of 125 is entered without units, then the value stored by JPA-SCPI Parser is:

Stored Value = 125 x 1e-3 = 0.125

The exponent of default units can have any integer value between –43 and + 43. Normally it is 0, meaning no scaling is performed.

14.6.2.4 Specifying the Numeric Value Type

You will now have:

- Decided the default units, if any, of the parameter
- If the parameter allows alternative units then either created a new alternative units list or found an existing one to use
- Decided the exponent of default units (default is 0)

We now need to either to find an existing Numeric Value Type with these same attributes or create a new one.

Locate the section of code in *cmds.c* titled *"Numeric Value Types"*. It will look something like this:

```
/*****************************************************************************/
/* Numeric Value Types                                                       */
/* -------------------                                                       */
/* USER: Create a structure for each type of Numeric Value supported         */
/* Notes:                                                                    */
/*    a) See JPA-SCPI Parser User Manual for details                         */
/*****************************************************************************/
/*                        Default   Alternative   Exponent of                */
/*        Name            Units     Units         Default Units              */
/*        -----           -------   -----------   -------------              */
NUM_TYPE  sNoUnits    = { U_NONE,   NAU,            0   };      /* No Units    */
NUM_TYPE  sVolts      = { U_VOLT,   NAU,            0   };      /* Volts only  */
```

```
NUM_TYPE  sAmps       = { U_AMP,    NAU,            0   };      /* Amps only   */
NUM_TYPE  sOhms       = { U_OHM,    NAU,            0   };      /* Ohms only   */
NUM_TYPE  sWatts      = { U_WATT,   NAU,            0   };      /* Watts only  */
NUM_TYPE  sDBWatts    = { U_DB_W,   NAU,            0   };      /*db Watts only*/
NUM_TYPE  sJoules     = { U_JOULE,  NAU,            0   };      /* Joules only */
NUM_TYPE  sFarads     = { U_FARAD,  NAU,            0   };      /* Farads only */
NUM_TYPE  sHenrys     = { U_HENRY,  NAU,            0   };      /* Henrys only */
NUM_TYPE  sHertz      = { U_HERTZ,  NAU,            0   };      /* Hertz only  */
NUM_TYPE  sSecs       = { U_SEC,    NAU,            0   };      /* Seconds only*/
NUM_TYPE  sMicroHenrys= { U_HENRY,  NAU,            0   };      /* Henrys only */
NUM_TYPE  sKelvin     = { U_KELVIN, NAU,            0   };  /* Deg Kelvin only */
NUM_TYPE  sCelsius    = { U_CELSIUS,NAU,            0   };  /* Deg Celsius only*/
NUM_TYPE  sFahren     = { U_FAHREN, NAU,            0   };  /* Deg Fahrenh only*/
NUM_TYPE  sTemperature= { U_KELVIN, eAltDegCAndF,   0   };     /* Kelvin; also */
                                                               /* allow C & F  */
```

Each entry in this section is a different Numeric Value Type. The three columns within the curly brackets correspond to the attributes we have been describing: default units, alternative units, and exponent of default units.

If any of the existing entries have the attributes that you require for the parameter, then there is no need to create a new entry here. Instead, just note the name of the entry, e.g. **sVolts**.

If no entry in the Numeric Value Types table matches the requirements of the parameter then you need to create a new one. To do this, add a line after the last entry. The line is of this form:

```
NUM_TYPE <name> = { <default units>, <alternative units>,
                    <exponent of default units> };
```

<name> should be used to describe the purpose of this Numeric Value type, e.g. **sGrams** for a Numeric Value Type that allows entries in grams.

The other attributes are as described above:

<default units> must be one of the types defined in the enumeration **enUnits** in your *cmds.h* file.

<alternative units> must be a name or an entry in the Alternative Units section of *cmds.c*, or **NAU** if no alternative units are allowed.

<exponent of default units> must be an integer between –43 and +43. Use 0 if you are not using this feature.

14.6.3 Defining Numeric Values with Units

Now that you have either created a new Numeric Value Type or found an existing Numeric Value Type to use, you can complete the specification of this Numeric Value parameter.

Return to the *"Command Specs - Part 2: Parameters"* section of your *cmds.c* file. To recap, so far you have specified if the parameter is required or optional, and that it is type Numeric Value. For instance, if the parameter is required, then the specification will look like this:

```
{ REQ NUM
```

The last step is to add a third column containing the name of the Numeric Value Type that contains the attributes of this parameter, and then to close the specification with a closing curly bracket.

For instance, if the parameter uses the Numeric Value Type **sVolts**, then the parameter specification will look like this:

```
{ REQ NUM sVolts }                    parameter is required
```

or this:

```
{ OPT NUM sVolts }
```
parameter is optional

14.7 Specifying a Boolean Parameter

If the parameter is type Boolean, then enter the second item in the parameter specification as:

```
BOOLEAN
```

For example, if the parameter is *required* and type Boolean, then its specification so far will be:

```
{ REQ BOOLEAN
```

A comma is not required between the items.

14.7.1 Default Value

Boolean parameters can have a *default value*. The default value is used if the parameter is not entered.

A Boolean parameter has 3 possibilities:

SCPI Notation	Default Value
{ON|OFF}	No default value
{ON|OFF}	Default value = ON
{ON|OFF}	Default value = OFF

Remember that if the parameter has a default value then the parameter must be *optional*.

14.7.2 Completing the Specification

The third column in the parameter's specification tells JPA-SCPI Parser about the default value.

- If the parameter has no default value then complete the Boolean parameter's specification with:

  ```
  sBNoDef
  ```

- If the parameter has a default value of *ON* (1) then complete the Boolean parameter's specification with:

  ```
  sBDefOn
  ```

- If the parameter has a default value of *OFF* (0) then complete the Boolean parameter's specification with:

  ```
  sBDefOff
  ```

In all cases, close the parameter specification with a closing curly bracket.

For example, if the parameter is optional, type Boolean and has a default value of *ON*, then its parameter specification is:

```
{ OPT BOOLEAN sBDefOn }
```

14.8 Specifying a Character Data Parameter

If the parameter is type Character Data, then enter the second item in the parameter specification as:

```
CH_DAT
```

For example, if the parameter is optional and type Character Data, then its specification so far will be:

```
{ OPT CH_DAT
```

A comma is not required between the items.

We now need to tell JPA-SCPI Parser what possible mnemonics are allowed and which one, if any, is the default item.

14.8.1 Defining Character Data Sequences

Character Data Sequences are strings that contain the choices of mnemonics allowed.

Take a look in your *cmds.*c file at the section titled *"Character Data Sequences"*:

```
/*************************************************************************/
/* Character Data Sequences                                              */
/* ------------------------                                              */
/* USER: Create an entry for each Character Data Sequence supported.     */
/* Notes:                                                                */
/*    a) Separate each Item in a Sequence with a pipe (|) char           */
/*    b) Enter required characters in Uppercase, optional characters in Lower  */
/*    c) Quotes (single and double) are allowed but must be matched      */
/*    d) Do not include spaces within the strings                        */
/*************************************************************************/
/*        Name                    Sequence                               */
/*        ----                    ---------------                        */
CHDAT_SEQ SeqMinMax[]           = "MINimum|MAXimum";
CHDAT_SEQ SeqMinMaxDef[]        = "MINimum|MAXimum|DEFault";
CHDAT_SEQ SeqBusImmExt[]        = "BUS|IMMediate|EXTernal";
```

Each entry contains the set of mnemonics allowed for certain Character Data parameters.

Take a look through the existing entries to see if any of the entries contains the set of mnemonics allowed by the parameter. If not, you will need to create a new entry. To do this, add a new line of this format:

```
CHDAT_SEQ <name>[] = "<sequence>";
```

<name> is the name of the Character Data Sequence. It should reflect the options within the sequence, e.g. **SeqMinMax** for the sequence *MINimum*|*MAXimum*.

<sequence> is the set of mnemonics allowed. Each mnemonic must be separated from each other by a pipe (`|`) character. Each mnemonic must obey these rules:

- Use lowercase characters to indicate the characters only required when entering the long form of a keyword (exactly how you do when specifying command keywords).
- Optional characters should be enclosed in square (`[`,`]`) brackets. Square brackets must not be nested.
- A numeric suffix can be allowed by entering a '**#**' character in the position it is to be entered, e.g. **OUTput#** allows entry of *OUTP5*, *OUTP*, etc.
- Quotes (single and double) are allowed but must be matched, i.e. there must be an even number of them.
- Spaces should not be included unless they are within quotes.

Take a look at the existing entries for examples of valid sequences.

14.8.2 Defining Character Data Types

Having selected the Character Data Sequence for the parameter, or having created a new one, we need to specify other attributes, such as the default item, if any.

This is done in the Character Data Type.

Locate the section of code titled *"Character Data Types"* in your file *cmds.c*:

```
/*****************************************************************************/
/* Character Data Types                                                      */
/* --------------------                                                      */
/* USER: Create a structure for each type of Character Data Sequence supported */
/*       Optional: Remove structures not required                            */
/* Notes:                                                                    */
/*    a) See JPA-SCPI Parser User Manual for details                         */
/*****************************************************************************/
/*                                                 Default   Alternative     */
/*          Name                  Sequence         Item #    Parameter       */
/*          ----                  --------         -------   -----------     */
CHDAT_TYPE  sMinMaxNoUnits    = { SeqMinMax,       NO_DEF,   P_NUM, (void *)&sNoUnits};
CHDAT_TYPE  sMinMaxVolts      = { SeqMinMax,       NO_DEF,   P_NUM, (void *)&sVolts};
CHDAT_TYPE  sMinMaxDefVolts   = { SeqMinMaxDef,    NO_DEF,   P_NUM, (void *)&sVolts};
CHDAT_TYPE  sMinMaxAmps       = { SeqMinMax,       NO_DEF,   P_NUM, (void *)&sAmps};
CHDAT_TYPE  sMinMaxDefAmps    = { SeqMinMaxDef,    NO_DEF,   P_NUM, (void *)&sAmps};
CHDAT_TYPE  sMinMaxOhms       = { SeqMinMax,       NO_DEF,   P_NUM, (void *)&sOhms};
CHDAT_TYPE  sMinMaxDefOhms    = { SeqMinMaxDef,    NO_DEF,   P_NUM, (void *)&sOhms};
CHDAT_TYPE  sMinMaxHertz      = { SeqMinMax,       NO_DEF,   P_NUM, (void *)&sHertz};
CHDAT_TYPE  sMinMaxDefHertz   = { SeqMinMaxDef,    NO_DEF,   P_NUM, (void *)&sHertz};
CHDAT_TYPE  sMinMaxSecs       = { SeqMinMax,       NO_DEF,   P_NUM, (void *)&sSecs};
CHDAT_TYPE  sMinMaxDefSecs    = { SeqMinMaxDef,    NO_DEF,   P_NUM, (void *)&sSecs};
CHDAT_TYPE  sBusImmExt        = { SeqBusImmExt,    NO_DEF,   ALT_NONE };
```

Each entry has three columns: Character Data Sequence, Default Item Number and Alternative Parameter Type.

14.8.2.1 Character Data Sequence

The *Character Data Sequence* is the name of the Character Data Sequence string that contains the allowed mnemonics.

14.8.2.2 Default Item Number

The *Default Item Number* is the number of the item within the Character Data Sequence that is used if the parameter is not entered. Item numbers start at 0 for the first mnemonic in the sequence.

14.8.2.3 Alternative Parameter Type

For now, we ignore this column. It is used when specifying a parameter that is type *Character Data with an Alternative Parameter Type*. We will discuss it later.

Character Data parameters that do not have an Alternative Parameter Type always use **`ALT_NONE`** in this column.

Now take a look at the existing Character Data Types defined in your *cmds.c* file. Does one match your parameter's specification? If so, then note down its name. Otherwise you will need to create a new Character Data Type. To do that, add a new line of this format:

`CHDAT_TYPE <name> = { <sequence>, <default item>, ALT_NONE };`

`<name>` should be formed so that it indicates the items in the sequence. For instance, if the parameter allows *MINimum*|*MAXimum* then you could use the name: **`sMinMax`**.

`<sequence>` is the name of a Character Data Sequence defined in the section of code titled *"Character Data Sequences"*.

<default item> is the number of the default item in the sequence. Item numbers start at 0 for the first item in the sequence. If there is no default item then use the value **NO_DEF** in this column.

As an example, a Character Data Type that allows entry of BUS|IMMediate|EXTernal that has a default value of EXTernal could be specified as:

```
CHDAT_TYPE sBusImmExt = { SeqBusImmExt, 2, ALT_NONE };
```

Take a look at existing entries in the Character Data Types section for more examples.

14.8.3 Completing the Specification of a Character Data Parameter

By this stage, you will have the name of the Character Data Type (either an existing one or one that you have created) that matches the specification of your parameter.

Returning to the command parameter specifications section of your *cmds.c* file, so far, your parameter specification looks like this:

```
{ REQ CH_DAT                    parameter is required
```

or this:

```
{ OPT CH_DAT                    parameter is optional
```

To complete the specification, add a third column containing the name of the Character Data Type and close the specification with a closing curly bracket.

For instance, if the parameter is required, and the Character Data Type used is **sBusImmExt**, then the parameter specification would be:

```
{ REQ CH_DAT sBusImmExt }
```

14.9 Specifying a String Parameter

If the parameter is type String, then enter the second item in the parameter specification as:

```
STRING
```

There is no third column with a String parameter, so simply close the curly brackets.

For example, if the parameter is required and type String, then its specification will be:

```
{ REQ STRING }
```

If the parameter is optional and type String, its specification will be:

```
{ OPT STRING }
```

A comma is not required between the items.

14.10 Specifying an Unquoted String Parameter

If the parameter is type Unquoted String, then enter the second item in the parameter specification as:

```
UNQ_STR
```

There is no third column with an Unquoted String parameter, so simply close the curly brackets.

For example, if the parameter is required and type Unquoted String, then its specification will be:

```
{ REQ UNQ_STR }
```

If the parameter is optional and type Unquoted String, its specification will be:

```
{ OPT UNQ_STR }
```

A comma is not required between the items.

14.11 Specifying a Numeric List Parameter

If the parameter is type Numeric List, then enter the second item in the parameter specification as:

```
NUM_L
```

For example, if the parameter is required and type Numeric List, then its specification so far will be:

```
{ REQ NUM_L
```

A comma is not required between the items.

The attributes of each Numeric List parameter are defined in a *Numeric List Type*. This is referenced from the parameter specification. Numeric List parameters with the same attributes can reference the same Numeric List Type.

14.11.1 Defining a Numeric List Type

A Numeric List Type has the following attributes:

- Allow real (non-integer) values? TRUE/FALSE
- Allow negative values? TRUE/FALSE
- Use range checking? TRUE/FALSE

  If TRUE, then additionally:

  - Minimum value allowed
  - Maximum value allowed

In this way, JPA-SCPI Parser can perform basic validation of entries in a Numeric List. You may need to perform your own checks in addition, e.g. if you only want to accept even numbers etc.

14.11.1.1 A Note on Range Checking

The values you can specify for range checking are stored as long integers. This means that the minimum and maximum values cannot include a decimal point, and are limited to the maximum value representable by a long integer on your system.

As a result of this, if the Numeric List Type is set to allow real (non-integer) numbers, then the range checking will not consider any digits after the decimal point, e.g. If the maximum value is set to 100, then any real value less than 101 will be permitted (e.g. 100.99 will be allowed). If this is insufficient, then you can, of course, perform your own range checking of the entries instead.

14.11.1.2 Specifying a Numeric List Type

Locate the section of code in *cmds.c* titled "*Numeric List Types*":

```
#ifdef SUPPORT_NUM_LIST
/*****************************************************************************/
/* Numeric List Types                                                        */
/* ------------------                                                        */
/* USER: Create a structure for each type of Numeric List supported          */
/* Notes:                                                                    */
/*    a) See JPA-SCPI Parser User Manual for details                         */
/*                                Allow   Allow   Range      Allowed Values  */
/*            Name                Reals?  Neg?    Check?    Minimum   Maximum */
/*            ----                ------  ----    -------   -------   ------- */
NUMLIST_TYPE  sNLAnyNumber    = { TRUE,   TRUE,   FALSE,    0,        0       };
NUMLIST_TYPE  sNLInts         = { FALSE,  TRUE,   FALSE,    0,        0       };
NUMLIST_TYPE  sNLPosInts      = { FALSE,  FALSE,  FALSE,    0,        0       };
NUMLIST_TYPE  sNL8BitPosInts  = { FALSE,  FALSE,  TRUE,     0,        255     };
#endif
```

If any of the entries have the attributes that you require for the parameter, then there is no need to create a new entry. Instead, just note the name of the entry, e.g. **sNLInts**.

If no entry in the Numeric List Types table matches the requirements of the parameter, then you need to create a new one. To do this, add a line after the last entry. The line is of this form:

```
NUMLIST_TYPE  <name>  = { <reals?>,  <negs?>, <range check?>, <min>, <max> };
```

<name> should be used to describe the purpose of this Numeric List Type, e.g. **sNLPosInts** is the name of a Numeric List Type that allows positive integers.

<reals?> should be **TRUE** if the entries in the numeric list can be real (non-integer) as well as integer values. If it is **FALSE**, then only integer values are allowed.

<neg?> should be **TRUE** if the entries in the numeric list can be negative as well as positive values. If it is **FALSE**, then only positive values are allowed.

<range check?> should be **TRUE** if you wish range checking to be performed on each entry in the numeric list. If it is **FALSE**, then no range checking will be performed.

Set ***<min>*** and ***<max>*** to the minimum and maximum values allowed in the numeric list. If ***<range check?>*** is set to **FALSE**, then these values are ignored.

14.11.2 Completing the Specification of a Numeric List Parameter

Returning to the command parameter specifications section of your *cmd.c* file, so far, your parameter specification looks like this:

```
{ REQ NUM_L                                   parameter is required
```

or this:

```
{ OPT NUM_L                                   parameter is optional
```

To complete the specification, add a third column containing the name of the Numeric List Type and close the specification with a closing curly bracket.

For instance, if the parameter is required, and the Numeric List Type used is **sNLPosInts**, then the parameter specification would be:

```
{ REQ NUM_L sNLPosInts }
```

14.12 Specifying a Channel List Parameter

If the parameter is type Channel List, then enter the second item in the parameter specification as:

```
CH_L
```

For example, if the parameter is optional and type Channel List, then its specification so far will be:

```
{ OPT CH_L
```

A comma is not required between the items.

The attributes of each Channel List parameter are defined in a *Channel List Type*. This is referenced from the parameter specification. Channel List parameters with the same attributes can reference the same Channel List Type.

14.12.1 Defining a Channel List Type

A Channel List Type has the following attributes:

- Allow real (non-integer) values? TRUE/FALSE
- Allow negative values? TRUE/FALSE
- Use range checking? TRUE/FALSE

  If TRUE, then additionally:
  - Minimum value allowed
  - Maximum value allowed
- Minimum number of dimensions allowed
- Maximum number of dimensions allowed

In this way, JPA-SCPI Parser can perform basic validation of entries in a Channel List. You may need to perform your own checks in addition, e.g. if you only want to accept even numbers etc.

14.12.1.1 A Note on Range Checking

The values you can specify for range checking are stored as long integers. This means that the minimum and maximum values cannot include a decimal point, and are limited to the maximum value representable by a long integer on your system.

As a result of this, if the Channel List Type is set to allow real (non-integer) numbers, then the range checking will not consider any digits after the decimal point, e.g. If the maximum value is set to 100, then any real value less than 101 will be permitted (e.g. 100.99 will be allowed). If this is insufficient, then you can, of course, perform your own range checking of the entries instead.

If the channel list accepts entries of more than one dimension, then you should also note that the same range checking is performed across all dimensions. If you wish to accept a different range of values in the second dimension, for instance, then you will need to perform your own range checking.

14.12.1.2 Dimensions

Some channel lists accept multi-dimensional entries, e.g. a two dimensional channel list can be used to accept entry of row and column information. Set the minimum and maximum dimensions according to the requirements of your channel list. For example, a channel list

that only accepts single dimension entries should have minimum and maximum dimensions both set to 1.

14.12.1.3 Specifying a Channel List Type

Locate the section of code in *cmds.c* titled "*Channel List Types*":

```
#ifdef SUPPORT_CHAN_LIST
/*****************************************************************************/
/* Channel List Types                                                        */
/* ------------------                                                        */
/* USER: Create a structure for each type of Channel List supported          */
/* Notes:                                                                    */
/*    a) See JPA-SCPI Parser User Manual for details                         */
/*****************************************************************************/
/*                                Allow  Allow   Range  Dimensions Allowed Vals*/
/*             Name               Reals? Neg?    Check?  Min Max    Min   Max  */
/*             ----               ------ ----    ------  --- ---    ----  ---- */
CHANLIST_TYPE  sCL1Dim        = { TRUE,  TRUE,   FALSE,  1,  1,     0,    0    };
CHANLIST_TYPE  sCL2Dim        = { TRUE,  TRUE,   FALSE,  2,  2,     0,    0    };
CHANLIST_TYPE  sCL1DimInts    = { FALSE, TRUE,   FALSE,  1,  1,     0,    0    };
CHANLIST_TYPE  sCL2DimPosInts = { FALSE, FALSE,  FALSE,  2,  2,     0,    0    };
#endif
```

If any of the entries have the attributes that you require for the parameter, then there is no need to create a new entry. Instead, just note the name of the entry, e.g. **sCL2Dim**.

If no entry in the Channel List Types table matches the requirements of the parameter, then you need to create a new one. To do this, add a line after the last entry. The line is of this form:

```
NUMLIST_TYPE  <name>  = { <reals?>,  <negs?>, <range check?>, <min dimensions>,
                         <max dimensions>, <min value>, <max value> };
```

<name> should be used to describe the purpose of this Channel List Type, e.g. **sCL2DimPosInts** is the name of a Channel List Type that take 2 dimensional entries, and allows positive integers.

<reals?> should be **TRUE** if the entries in the channel list can be real (non-integer) as well as integer values. If it is **FALSE**, then only integer values are allowed.

<neg?> should be **TRUE** if the entries in the channel list can be negative as well as positive values. If it is **FALSE**, then only positive values are allowed.

<range check?> should be **TRUE** if you wish range checking to be performed on each entry in the channel list. If it is **FALSE**, then no range checking will be performed.

Set ***<min dimensions>*** and ***<max dimensions>*** to only allow entries with the correct number(s) of dimensions. If all entries in the channel list must have the same number of dimensions, then both these values should be the same.

Set ***<min value>*** and ***<max value>*** to the minimum and maximum values allowed in the numeric list. If ***<range check?>*** is set to **FALSE**, then these values are ignored.

14.12.2 Completing the Specification of a Channel List Parameter

Returning to the command parameter specifications section of your *cmd.c* file, so far, your parameter specification looks like this:

```
{ REQ CH_L                              parameter is required
```

or this:

```
{ OPT CH_L                              parameter is optional
```

To complete the specification, add a third column containing the name of the Numeric List Type and close the specification with a closing curly bracket.

For instance, if the parameter is required, and the Numeric List Type used is **sNLPosInts**, then the parameter specification would be:

```
{ REQ CH_LsCL2DimPosInts }
```

14.13 Specifying an Expression Parameter

If the parameter is type Expression, then enter the second item in the parameter specification as:

```
EXPR
```

There is no third column with an Expression parameter, so simply close the curly brackets.

For example, if the parameter is required and type Expression, then its specification will be:

```
{ REQ EXPR }
```

If the parameter is optional and type Expression, its specification will be:

```
{ OPT EXPR }
```

A comma is not required between the items.

14.14 Specifying a Character Data Parameter with an Alternative Parameter Type

For parameters of type *Character Data with an Alternative Parameter Type* you first need to carry out these steps:

1) If the Alternative Parameter Type is *Numeric Value* then create or select a Numeric Value Type that matches the parameter's requirements when a Numeric Value is entered – see section *"14.6.2 Defining Numeric Value Types"* for details.
2) Create or select a Character Data Sequence that matches the requirements of the parameter when used in Character Data form – see section *"14.8.1 Defining Character Data Sequences"*.

Now you can create or select a Character Data Type for the parameter. Follow the instructions in section *"14.8.2 Defining Character Data Types"*, but this time we will specify the Alternative Parameter Type, rather than using **ALT_NONE** in that column. You may also want to indicate in the name of the Character Data Type what other parameter is allowed, e.g. **sMinMaxVolts** could be used to indicate a Character Data Type that accepted *MINimum*|*MAXimum*|<*volts*>.

As always, if, after deciding what Alternative Parameter Type is needed, a Character Data Type already exists with all the same attributes required by the parameter, don't create a new Character Data Type, just use the existing one instead.

14.14.1 Alternative Parameter Type

The Alternative Parameter Type in the Character Data Type entry tells JPA-SCPI Parser what other type of parameter can be entered. Its specification depends on the type of Alternative Parameter.

Follow one of the sections below, according to what type of alternative parameter is allowed.

14.14.1.1 Alternative Parameter Type is Numeric Value

In this case Alternative Parameter Type takes the form:

```
P_NUM, (void *)&<Numeric Value Type>
```

For instance, if the Numeric Value Type used is called **sOhms**, then the Alternative Parameter Type entry in the Character Data Type table will be:

```
P_NUM, (void *)&sOhms
```

14.14.1.2 Alternative Parameter Type is Boolean

If the Boolean value has no default value, then the Alternative Parameter Type is:

```
P_BOOL, (void *)&sBNoDef
```

If the Boolean value has a default of On (1), then the Alternative Parameter Type is:

```
P_BOOL, (void *)&sBDefOn
```

If the Boolean value has a default of Off (0), then the Alternative Parameter Type is:

```
P_BOOL, (void *)&sBDefOff
```

14.14.1.3 Alternative Parameter Type is String

In this case, the Alternative Parameter Type is always:

```
P_STR, (void *)0
```

14.14.1.4 Alternative Parameter Type is Unquoted String

In this case, the Alternative Parameter Type is always:

```
P_UNQ_STR, (void *)0
```

14.14.1.5 Alternative Parameter Type is Numeric List

In this case Alternative Parameter Type takes the form:

```
P_NUM_LIST, (void *)&<Numeric List Type>
```

For instance, if the Numeric List Type used is called **sNLInts**, then the Alternative Parameter Type entry in the Character Data Type table will be:

```
P_NUM_LIST, (void *)&sNLInts
```

14.14.1.6 Alternative Parameter Type is Channel List

In this case Alternative Parameter Type takes the form:

```
P_CHAN_LIST, (void *)&<Channel List Type>
```

For instance, if the Channel List Type used is called **sCL2Dim**, then the Alternative Parameter Type entry in the Character Data Type table will be:

```
P_CHAN_LIST, (void *)&sCL2Dim
```

14.14.1.7 Alternative Parameter Type is Expression

In this case, the Alternative Parameter Type is always:

```
P_EXPR, (void *)0
```

14.14.2 Completing the Specification of a Character Data Parameter with an Alternative Parameter Type

Returning to the command parameter specifications section of your *cmds.c* file, add a second column to your parameter specification:

```
CH_DAT
```

The parameter specification will, so far, look like this:

```
{ REQ CH_DAT                    parameter is required
```

or this:

```
{ OPT CH_DAT                    parameter is optional
```

You also know the name of the Character Data Type (either an existing one or one that you have created) that matches the specification of your parameter.

So to complete the parameter specification, add a third column containing the name of the Character Data Type and close the specification with a closing curly bracket.

For instance, if the parameter is required, and the Character Data Type used is **sMinMaxVolts**, then the parameter specification is:

```
{ REQ CH_DAT sMinMaxVolts }
```

15 Remove Unused Declarations

The template file(s) that you used as the basis for your own *cmds.c* and *cmds.h* files may have included some definitions and structures that are not needed by your command set. For instance, you may not need to support *Joules* as base units, or you do not require the Character Data Sequence **`BUS|IMMediate|EXTernal`**.

These unused definitions will be taking up ROM space on your platform, but apart from that they do no harm. If you want to you may delete unused definitions from the *cmds.c* and *cmds.h* files. You may like to follow these tips:

- Try compiling the *cmds.c* module before removing any of the unused components – the compiler's output report may include a list of unused items that you can delete without causing problems.
- Take a backup of *cmds.c* and *cmds.h* before deleting any items, in case you find you do need them after all.

16 Integrate into Your Source Code

16.1 Compiler Requirements

Before we discuss the steps to integration, you will need to carry out a few basic steps to physically include JPA-SCPI Parser in your project.

1) Add these files to your project's build list:
   - *scpi.c*
   - *cmds.c*
2) If you need to include complier/platform-specific header files in these files in order for them to compile on your system, then open *scpi.c* and *cmds.c* for editing. Near the top of each file is a list of header files and a comment saying *"Include any headers required by your compiler here"*. Insert any **`#include`**'s required into the files.
3) *scpi.c* implements its own versions of 4 functions that are available in standard C libraries: **`strlen()`**, **`tolower()`**, **`islower()`**, and **`isdigit()`**. If you want, you can remove these function definitions from *scpi.c*. Include the required C libraries in your project. You will also need to add the appropriate **`#include`**'s to *scpi.c*.
4) Open *cmds.h*. It includes definitions such as **`ULONG_MAX`** that tell JPA-SCPI Parser the maximum value that can be represented by different C variable types on your platform. Adjust these values to match your compiler/platform. Alternatively, you can comment out (or delete) those lines and instead include the standard C library header file *limits.h* in *cmds.h* – it includes the definitions required here.
5) Any of your own modules that require access to JPA-SCPI Parser's Access Functions or variable types need to include these header files, in this order:

```
#include "cmds.h"
#include "scpi.h"
```

16.2 Integration Overview

When using JPA-SCPI Parser to interpret commands received on your instrument's communications port, the process is this:

1) When a command line terminator is received in the input buffer of the communication's port, copy the contents of the input buffer into a character array for parsing
2) Call **`SCPI_Parse()`** Access Function to parse the first command in the command line
3) If the command is valid then:
   a) Validate the numeric suffices in the command, if any
   b) Retrieve the parameters, if any, using JPA-SCPI Parser Access Functions
   c) Validate the parameters
   d) If the parameters are valid then perform the action required by the command
4) If an error has occurred then handle it
5) If no error has occurred and there are more commands in the command line then repeat steps 2 onwards for each subsequent command

16.3 Copy Command Line from Input Buffer

The command line comprises all the characters in the input buffer up to the command line terminator. Depending on your communications protocol, the terminator could be the EOI signal (if GPIB), a Carriage Return character, a Linefeed character or a combination of these. In fact whatever your instrument allows as a terminator is acceptable.

When a command line terminator is received, the parsing of the command line can begin. In order to allow the input buffer to be used during parsing, the first job is to copy the contents of the input buffer up to the terminator into a character array for parsing.

When copying the input buffer into the character array, the array must be terminated with the null character (*'\0'*). If the input buffer uses a terminator character (such as carriage return), then the null character can either replace this character or be placed after it – JPA-SCPI Parser ignores carriage return and linefeed characters (it treats them, and all ASCII characters with codes 1-32, as white-space).

Parser Limitations: By default, the command line cannot be more than 255 characters long. This can be increased if you wish. See 12.5 Option to Support More than 255 Commands *for details.*

16.4 Parsing Loop

Now we have a copy of the command line to be parsed, we can parse its contents.

A command line may contain one command or it may contain many. JPA-SCPI Parser parses a single command at a time, returning a pointer to the start of the next command in the command line to be parsed (if there is one).

When parsing a command line, the commands are carried out in order. If an error occurs, it is usual to stop parsing the command line – subsequent commands are ignored. This is the approach we will illustrate here.

The code varies slightly depending on whether support for Numeric Suffices is enabled, i.e. if **`SUPPORT_NUM_SUFFIX`** is #defined. By default it is #defined – see section "*12 Optional Support Features*" for more information. Follow the section below that matches your configuration.

16.4.1 When SUPPORT_NUM_SUFFIX is #defined

Note: Code that is present because **`SUPPORT_NUM_SUFFIX`** is #defined in shown below with a grey background.

```
char SInput[256];                   // Copy of command line
char *SCmd;                         // Pointer to command to be parsed
UCHAR Err;                          // Returned Error code
BOOL bResetCmdTree;                 // Resets command tree if TRUE
SCPI_CMD_NUM CmdNum;                // Returned number of matching cmd
struct strParam sParams[MAX_PARAMS];      // Returned parameters
unsigned int uiNumSuf[MAX_NUM_SUFFIX];    // Returned numeric suffices
UCHAR NumSufCnt;                    // Returned numeric suffix count
   :
copy input buffer into SInput[], with null terminator
   :
// Parsing Loop
SCmd = &(SInput[0]);              // Point to first command in line
bResetCmdTree = TRUE;             // Reset tree for first command
```

```
do                                  // Loop for each command in line
{
  Err = SCPI_Parse (&SCmd, bResetCmdTree, &CmdNum, sParams,
                    &NumSufCnt, uiNumSuf);
                                    // Parse command
  if (Err == SCPI_ERR_NONE)         // If command is valid
  {
    // Dispatch Table
    switch (CmdNum)
    {
      case 0: command_handler_0 (sParams); break;
      case 1: command_handler_1 (); break;
      case 2: command_handler_2 (sParams, NumSufCnt, uiNumSuf); break;
        :
    }
  }
  else                              // If command is invalid
  {
    switch (Err)
    {
      case SCPI_ERR_TOO_MANY_NUM_SUF:  handle too many numeric
                                       suffices in command;      break;
      case SCPI_ERR_NUM_SUF_INVALID:   handle invalid numeric suffix;
                                                                 break;
      case SCPI_ERR_INVALID_VALUE: handle invalid value in list; break;
      case SCPI_ERR_INVALID_DIMS:      handle invalid dimensions
                                       in channel list entry;    break;
      case SCPI_ERR_PARAM_OVERFLOW:    handle overflow;          break;
      case SCPI_ERR_PARAM_UNITS:       handle wrong units;       break;
      case SCPI_ERR_PARAM_TYPE:        handle wrong param type;  break;
      case SCPI_ERR_PARAM_COUNT:       handle wrong param count; break;
      case SCPI_ERR_UNMATCHED_QUOTE:   handle unmatched quote;   break;
      case SCPI_ERR_UNMATCHED_BRACKET: handle unmatched bracket; break;
      case SCPI_ERR_INVALID_CMD:       handle invalid command;   break;
    }
  }
  if (bResetCmdTree)                // Don't reset command tree
    bResetCmdTree = FALSE;          // after first command in line

} while (Err == SCPI_ERR_NONE);     // Parse while no errors and
                                    // commands left to be parsed
```

16.4.2 When SUPPORT_NUM_SUFFIX is not #defined

```
char SInput[256];                   // Copy of command line
char *SCmd;                         // Pointer to command to be parsed
UCHAR Err;                          // Returned Error code
BOOL bResetCmdTree;                 // Resets command tree if TRUE
SCPI_CMD_NUM CmdNum;                // Returned number of matching cmd
struct strParam sParams[MAX_PARAMS];      // Returned parameters
   :
copy input buffer into SInput[], with null terminator
   :
// Parsing Loop
SCmd = &(SInput[0]);                // Point to first command in line
bResetCmdTree = TRUE;               // Reset tree for first command
```

```
do                                  // Loop for each command in line
{
  Err = SCPI_Parse (&SCmd, bResetCmdTree, &CmdNum, sParams);
                                    // Parse command
  if (Err == SCPI_ERR_NONE)         // If command is valid
  {
    // Dispatch Table
    switch (CmdNum)
    {
      case 0: command_handler_0 (sParams); break;
      case 1: command_handler_1 (); break;
      case 2: command_handler_2 (sParams); break;
        :
    }
  }
  else                              // If command is invalid
  {
    switch (Err)
    {
      case SCPI_ERR_INVALID_VALUE: handle invalid value in list; break;
      case SCPI_ERR_INVALID_DIMS:      handle invalid dimensions
                                       in channel list entry;    break;
      case SCPI_ERR_PARAM_OVERFLOW:    handle overflow;          break;
      case SCPI_ERR_PARAM_UNITS:       handle wrong units;       break;
      case SCPI_ERR_PARAM_TYPE:        handle wrong param type;  break;
      case SCPI_ERR_PARAM_COUNT:       handle wrong param count; break;
      case SCPI_ERR_UNMATCHED_QUOTE:   handle unmatched quote;   break;
      case SCPI_ERR_UNMATCHED_BRACKET: handle unmatched bracket; break;
      case SCPI_ERR_INVALID_CMD:       handle invalid command;   break;
    }
  }
  if (bResetCmdTree)                // Don't reset command tree
    bResetCmdTree = FALSE;          // after first command in line

} while (Err == SCPI_ERR_NONE);     // Parse while no errors and
                                    // commands left to be parsed
```

16.4.3 Variables

Let's take a look at this code, whichever variety above you are looking at.

Starting at the top are the variable declarations used. **SInput[256]** contains the command line copied from the input buffer. You can either use a variable like **SInput**, declared locally within a function, or make it global to your module. You will need access to the contents of this character array if any of your command specifications allow a String or Unquoted String parameter – these types of parameter contain a pointer to the first character of the string in the command line character array. For this reason, it is often best to declare the **SInput** variable global to the module.

SCmd is a pointer used to point to the first character of the command to be parsed within the command line. Initially it points to the first character in the command line.

Below this is variable **Err**. **Err** is declared as a **UCHAR**. **UCHAR** is #defined as **unsigned char** in *scpi.h*, i.e. an unsigned 8-bit number. **Err** is used to contain the return value from the calls to the JPA-SCPI Parser Access Functions. All Access Functions return one of a set of common error codes. These are described later.

bResetCmdTree is a Boolean variable. It is used to tell JPA-SCPI Parser whether or not to reset the command tree. The command tree stores the current node that was reached by the previous command – remember that SCPI commands use the level reached by the previous command in the command line by default.

CmdNum is used to contain the returned number of the command specification that matches the command parsed by **SCPI_Parse()**. This is then used in the dispatch table (described later) to call the appropriate function to handle the command.

sParams[MAX_PARAMS] is used to contain the returned parameters of the matching command. **MAX_PARAMS** is defined in *cmds.h*, by default it is **2**. JPA-SCPI Parser Access Functions can be used to allow easy conversion from parameter structures to C language variable types, such as integers, doubles, etc. They are described later.

*If **SUPPORT_NUM_SUFFIX** is #defined...*

uiNumSufCnt[MAX_NUM_SUFFIX] is used to contain the returned numeric suffices embedded in the command. **MAX_NUM_SUFFIX** is defined in *cmds.h*. Note, any numeric suffix that is not entered will be given the default value by JPA-SCPI Parser.

NumSufCnt is used to hold the number of numeric suffices entered in the command (including numeric suffices given the default value).

16.4.4 Parsing the Command Line

After the input buffer of the instrument's communications port is copied into **SInput**, this loop is in charge of parsing each command within it.

16.4.4.1 Calling SCPI_Parse()

The first job is to call the **SCPI_Parse()** function. This parses a single command in the command line. The command within the command line to be parsed is pointed to by the first parameter of **SCPI_Parse()**, **SCmd** in this case. **SCPI_Parse()** parses the command up until the first command delimiter character, i.e. a semi-colon (*;*). **SCPI_Parse()** returns this pointer so that it points to the first character in the next command. In this way, it is ready for the next call to **SCPI_Parse()**.

The second parameter in **SCPI_Parse()** is **bResetCmdTree**. If *TRUE* then the command tree is reset, otherwise the command tree is not reset and the current node reached by the previous command is maintained. **bResetCmdTree** is set to *TRUE* for the first command in the command line – in SCPI, the command tree is always reset to the base node for each new command line. For subsequent commands, **bResetCmdTree** is set to *FALSE*.

SCPI_Parse() parses the command pointed to by **SCmd**. If the command and its parameters (if any) match any of the command specifications in the *cmds.c* file, then **SCPI_Parse()** returns the **SCPI_ERR_NONE** code, to indicate a successful match. It also returns the number of the matching command specification (via **CmdNum**) and the contents of the commands parameters (in **sParams**).

*If **SUPPORT_NUM_SUFFIX** is #defined...*

SCPI_Parse() also returns the numeric suffices in the command (via **uiNumSuf[]**) and the number of numeric suffices returned (**NumSufCnt**).

16.4.4.2 Dispatch Table

If a successful match exists then you now need to call the appropriate *command handler* function in your code to handle that command. This process often uses a *dispatch table*. It usually comprises a **`switch(CmdNum)`** table, as shown in the code above. For each command number, call a function to handle that command. Any commands that accept parameters will need their handler function to validate and use the parameters. These functions therefore need to include **`sParams`** in their function parameters (e.g. **`command_handler_0()`** and **`command_handler_2()`** in the code above.

> *If **`SUPPORT_NUM_SUFFIX`** is #defined...*
>
> If any of the command handler functions need to know the numeric suffix(ces) that was/were entered, then you will also want to pass the **`uiNumSuf[]`** array, and possibly **`NumSufCnt`** as well. For example, see the call to **`command_handler_2()`** in the code above (section 16.4.1).

16.4.4.3 Handling Invalid Commands

Alternatively, if the command does not match a valid command, maybe because the command keywords are invalid, or the parameters are the wrong type for the command specification, then **`SCPI_Parse()`** will returns an error code instead of **`SCPI_ERR_NONE`**.

The error codes are caught within a **`switch (Err)`** table in the code above. It is then up to you how you want to handle invalid command errors. A common way is to add the error code to a FIFO error buffer. The error codes can be retrieved by the remote computer using the SCPI command **`SYSTem:ERRor[:NEXT]?`**. Refer to the SCPI Standard for details.

There is one error code that is returned by **`SCPI_Parse()`** that is not actually an error: **`SCPI_ERR_NO_COMMAND`**. This code is returned by **`SCPI_Parse()`** to indicate there was no command to be parsed. This will occur:

- if the command line is empty or contains only whitespace and/or semi-colons
- if the last command in the command line has been parsed

The simplest way to handle this error code is to just exit the parsing loop, as performed in the example code above using the line **`while(Err == SCPI_ERR_NONE)`**. In addition, if **`SCPI_ERR_NO_COMMAND`** is returned by **`SCPI_Parse()`** you should not add it to your error buffer.

16.5 Command Handler Functions

As described above, each command in the command specification requires a *command handler function*. This function:

- Retrieves and validates the numeric suffix(ces) in the command (if numeric suffices are supported and validation is required)
- Checks the types of command parameters (if required)
- Converts the command parameters (if any) into C language variables
- Validates the parameters (if any)
- Carries out the appropriate actions for the command

The actual code used depends on what your command does and what types of parameter it can take. In general though, the steps are always the same.

Parameters returned by the **SCPI_Parse()** function are stored in an array of **struct strParam** structures. JPA-SCPI Parser provides Access Functions that make conversion of the parameters into standard C variable types easy. Alternatively you can access the elements of the structure and its related structures directly – information about the structures is given in the *Design Notes* document.

16.5.1 Numeric Suffices

If your system supports numeric suffices (i.e. **SUPPORT_NUM_SUFFIX** is #defined) and the command takes one or more numeric suffix, then you will want to retrieve the numeric suffix(ces) entered and check they are valid.

SCPI_Parse() returns an array of numeric suffices (**uiNumSuf[]**) and the number of numeric suffices in the command (**NumSufCnt**). Starting at element 0, the parser populates an element of **uiNumSuf[]** each time it encounters a '**#**' symbol in the command specification (or in the character data specification of a parameter). If the user has not entered a numeric suffix, then the value is taken as the default value (**NUM_SUF_DEFAULT_VAL**).

For instance, if the command specification is

OUTPut#:RELay# {INTernal|EXTernal#}

and the command entered is

OUTP2:REL EXT3

then **SCPI_Parse()** will return these values:

NumSufCnt==3

uiNumSuf[0]==2, **uiNumSuf[1]==1** (assuming default value of 1), **uiNumSuf[2]==3**

But, for the same command specification, if the command entered is

OUTP3:REL2 INT

then **SCPI_Parse()** will return these values:

NumSufCnt==2

uiNumSuf[0]==3, **uiNumSuf[1]==2**

Note in this second case that, since the Character Data entered (**INTernal**) did not take a numeric suffix, then the number of numeric suffices returned is one less.

That was somewhat of an extreme example; in most cases you will only have one or maybe two numeric suffices that can be entered, making your validation of them easier.

Since you know the command specification for the command being handled, your code knows what each numeric suffix in the command relates to, whether it is an output channel number, the number of an external trigger source or whatever. You can therefore validate and use the numeric suffices as you require in your command handler function.

> JPA-SCPI Parser does include basic range checking for numeric suffices. **SCPI_Parse()** checks that the numeric suffices entered are all within range of **NUM_SUF_MIN_VAL** to **NUM_SUF_MAX_VAL**. If any are outside that range (and are not equal to **NUM_SUF_DEFAULT_VAL**), then it returns error code **SCPI_ERR_NUM_SUF_INVALID**.

16.5.2 Parameter Types

If you are handling a command that has one or more optional parameters, you will need to know which parameters were entered. If you are handling a command that includes a parameter of type *Character Data with Alternative Type* then you will need to know whether the parameter was entered as Character Data or another type. If your command includes a Numeric Value, you may need to know if the number was positive or negative, and whether it was integer value.

In all these cases, you can call this Access Function:

```
SCPI_ParamType(..)
```

SCPI_ParamType() takes a parameter structure, as returned by **SCPI_Parse()** and tells you:

- The type of parameter entered, or no parameter if not entered
- If the type is Numeric Value, then whether the value is positive/negative and integer/real

In your command handler function, you could include this code:

```
UCHAR Err;
enum enParamType ePType;
UCHAR NumSubtype;
Err = SCPI_ParamType (&(sParams[0]), &ePType, &NumSubtype)
```

where **sParams[0]** is the first parameter returned by **SCPI_Parse()**.

Err will be **SCPI_ERR_NONE** if no errors occurred, otherwise it will be an error code. See Appendix B for more information.

If **Err** is **SCPI_ERR_NONE**, then **ePType** will contain the type of parameter:

ePType	Meaning
P_NONE	No parameter was entered
P_NUM	Numeric Value
P_BOOL	Boolean
P_CH_DAT	Character Data
P_STR	String
P_UNQ_STR	Unquoted String
P_NUM_LIST	Numeric List
P_CHAN_LIST	Channel List
P_EXPR	Expression

If the parameter was optional, then **P_NONE** indicates that the parameter was not entered.

If the parameter is type Numeric Value (**ePType==P_NUM**) then the other parameter of **SCPI_ParamType()**, **NumSubType**, contains further information.

NumSubType comprises 8 bits:

Bit Number	Use
7-2	*Not Used*
1	1=Real number, 0=Integer
0	1=Negative number, 0=Positive

By seeing which bits are set, you can determine which attributes the Numeric Value parameter has. To assist this, two **#define**'s are present in *scpi.h*:

```
#define SCPI_NUM_ATTR_NEG     (1)
#define SCPI_NUM_ATTR_REAL    (2)
```

You can therefore use this kind of code to determine the attributes of a Numeric Value parameter:

```
UCHAR Err;
enum enParamType ePType;
UCHAR NumSubtype;
Err = SCPI_ParamType (&(sParams[0]), &ePType, &NumSubtype)
if (Err == SCPI_ERR_NONE)
{
  if (ePType == P_NUM)// Numeric Value
  {
    if (NumSubType & SCPI_NUM_ATTR_NEG)
      // Value is negative
    else
      // Value is positive
    if (NumSubType & SCPI_NUM_ATTR_REAL)
      // Value is real
    else
      // Value is an integer
  }
}
```

where **sParams[0]** is the first parameter returned by **SCPI_Parse()**.

16.5.3 Converting Parameters to C Variables

Once you know the types of parameters returned (and which optional parameters were entered), you will want to convert the parameters into native C variable types.

The conversion performed depends on the type of parameter returned.

16.5.3.1 Converting a Numeric Value Parameter

There are 5 Access Functions for converting a Numeric Value parameter to a C variable. Each one returns a different type of C variable.

Access Function	Returns
SCPI_ParamToUnsignedInt()	unsigned int
SCPI_ParamToInt()	int
SCPI_ParamToUnsignedLong()	unsigned long
SCPI_ParamToLong()	long
SCPI_ParamToDouble()	double

Remember to check the attributes of the Numeric Value if you need to before calling one of these functions. For instance, if the value entered was negative and you only allow positive numbers for this parameter, then you know immediately that the value is invalid and do not need to perform this conversion. The same procedure applies if you only allow integers for the value and the value entered was *real*.

Say you want to convert the Numeric Value entered into a variable of type unsigned long. You could use code like this:

```
UCHAR Err;
unsigned long ulVal;
    :
Err = SCPI_ParamToUnsignedLong (&(sParam[1]), &ulVal);
```

where **sParam[1]** is the second parameter returned by **SCPI_Parse()**.

The function will return the value of the parameter in **ulVal**.

Whatever Access Function you are using to convert a Numeric Value parameter, you should also check the value of **Err**. It will be **SCPI_ERR_NONE** if the conversion went ok. Otherwise, **Err** indicates a problem with the conversion:

Err	Description
SCPI_ERR_PARAM_TYPE	Parameter is not type Numeric Value
SCPI_ERR_OVERFLOW	Value of parameter was too big to be contained in the return value, e.g. parameter has value *123456* and return value was type *int* which has a maximum value of 32767 (if *int* is 16-bit)

Note: **Err** will be **SCPI_ERR_NONE** if you convert a negative value parameter to an unsigned C variable, e.g. using **SCPI_ParamToUnsignedInt()**. The value returned will be the value with the negative sign ignored. If you wish to disallow negative numbers, then check the attributes of the Numeric Value first, as discussed earlier.

There is also an Access Function for retrieving the units of the Numeric Value parameter. Normally you will not need to call this if, for example, only Volts are allowed then all values returned will be in volts. If you allow one of more types of units, then you will want to know what units were entered.

The Access Function to use is:

```
SCPI_ParamUnits(..)
```

Your code could look like this:

```
UCHAR Err;
enum enUnits eUnits;
    :
Err = SCPI_ParamUnits (&(sParam[0]), &eUnits);
```

where **sParam[0]** is the first parameter returned by **SCPI_Parse()**.

Err will be **SCPI_ERR_NONE** if no errors occurred. See Appendix B for a list of error codes and more information.

If **Err** is **SCPI_ERR_NONE** then **eUnits** will contain the type of units entered, or **U_NONE** if no units were entered. The types of units are defined in your *cmds.h* file.

16.5.3.2 Converting a Boolean Parameter

This Access Function is used to convert a Boolean parameter into a C variable of type BOOL:

```
SCPI_ParamToBOOL(..)
```

In code, you can use it like this:

```
UCHAR Err;
BOOL bVal;
    :
Err = SCPI_ParamToBool (&(sParam[0]), &bVal);
```

where **sParam[0]** is the first parameter returned by **SCPI_Parse()**.

If **Err** is **SCPI_ERR_NONE** then the conversion was ok and **bVal** will contain the Boolean value of the parameter, either *0* (OFF) or *1* (ON).

If **Err** is not **SCPI_ERR_NONE** then an error occurred. See Appendix B for a list of error codes and more information.

16.5.3.3 Converting a Character Data Parameter

This Access Function is used to convert a Character Data parameter into a number representing the item in the Character Data Sequence that was entered:

```
SCPI_ParamToCharDataItem(..)
```

For example, say that the first parameter returned was type Character Data. By using the code below, you can retrieve the number of the item entered.

```
UCHAR Err;
UCHAR ItemNum;
    :
Err = SCPI_ParamToCharDataItem (&(sParam[0]), &ItemNum);
if (Err == SCPI_ERR_NONE)
{
  // ItemNum contains number of item entered
}
```

where **sParam[0]** is the first parameter returned by **SCPI_Parse()**.

Since you know the Character Data Sequence used for that parameter (it is in your *cmds.c* specifications), you can determine what choice was made, e.g. if your sequence is *BUS*|*IMMediate*|*EXTernal*, and the item number returned was 1 then the item entered was *IMMediate* (item numbers start at 0 for the first item in the list).

If **Err** is not **SCPI_ERR_NONE** then an error occurred. See Appendix B for a list of error codes and more information.

16.5.3.4 Converting a String Parameter or an Unquoted String Parameter

If a parameter returned is type String or Unquoted String, you will want to know the contents of the string entered. In both cases, use this Access Function:

```
SCPI_ParamToString(..)
```

In code, it can be used like this:

```
UCHAR Err;
char *SString;
SCPI_CHAR_IDX Len;
char Delimiter;
    :
Err = SCPI_ParamToString (&(sParam[1]), &SString, &Len,
&Delimiter);
```

```
if (Err == SCPI_ERR_NONE)
{
  // SString is a pointer to the string entered
  // Len contains the length of the string
  // Delimiter contains the character used to delimit the
  //    string (only applies to quoted strings)
}
```

where **sParam[1]** is the second parameter returned by **SCPI_Parse()**.

As always, **Err** will be **SCPI_ERR_NONE** if the conversion was ok, otherwise it will contain an error code. See Appendix B for a full list of error codes and more information.

If **Err** is **SCPI_ERR_NONE** then 3 parameters are returned:

- The first return parameter is **SString**. This is a pointer to the start of the string entered. It will always point to a character within the array of characters used in the call to **SCPI_Parse()** that parsed this parameter (e.g. the array of characters **SInput**, if the code used is as shown in *"16.4 Parsing Loop"*). For this reason, the array of characters used must be not be over-written or discarded until you have used the string parameter.

  Note that, for parameters of type String (i.e. quoted strings), this parameter points to the first character *after* the delimiting quote.

- The second return parameter is **Len**. This contains the length of the string entered as a number of characters. This is required as the string is not null-terminated.

  Note that, for parameters of type String (i.e. quoted strings), the value of this parameter excludes the delimiting quotes.

- The third return parameter is **Delimiter**. This is only used when the parameter being converted is type (quoted) String. **Delimiter** will contain the character used to delimit the string, either single quote (**'**) or double quote (**"**). This is useful to know, since the same quote may also appear within the string. If the string is delimited by double quotes, then the user can represent a double quote within the string by entering it twice, for example, the string ***"Say ""Hello"" to John"***. This represents the text:

  Say "Hello" to John

  In the same way, a single quote can be represented in a string delimited by single quotes by entering it twice. *Note that, if the quote within the string is not the same type as the delimiting quote, then it does not need to be doubled up in this way.*

  So when you come to interpret the (quoted) string, remember to check the contents of **Delimiter**, and if you come across the 2 adjacent quote characters of the same type within the string, then treat it as a single character.

You may wish to use a call to the C string library's function **strncpy(..)** in order to copy the string into your own variable. Once copied, you can then re-use or free up the array of characters used in the call to **SCPI_Parse()**.

16.5.3.5 Converting a Numeric List Parameter

A Numeric List parameter can contain a variable number of entries. Each entry can be either a single value, or a range of values, represented by a *first* value and a *last* value separated by a colon (e.g. ***2.5:7.9***). To retrieve those values and convert them into C variables, requires these steps:

1. Retrieve an entry from the numeric list
2. Convert the entry's contents into C variable type(s)
3. Repeat the steps for each of the entries in the numeric list

To retrieve a single entry from the numeric list, use the Access Function **SCPI_GetNumListEntry(...)**

It can be used like this:

```
UCHAR Err;
UCHAR Index = 0;
BOOL bRange;
struct strParam sFirst, sLast;
Err = SCPI_GetNumListEntry (&(sParam[0]), Index, &bRange,
        &sFirst, &sLast);
```

where **Index** is the number of the entry in the numeric list to retrieve (the first entry is 0)

and where **sParam[0]** is the first parameter returned by **SCPI_Parse()**.

Err will be **SCPI_ERR_NONE** if no errors occurred. If **Err** is **SCPI_ERR_NO_ENTRY**, then there is no entry with the given Index number. This can be used to limit a loop that is retrieving each entry from a numeric list. For other error codes, see Appendix B.

If **Err** is **SCPI_ERR_NONE** then 3 parameters are returned:

- **bRange**. This is returned as **TRUE** if the entry in the numeric list is a range (e.g. ***1:23***), or **FALSE** if the entry is a single number (e.g. ***15***).
- **sFirst** is a returned parameter structure. If **bRange** is **TRUE**, then it contains the first value in the range of the entry. If **bRange** is **FALSE** then **sFirst** contains the only value in the entry. Note, the value is returned as a parameter structure rather than a numeric C variable type since you may want to use the value as an integer, a double-precision floating-point number or whatever. This allows total flexibility in retrieving the values entered.
- **sLast** is a returned parameter structure. If **bRange** is **TRUE**, then it contains the last value in the range of the entry. If **bRange** is **FALSE** then **sLast** is not used.

Now you have retrieved the contents of the entry in **sFirst** (and **sLast** if **bRange** is **TRUE**), you can convert the parameters into numeric C variables as you require. For example if you want to convert them into doubles then you can use this code:

```
UCHAR Err;
double fdValFirst, fdValLast;
Err = SCPI_ParamToDouble (&sFirst, &fdValFirst);
if (bRange)
   Err = SCPI_ParamToDouble (&sLast, &fdValLast);
```

For example, using this code, if the entry was **1:23**, then **fdValFirst** would be 1 and **fdValLast** would be 23.

Use whatever **SCPI_ParamTo...()** function you require for the type of numeric variable you want, e.g. **SCPI_ParamToInt()**, etc.

In addition, if you want to, you can call **SCPI_ParamToString()** to return the string of characters that make up the numeric list. See "*B.11 SCPI_ParamToString()*" for information.

16.5.3.6 Converting a Channel List Parameter

A Channel List parameter can contain a variable number of entries. Each entry can be either a single value, a multi-dimension value, a range of values, represented by a *first* value and a *last* value separated by a colon (e.g. ***2.5:7.9***), or even a range of multi-dimension values (e.g. 1!2:7!5). To retrieve those values and convert them into C variables, requires these steps:

1. Retrieve an entry from the channel list
2. Convert the entry's contents into C variable type(s)
3. Repeat the steps for each of the entries in the channel list

To retrieve a single entry from the channel list, use the Access Function **`SCPI_GetChanListEntry(...)`**

It can be used like this:

```
UCHAR Err = SCPI_ERR_NONE;
UCHAR Index = 0;
UCHAR DimCnt = MAX_DIMS;
BOOL bRange;
struct strParam sFirst[MAX_DIMS], sLast[MAX_DIMS];
Err = SCPI_GetChanListEntry (&(sParams[0]), Index, &DimCnt,
        &bRange, sFirst, sLast);
```

where **`Index`** is the number of the entry in the numeric list to retrieve (first entry is 0), where **`DimCnt`** is the maximum dimensions allowed in an entry, and where **`sParam[0]`** is the first parameter returned by **`SCPI_Parse()`**.

`Err` will be **`SCPI_ERR_NONE`** if no errors occurred. If **`Err`** is **`SCPI_ERR_NO_ENTRY`**, then there is no entry with the given Index number. This can be used to limit a loop that is retrieving each entry from a channel list.

Err will be **`SCPI_ERR_INVALID_DIMS`** if any entries in the channel list have too many or too few dimensions (according to the minimum and maximum limits set in the Channel List Type of the parameter specification). For other error codes, see Appendix B.

If **`Err`** is **`SCPI_ERR_NONE`** then 4 parameters are returned:

- **`DimCnt`** contains the number of dimensions present in the entry. Note that if the entry is a range, then the dimensions of the first and last values in the range must be the same.
- **`bRange`**. This is returned as **`TRUE`** if the entry in the numeric list is a range (e.g. ***1:23*** or ***3!2:4!7***), or **`FALSE`** if the entry is a single value (e.g. ***15*** or ***12!4!7***).
- **`sFirst[]`** is an array of returned parameter structures. Each element of the array corresponds to one of the dimensions of the entry. For example, if there is one dimension (**`DimCnt`**==1) then only **`sFirst[0]`** will be populated. If the entry has two dimensions (**`DimCnt`**==2) then **`sFirst[0]`** will contain the value of the first dimension and **`sFirst[1]`** will contain the value of the second dimension.

  If **`bRange`** is **`TRUE`**, then each of these **`sFirst`** parameter structures contains the first value in the range of the entry. If **`bRange`** is **`FALSE`** then the **`sFirst`** element contains the only value in the entry. Note, the value is returned as a parameter structure rather than a numeric C variable type since you may want to use the value

as an integer, a double-precision floating-point number or whatever. This allows total flexibility in retrieving the values entered.

- **sLast[]** is an array of returned parameter structures. The elements of the array are used for each dimension, in the same way as **sFirst[]**. If **bRange** is **TRUE**, then the element contains the last value in the range of the entry. If **bRange** is **FALSE** then **sLast[]** is not used.

Now you have retrieved the contents of the entry in **sFirst[]** (and **sLast[]** if **bRange** is **TRUE**), you can convert the parameters into numeric C variables as you require. For example if you want to convert them into doubles then you could use this code:

```
UCHAR Dim;
double fdValFirst[MAX_DIMS], fdValLast[MAX_DIMS];
for (Dim = 0; Dim < DimCnt; Dim++)
{
 Err = SCPI_ParamToDouble (&(sFirst[Dim]), &(fdValFirst[Dim]));
 if (bRange)
  Err = SCPI_ParamToDouble (&(sLast[Dim]), &(fdValLast[Dim]));
}
```

You can use whatever **SCPI_ParamTo...()** function you require for the type of numeric variable you want, e.g. **SCPI_ParamToInt()**, etc.

16.5.3.6.1 A Few Conversion Examples

Channel Lists can be confusing, particularly if entries are ranges rather than single values, and/or have multiple dimensions.

Here are a few example entries, and how they convert to C variables using the code above.

Entry	General Info		Returned Values		
	Dimensions	Range?	bRange	sFirst[]	sLast[]
15.7	*1*	*No*	FALSE	sFirst[0]==1	N/A
12!14	*2*	*No*	FALSE	sFirst[0]==12 sFirst[1]==14	N/A
13.4:17.6	*1*	*Yes*	TRUE	sFirst[0]==13.4	sLast[0]==17.6
1!4:2!17	*2*	*Yes*	TRUE	sFirst[0]==1 sFirst[1]==4	sLast[0]==2 sLast[1]==17
1.5!3.7!4.2:3.4!5.6!7.8	*3*	*Yes*	TRUE	sFirst[0]==1.5 sFirst[1]==3.7 sFirst[2]==4.2	sLast[0]==3.4 sLast[1]==5.6 sLast[2]==7.8

In addition to this approach, you can perform the parsing yourself if you wish, by calling access function **SCPI_ParamToString()** to return the string of characters that make up the channel list. See "*B.11 SCPI_ParamToString()*" for information.

For more information on channel lists refer to section "*A.3.8.7 Channel List*" and look at the SCPI Standard (V1999.0 section 8.3.2).

16.5.3.7 Converting an Expression Parameter

If a parameter returned is type Expression then you can retrieve the contents of the expression in string form for your own use. It uses the same Access Function as for String parameters:

```
SCPI_ParamToString(..)
```

In code, it can be used like this:

```
UCHAR Err;
char *SExpr;
SCPI_CHAR_IDX Len;
char Dummy;
    :
Err = SCPI_ParamToString (&(sParam[1]), &SExpr, &Len, &Dummy);
if (Err == SCPI_ERR_NONE)
{
  // SString is a pointer to the string entered
  // Len contains the length of the string
  // Dummy is not used
}
```

where **sParam[1]** is the second parameter returned by **SCPI_Parse()**.

As always, **Err** will be **SCPI_ERR_NONE** if the conversion was ok, otherwise it will contain an error code. See Appendix B for a full list of error codes and more information.

If **Err** is **SCPI_ERR_NONE** then two parameters are returned:

- The first return parameter is **SExpr**. This is a pointer to the start of the expression string entered. It will always point to a character within the array of characters used in the call to **SCPI_Parse()** that parsed this parameter (e.g. the array of characters **SInput**, if the code used is as shown in *"16.4 Parsing Loop"*). For this reason, the array of characters used must be not be over-written or discarded until you have used the string parameter. The first character pointed to will always be the opening bracket (**(**) of the expression.
- The second return parameter is **Len**. This contains the length of the expression string as a number of characters. This is required as the string is not null-terminated. Note, **Len** includes the terminating closing bracket (**)**).
- The third return parameter is **Dummy**. This is not used when the Access Function is used for parameters of type Expression.

You may wish to use a call to the C string library's function **strncpy(..)** in order to copy the string into your own variable. Once copied, you can then re-use or free up the array of characters used in the call to **SCPI_Parse()**.

16.5.4 Validate Parameters

Now that the parameters have been converted into native C variable types, you can perform whatever validation checks you need, e.g. checking that a value entered is within allowable range.

16.5.5 Act on Command

The final job of your command handler function is to perform some action(s) according to the command entered and the values of the parameters. JPA-SCPI Parser has provided your code with this information. What your code does now depends on your system's design. For instance, you may simply perform an action straight away, or you may queue a task to be carried out when all previous tasks have been performed.

17 Advanced Topics

You may have a specific need for your SCPI parser, but are unsure how to go about implementing it. The sections here deal with certain situations you may encounter.

17.1 How can I Support Nested Optional Parameters?

In some cases you may have nested square brackets around parameters in your SCPI notation, e.g.:

```
CONF:CURRent:DC  [ {<range>|MIN|MAX} [ , {<res>|MIN|MAX} ] ]
```

This allows commands to be entered such as:

CONF:CURR:DC
CONF:CURR:DC 1KV
CONF:CURR:DC MAX,100MV

In other words, the second parameter may only be entered if the first parameter is entered as well.

What if the command was specified slightly differently in SCPI notation? i.e.:

```
CONF:CURRent:DC  [ {<range>|MIN|MAX} ] [ , {<res>|MIN|MAX} ]
```

In this case, parameter 2 may be entered without parameter 1, allowing commands such as:

CONF:CURR:DC
CONF:CURR:DC 1KV
CONF:CURR:DC ,100MV
CONF:CURR:DC MAX,100MV

Notice the third command – this was not possible with the first form of SCPI notation.

JPA-SCPI Parser supports both forms of notation. In fact it does not distinguish between the two. In both cases, parameter 1 and parameter 2 are both classed as *optional*. By default, JPA-SCPI Parser will allow all the forms of command entry of the 2nd case, i.e. un-nested square brackets.

If you want to enforce nested square brackets, i.e. the 1st case, then you can easily do so in your code after the command has been parsed.

Use the **`SCPI_ParamType()`** Access Function to determine which parameters were entered. Disallow commands where the second parameter was entered but the first parameter was not.

17.2 How do I Support the UNIT Subsystem?

SCPI's **`UNIT`** subsystem comprises a small set of commands for changing the default units used when sending commands. For instance, take a look at this sequence of commands sent:

```
UNIT:POWER DBM              // sets default units for power levels to dbm
SOURCE:POWER:LEVEL 100      // sets output power level to 100dbm
UNIT:POWER V                // sets default units for power levels to volts
```

SOURCE:POWER:LEVEL 100 // sets output power level to 100 volts

SOURCE:POWER:LEVEL 10W // sets output power level to 10 watts

The **UNIT:POWer** command changes the default units of any subsequent commands that take parameters of power. Notice, though, the last command sent; even though default units were set to Volts, the *10W* over-rode this and set the power level to 10 watts.

The **UNIT** subsystem has lower-level nodes for setting other units as well as power, e.g. **CURRent**, **TIMe**, **VOLTage**, etc. See the SCPI Standard for full details.

If you wish to support the **UNIT** subsystem, then you will need to carry out the following steps:

17.2.1 Specify the UNIT Commands Supported

Look at the SCPI Standard and decide which commands in the **UNIT** subsystem you want to support.

For each one you will need to specify a character data sequence and character data type for the set of units allowed. For instance, if you are implementing the **UNIT:POWer** command, you will need to specify a character data sequence such as this:

```
CHDAT_SEQ SeqPowerUnits[] = "W|V|DBNW|DBUW|DBMw|DBW";
```

and a character data type such as this:

```
CHDAT_TYPE sPowerUnits     = { SeqPowerUnits, NO_DEF, ALT_NONE };
```

Now specify the command in the command keywords section of *cmds.c*, e.g.:

```
"UNIT:POWer",
```

And specify the command parameters specification in *cmds.c*, e.g.:

```
{{ { REQ  CH_DAT  sPowerUnits   },{ NOP                          } }},
```

17.2.2 Create Alternative Units

Create an entry in the Alternative Units section of *cmds.c* that lists all the possible units allowed for the commands that will be affected by the **UNIT** command. For instance, if the **UNIT:POWer** command allows units of **W|V|DBNV|DBUV|DBMw|DBW**, then you need this alternative units list:

```
ALT_UNITS_LIST eAltPower[] = {U_VOLT, U_WATT, U_DB_W, U_END};
```

Note, only the base units are needed in this list. If any of the base units are not yet defined, define them in the usual way in *cmds.h* and specify the unit strings recognized in *cmds.c*, as usual.

17.2.3 Create a Numeric Value Type for Unit Choices

Create a Numeric Value Type in *cmds.c* that will be used for parameters of commands that are affected by the default units selection. The Numeric Value Type must not have any default units. Continuing the example above, you could specify this Numeric Value Type:

```
NUM_TYPE sPower  = { U_NONE,  eAltPower,  0 };
```

17.2.4 Specifying Command Parameters

You can now specify the parameters for each of the commands affected by the **UNIT** command. Use your Numeric Value Type in the same way as normal, for instance to specify

a numerical parameter that uses the Numeric Value Type in the example above, your parameter specification would be:

```
{ REQ   NUM   sPower          }
```

17.2.5 Implementation Requirements

Your instrument now supports the **UNIT** subsystem commands required. You have also specified the Numeric Value parameters of commands affected so that they can be entered with any of the allowed units or no units at all.

If such a parameter is entered without units, the command should be used as if the units specified by the last **UNIT** command were entered.

To keep track of the default units set by any **UNIT** commands you will need to maintain a variable and update its value when a **UNIT** command is received.

17.3 How can I allow entry of either a Numeric Value or an Expression Parameter?

As explained already, JPA-SCPI Parser has built-in support for parameters that can take either Character Data or a different type of parameter. This makes use of the Alternative Parameter Type feature of the Character Data parameter specification. But what if you want to allow entry of either a Numeric Value or an Expression? There is no Alternative Parameter Type facility here, but it is still easily possible.

In fact, the following approach applies to all types of parameter – it is possible to allow entry of any combination of parameter types that you require.

This method makes use of the fact that JPA-SCPI Parser's **SCPI_Parse()** function attempts to match each command specification in turn, starting at the first entry in **sSpecCommand[]** (*cmds.c*). When a valid match of both command keywords and parameters is found, then the searching stops and the results of the match are returned.

To allow entry of different types of parameter to the same command:

1. Create duplicate entries of the command keywords in the **SSpecCmdKeywords[]** array (*cmds.c*). You need as many entries as there are different types of parameter.
2. Create corresponding duplicate entries of the command specifications in **sSpecCommand[]** (*cmds.c*), but with a different parameter type for each one.

For example, say we want to implement this command specification:

```
APPLy[:SOURce]:VOLTage[:LEVel] <value>|<expression>
```

We first specify the command keywords twice:

```
const char *SSpecCmdKeywords[] =
{
        :
  "APPLy[:SOURce]:VOLTage[:LEVel]",          /*  15      */
  "APPLy[:SOURce]:VOLTage[:LEVel]",          /*  16      */
        :
}
```

We now create the two versions of the command specification, one for each type of command parameter:

```
const struct strSpecCommand sSpecCommand[] =
{
        :
  {{ { REQ  NUM   sNoUnits } }},              /*  15     */
  {{ { REQ  EXPR          } }},              /*  16     */
        :
}
```

In your code that calls Access Function **SCPI_Parse()**, you can determine what type of parameter was entered by checking the command specification number that it returns. For example, in the example above, **SCPI_Parse()** would return *15* as the command specification number if the user entered a numeric value, or *16* if the user entered an expression.

Caution: Be careful how you order your command specifications. For instance, if you wanted to allow both a Channel List and an Expression then ensure the command specification in **sSpecCommand[]** for the Channel List is before the specification for the Expression. This is because a Channel List parameter is also always a valid Expression parameter, and **SCPI_Parse()** returns the first match that is valid. Put your more specific type of parameter *before* your more general type of parameter (e.g. Unquoted String, Expression).

17.4 Commands that allow Many Parameters

Occasionally you may want to support some commands that can take a long list of parameters. You can specify up to 255 parameters, by defining **MAX_PARAMS** in *cmds.h* to be the maximum number of parameters allowed by any command.

When supporting a large number of parameters for some commands, there are two issues:

- Extra Memory (ROM) usage
- Readability in *cmds.c*

17.4.1 Extra Memory Usage

JPA-SCPI Parser defines the parameters allowed by each command as an array of structures (indices **0** to **MAX_PARAM-1**). This means that if **MAX_PARAM** is increased, the memory used by JPA-SCPI Parser to store the parameter specifications will also increase. This approach of fixed size arrays, rather than using dynamic structures, was taken in order to allow the parameter specifications to be defined as constants and therefore able to reside in ROM rather than RAM; RAM is often in short supply in embedded systems.

Each unused (**NOP**) parameter specification occupies 8 bytes of ROM[1]. So, for instance, if your instrument supports 60 commands and 56 of them accept 1 or 2 parameters, but you

[1] Figures obtained when compiling for a Microchip PIC18C452 with the HiTech C-18 compiler. Size will vary with platform/compiler.

need **MAX_PARAMS** to be 10, then you have 448 (56*(10-2)) unused parameters using 3.5KBytes of ROM[1].

If the extra ROM space required is a significant problem you may wish to consider altering the command specifications, e.g. instead of having commands that can accept a large number of parameters, specify a command that takes a single parameter and adds it to a queue of parameters.

17.4.2 Readability

In *cmds.c*, the section of code titled *"Command Specs – Part 2: Parameters"* contains a table with sets of columns representing command parameters. The more command parameters allowed, the more columns are required and the wider the table becomes.

If most of your commands accept 1 or 2 parameters, but a few commands accept, say, 10 parameters, then it is a waste of space to give each row of the table 10 full-size sets of parameter columns.

One suggestion you may like to follow is this:

1 Subtract the maximum number of parameters accepted by most of your commands from **MAX_PARAM**. For example, if most of your commands take 0, 1 or 2 parameters and **MAX_PARAM** is 10, then this number is 10 – 2 = **8**.

2 Define a symbol representing a set of **NOP** parameters. The number of **NOP** parameters should be the same as the figure obtained in the previous step. If the number obtained was 8, then, for example:

```
#define NOPx8 {NOP},{NOP},{NOP},{NOP},{NOP},{NOP},{NOP},{NOP}
```

3 Include this symbol after the parameter specifications for all the commands that take a small number of commands. Effectively, this adds parameter columns to each command without taking up much space in the row. For example, this command specification only takes 1 parameter and **MAX_PARAMS** is defined as 10:

```
{{ {REQ CH_DAT sMinMaxOhms},{NOP                    },NOPx8 }},
```

4 For the commands that do allow many parameters, specify their parameters in the normal way.

Often, the parameters of these types of commands are all the same type. If this is the case, you could define a short symbol to represent the parameter type and then repeat it in the command specification. For example:

```
#define VNUM    {OPT NUM sVolts}
```

thus allowing you to use this command specification:

```
{{ VNUM,VNUM,VNUM,VNUM,VNUM,VNUM,VNUM,VNUM,VNUM,VNUM }},
```

Of course, these are all just ideas and you may prefer a different approach.

Appendices

Appendix A – An Introduction to SCPI

A.1 Benefits of SCPI

The SCPI Standard was defined in order to provide a consistent command language for all types of remotely programmable instruments. In doing so, SCPI aims to reduce significantly the learning curve required by a technician to be able to program a particular instrument.

In addition, SCPI defines specific 'core' command sets for some types of instrument, such as digital meters, signal switchers, etc. By defining a set of commands that must be supported by certain types of equipment, it means that equipment from different manufacturers should be almost interchangeable in an ATE system, for instance.

As well as saving time for the customer's technicians to learn to control a new piece of equipment, SCPI also has benefits for the instrument manufacturer too. It provides a framework for defining the command set of an instrument and therefore saves time designing a proprietary command structure and syntax. It may also save time supporting the instrument, since many technicians are now familiar with SCPI and so will be able to grasp the command syntax straight away.

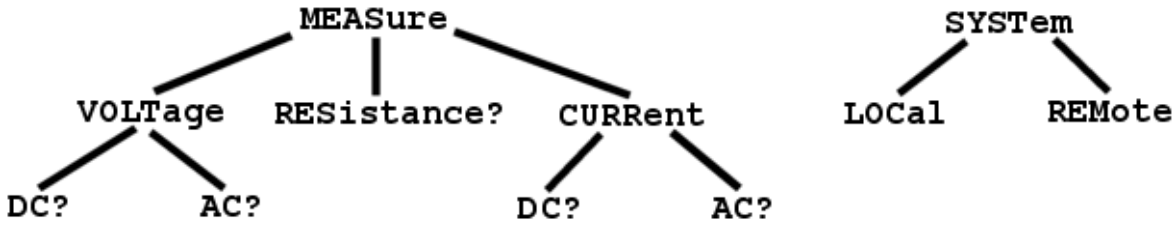
In practice, many manufacturers choose to support a *SCPI-like* interface, rather than implementing all of the features required by SCPI to claim SCPI compliancy. This is a valid approach, since the look-and-feel of SCPI will again reduce the technician's time spent learning the instrument. Either approach, SCPI-like or full SCPI compliancy, is possible using JPA-SCPI Parser.

A.2 Background to SCPI

SCPI was developed, and is still being expanded, by the SCPI Consortium (http:\\www.scpiconsortium.org). It uses another standard IEEE488.2 as its basis, except that SCPI is usable whatever the physical interface used (e.g. GPIB, RS232, USB, etc.), whereas IEEE488.2 only applies to the GPIB (IEEE488.1) interface.

A.3 Command Structure

SCPI commands are hierarchical, being based on a *tree system*, for example:



The *nodes* of the tree represent *command keywords*, e.g. **`MEASure`**, **`VOLTage`**, **`DC?`**, **`RESistance`**.

At the top of the trees are the *root nodes*, i.e. **`MEASure`** and **`SYStem`**. Under each root node is what is known as a *subsystem*.

A command is formed by traversing the tree from a root node downwards until a node is reached with no further nodes below it.

Instead of drawing command trees, SCPI uses a *notation* to represent command specifications. When writing a command, the root node is written first, followed by the keywords on the lower levels. Colons (:) are used to separate keywords on different levels of the tree. For example:

```
MEASure:VOLTage:DC?
```

or:

```
SYSTem:LOCal
```

In addition, SCPI notation represents the levels of the tree by the horizontal indentation of the keywords. The root node is in the leftmost position and so on. The commands represented by the diagram above would be written as:

```
MEASure
  :VOLTage
    :DC?
    :AC?
  :RESistance?
  :CURRent
    :DC?
    :AC?
SYSTem
  :LOCal
  :REMote
```

A.3.1 Long and Short Form Keywords

You will see that many of the keywords above have upper and lowercase letters. This system is used to represent the *long form* and *short form* of each keyword. The long form of the keyword comprises all the characters of the keyword. The short form is made up of just the characters in uppercase.

For instance, for **`MEASure`** above:

- Long Form is **`MEASURE`**
- Short Form is **`MEAS`**

Usually, the short form of a keyword comprises the first four letters. However, if the fourth letter is a vowel, then the short form normally only uses the first 3 letters (e.g. **`CALibration`**).

Commands sent to a SCPI instrument can include any combination of long and short form keywords. For the command set above, all of these commands are valid:

MEASURE:VOLTAGE:DC?

MEAS:VOLT:DC?

MEASURE:VOLT:DC?

MEAS:VOLTAGE:DC?

Note, that commands sent to a SCPI instrument are case-insensitive, for example, all of these are also valid commands:

Meas:Volt:DC?

measure:Voltage:dc?

By convention, however, example SCPI commands are usually shown in uppercase form. This is what we use in this manual.

A.3.2 Query Commands

Any SCPI commands that expect data to be sent back over the remote interface are termed *query commands*. Such commands might request the voltage reading from a digital voltmeter, or request the identity of the instrument.

All query commands end in a question mark. In the command set above, query commands include:

MEASure:VOLTage:DC?

MEASure:RESistance?

etc.

A.3.3 Default Keywords

To shorten command entry, SCPI allows the use of *default keywords* (also known as *default nodes*). These are keywords that can be left out of commands without affecting the meaning of the command.

Default keywords are shown in command specifications by enclosing them in square brackets. For example, the keywords of a command specification might be:

APPly:[SOURce:]CURRent[:LEVel][:IMMediate]:AMPLitude

In this case, valid forms of the command include:

APPLY:SOURCE:CURRENT:LEVEL:IMMEDIATE:AMPLITUDE

APP:CURR:AMPL

APP:CURRENT:LEV:AMPLITUDE

etc.

Notice how a colon can be included within the square brackets, in order that all possible command constructs will contain keywords separated by a single colon.

A.3.4 Numeric Suffices

An instrument may have more than one outputs, trigger sources, etc. In order to specify which of these channels a command is referring to, a numeric suffix can be added to the command. For instance, a command to set the voltage range on a multi-channel oscilloscope might be specified as:

[SENSe:]VOLTage[:DC]<channel#>:RANGe {<voltage>|MIN|MAX}

To set channel 1 to 200mV range, the user might enter this command as:

VOLT1:RANG 200MV

To set channel 2 to 10V range, the user could enter this command:

SENS:VOLT2:RANG 10V

In addition, if the user does not enter a numeric suffix, then the value 1 is assumed. So in this case, these two commands are equivalent, both setting channel 1 to the 2V range:

VOLT1:DC:RANG 2V

VOLT:DC:RANG 2V

Note, a command may have more than one numeric suffix. For example, this command might be used to set the 2nd FM signal component of the 3rd output channel:

OUTP3:FM2

A.3.5 Compound Commands

It is possible to send compound commands to a SCPI instrument. Commands are separated within a command line by a semi-colon (*;*). For example:

MEAS:VOLT:DC?;AC?

Note that the second command is not the full command but rather it uses the command tree that was reached by the command before it. This shorthand is used by SCPI to reduce the length of command lines.

If you need to use a command higher up the tree, then the second command must be prefixed by a colon (:). This has the effect of resetting the command tree to the root. The next command must therefore be in its full form, for example:

MEAS:VOLT:DC?;:MEAS:CURR:DC?

or:

MEAS:VOLT:DC?;:SYST:LOC

A.3.6 IEEE488.2 Common Commands

SCPI-compliant instruments must support a small set of *IEEE488.2 common commands* defined in the IEEE488.2 standard. These include:

`*RST`

`*CLS`

IEEE488.2 common commands are used to reset the device, query its status registers, reset the interface etc. For more information on these commands, refer to the SCPI Standard and/or the IEEE488.2 Standard.

A.3.7 Parameters

Many commands take one or more parameters in order to provide the instrument with more information, for example, the voltage level to set on a programmable power supply, or the resolution to use on a digital resistance meter.

Parameters appear after the command keywords, separated from the keywords by a space (no spaces are allowed within the command keywords).

If more than one parameter is allowed by the command, each parameter is separated by a comma.

For example, a command might be sent as:

MEAS:VOLT? 1KV, 10MV

This would take a measurement from a digital voltmeter using a range of 1 kilovolts, and with a resolution of 10 millivolts. The parameters are *1KV* and *10MV*.

A.3.8 Types of Parameter

The different types of parameter allowed are:

A.3.8.1 Numeric Value

This is a number with or without units. It could represent a voltage, a frequency, a count – anything that can be represented numerically.

Numeric Values can also include units after the number. These are usually optional.

In the example above, both *1KV* and *10MV* are Numeric Values.

A.3.8.1.1 Number Bases

SCPI allows Numeric Values to be entered in some number bases other than decimal: binary, octal and hexadecimal.

When entering a number in one of these other bases, the number must be prefixed to indicate the base.

Base	Prefix	Example
Binary	**#B**	*#B11001010* = 202_{10}
Octal	**#Q**	*#Q107* = 71_{10}
Hexadecimal	**#H**	*#H10FF* = 4351_{10}

Note: Negative and real (non-integer) numbers are only allowable in decimal.

A.3.8.2 Boolean

Sometimes, a parameter is required to set a state to on or off, e.g. auto-ranging on a digital meter can either be on or off. Parameters that just require two states are known as *Booleans*. Boolean parameters can be entered in a number of ways:

```
ON
OFF
1                    // same as entering ON
0                    // same as entering OFF
```

In addition, SCPI allows entry of a number where a Boolean parameter is permitted. The number is converted to **ON** (1) or **OFF** (0) according to these rules:

- The sign of the number is ignored
- The number is rounded to the nearest integer, where .5 and above is rounded upwards
- If the resulting number is zero, the Boolean parameter is **OFF** (0). Otherwise the Boolean parameter is **ON** (1).

Boolean parameters can be specified with a *default value*, i.e. if the parameter is not entered it is equivalent to the parameter being entered with the default value. In SCPI notation, default values are shown in bold type (or you can use underline if bold type is not available). For example:

```
{ON|OFF}
```

Here, the default value is **ON**.

A.3.8.3 Character Data

SCPI also allows mnemonics to be entered as parameters. These are called *Character Data* parameters. For example, the command specification:

TRIGger:SOURce {BUS|IMMediate|EXTernal}

The possible values of the Character Data parameter are: **BUS**, **IMM**, **IMMEDIATE**, **EXT** or **EXTERNAL**.

In addition, Character Data choices are often combined with another type of parameter, e.g. a Numeric Value or Boolean parameter.

For example, the specification of a command to set the resistance range of an ohmmeter might be:

SENSe:RESistance:RANGe {<range>|MINimum|MAXimum}

This command allows entries such as:

SENS:RES:RANG 1000

SENS:RES:RANG 1GOHM

SENS:RES:RANG MAX

SENS:RES:RANG MINIMUM

As with Boolean parameters, Character Data parameters can have a default value. For example:

{BUS|IMMediate|EXTernal}

Here, **IMMediate** is the default value used if the parameter is not entered.

A.3.8.4 String

Occasionally, an instrument may wish to accept a parameter made up of a string of characters. For example, a command to display a text message on the instrument's readout.

Strings in SCPI must be delimited by quotes (either double or single). For instance, a command specification such as:

DISPlay:TEXT <message string>

would accept commands such as:

DISP:TEXT "hello world"

DISP:TEXT 'Set function to "Volts".'

In addition, JPA-SCPI Parser also supports a type we call *Unquoted Strings*. These function exactly the same as normal strings except that they do not require quotes to delimit them. Instead they are delimited by the commas (if any) that surround any parameter.

Unquoted Strings are useful for entry of passwords, for instance, to allow access to calibration factors or maintenance functions.

For example, the command specification:

CALibration:SECure:CODE <code>

would accept entries such as:

CAL:SEC:CODE ABC123

A.3.8.5 Expression

SCPI defines various other types of parameter as expressions. These types include:

- Numeric Expressions, e.g. (15*5+4)
- Numeric Lists, e.g. (1,2,3:7,9)
- Channel Lists, e.g. (@1!3,2!4:5!5)
- DIF (Data Interchange Format) Expressions

All these types of expression start with an opening bracket ('**(**') and end with a closing bracket ('**)**').

Support for expressions is optional in SCPI. In fact most instrument do not support them. DIF, for instance, is used for transferring large amounts of data from an instrument to a computer, so is useful for logging instruments etc.

Numeric Lists and Channel Lists are amongst the most useful of the expressions, and these are explained further below.

A.3.8.6 Numeric List

A numeric list is used to allow entry of a variable number of numeric values and ranges of numeric values.

The format of a numeric list is:

(*<entry>*[,*<entry>*[,*<entry>*....]]])

where ***<entry>*** has the format:

<numeric value>*|*<numeric value>*:*<numeric value>

Ranges are indicated by the first number in the range and last number in the range separated by a colon (**:**).

For example, a numeric list could be:

(5,7:17,20.5)

This numeric list has 3 entries: the value 5, the range 7 through to 17, and the value 20.5.

Note, the order of entries in a numeric list does not matter – there is no order implied by the ordering of the entries in the numeric list.

A.3.8.7 Channel List

A channel list is used to specify a set of electrical ports on an instrument. The most common use is for specifying signal routing and switching.

The format of a channel list is:

(@*<entry>*[,*<entry>*[,*<entry>*....]]])

where ***<entry>*** has the format:

<channel spec>*|*<channel spec>*:*<channel spec>

Ranges are indicated by the first number in the range and last number in the range separated by a colon (**:**).

In addition to allowing ranges of values just like numerical lists, channel list entries can have more than one dimension. A two dimensional entry comprises the value of the first dimension followed by a '**!**' symbol followed by the value of the second dimension.

The specification for ***<channel spec>*** is:

***<numeric value>*[!*<numeric value>*[!*<numeric value>*....]]]**

For example, a channel spec could be:

3

or *5!6*

or *4!7!9*

The number of **!** symbols is one less than the number of dimensions, so the last example above has 3 dimensions.

Dimensions are useful for representing a matrix of switches, for example. Say you have 10 rows and 12 columns of switches. The 1st dimension represents the row number and the 2nd dimension represents the column number:

Row	Column											
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
1	1!1	1!2	1!3	1!4	1!5	1!6	1!7	1!8	1!9	1!10	1!11	1!12
2	2!1	2!2	2!3	2!4	2!5	2!6	2!7	2!8	2!9	2!10	2!11	2!12
3	3!1	3!2	3!3	3!4	3!5	3!6	3!7	3!8	3!9	3!10	3!11	3!12
4	4!1	4!2	4!3	4!4	4!5	4!6	4!7	4!8	4!9	4!10	4!11	4!12
5	5!1	5!2	5!3	5!4	5!5	5!6	5!7	5!8	5!9	5!10	5!11	5!12
6	6!1	6!2	6!3	6!4	6!5	6!6	6!7	6!8	6!9	6!10	6!11	6!12
7	7!1	7!2	7!3	7!4	7!5	7!6	7!7	7!8	7!9	7!10	7!11	7!12
8	8!1	8!2	8!3	8!4	8!5	8!6	8!7	8!8	8!9	8!10	8!11	8!12
9	9!1	9!2	9!3	9!4	9!5	9!6	9!7	9!8	9!9	9!10	9!11	9!12
10	10!1	10!2	10!3	10!4	10!5	10!6	10!7	10!8	10!9	10!10	10!11	10!12

A channel list to specify the switch at row 2, column 3 and row 9, column 11 would be:

(@2!3,9!11)

As mentioned above you can also specify ranges in a channel list. For a single dimensional channel list, this is exactly the same as a numeric list, e.g. for the range 5 through to 11, then channel list would be:

(@5:11)

But what happens when you want to specify a range of values in a 2 dimensional, or multi-dimensional channel list?

For instance, say we wanted to specify the switches in the table below that are shown with the grey background. We pick the first element in the group (*3!3*) and the last element (*7!11*) and separate them with a colon (**:**), i.e.:

3!3:7!11

This tells the instrument to operate on all the switches in the area marked. Not only that, but the order of operation is also implied by the order of the values in the range. In that example it means start at 3!3, then 3!4 and so on until 3!11. Now continue with 4!3 through to 4!11, and so on until 7!3 through to 7!11.

If we wanted to operate in reverse order then we would simply reverse the order of the numbers, i.e.:

7!11:3!3

Unlike a numeric list, the order of operation with entries in a channel list is implied by the order of the entries. For example:

(@1!3,2!5:3!1,4!4)

means operate in the following order:

1!3, 2!5, 2!4, 2!3, 2!2, 2!1, 3!5, 3!4, 3!3, 3!2, 3!1, 4!4

As well as numeric entries, SCPI allows channel lists to include alphanumeric entries such as module specifiers and path names. These are not very common in use and are beyond the scope of this introduction to SCPI. You may wish to refer to the SCPI Standard for more information.

Appendix B – JPA-Parser Access Functions

This appendix describes each of the JPA-SCPI Parser Access Functions.

Note*: In-bound* parameters are passed by value – they are not changed by the function. *Out-bound* parameters are passed by reference – they may be changed by the function. Parameters that are both *in-* and *out-bound* are also passed by reference. Their value is used by the function and may be returned modified.

B.1 SCPI_Parse()

```
#ifdef SUPPORT_NUM_SUFFIX

UCHAR SCPI_Parse (char **pSInput, BOOL bResetTree, SCPI_CMD_NUM
*pCmdSpecNum, struct strParam sParam[], UCHAR *pNumSufCnt,
unsigned int uiNumSuf[]);

#else

UCHAR SCPI_Parse (char **pSInput, BOOL bResetTree, SCPI_CMD_NUM
*pCmdSpecNum, struct strParam sParam[]);

#endif
```

B.1.1 Description

Parses a command in the command line string. If a match is found then returns number of matching command specification, and returns values and attributes of any parameters entered. Also returns any numeric suffices entered (if numeric suffix support is enabled).

B.1.2 Parameters

Parameter	In/Out-Bound?	Description
`pSInput`	In & Out	Pointer to first character of the command line string to be parsed. The string must be a null-terminated string of length 255 or less, unless the maximum command length has been increased from the default. See *12.5 Option to Support More than 255 Commands* for details. Parameter is returned modified, so as to point to first character of the next command in the command line to be parsed.
`bResetTree`	In	If TRUE then the command tree is reset to the root node; if FALSE then the command tree stays at the node set by the previous command. Note: Set this to TRUE when parsing the first command of the command line string, and FALSE otherwise.
`pCmdSpecNum`	Out	Pointer to returned number of the command specification that matches the command in the command line string that was parsed. Value is undefined if no matching command specification is found.
`sParam[]`	Out	Array [0..MAX_PARAM-1] of returned parameters containing the parsed parameter values and attributes. Contents of returned parameters are undefined if no matching command specification is found.
`pNumSufCnt`	Out	Pointer to returned count of numeric suffices encountered

uiNumSuf[]	Out	Array [0..MAX_NUM_SUFFIX-1] of returned numeric suffices

B.1.3 Return Value

Value	Meaning
SCPI_ERR_NONE	OK. A matching command specification was found
SCPI_ERR_NO_COMMAND	Error. There was no command to be parsed in the command line string
SCPI_ERR_INVALID_CMD	Error. The command keywords did not match any command specification command keywords.
SCPI_ERR_PARAM_CNT	Error. The command keywords match a command specification but the wrong number of parameters was given in the command.
SCPI_ERR_PARAM_TYPE	Error. A parameter within the command does not match a valid type of parameter for the command specification.
SCPI_ERR_PARAM_UNITS	Error: A parameter within the command has the wrong type of units for the command specification.
SCPI_ERR_PARAM_OVERFLOW	Error. The command contains a parameter of type Numeric Value that was too large to be stored internally. This occurs if the value has an exponent greater than +/-43.
SCPI_ERR_UNMATCHED_BRACKET	Error. The parameters in the command contain an unmatched bracket.
SCPI_ERR_UNMATCHED_QUOTE	Error. The parameters in the command contain an unmatched single or double quote.
SCPI_ERR_TOO_MANY_NUM_SUF	Error. Too many numeric suffices in the command to be returned in uiNumSuf[].
SCPI_ERR_NUM_SUF_INVALID	Error. One or more numeric suffix in the command is invalid, e.g. out of range.
SCPI_ERR_INVALID_VALUE	Error. One or more values in a numeric/channel list parameter is invalid, e.g. floating point when not allowed
SCPI_ERR_INVALID_DIMS	Error. One or more entries in a channel list parameter has an invalid number of dimensions.

B.1.4 Example Code (SUPPORT_NUM_SUFFIX is not #defined)

```
char SCmdLine[256];
    :
UCHAR Err;
char *SCmd = SCmdLine;
BOOL bResetTree = TRUE;
SCPI_CMD_NUM CmdNum;
struct strParam sParams[MAX_PARAMS];
    :
do
{
  Err = SCPI_Parse (&SCmd, bResetTree, &CmdNum, sParams);
    :
  bResetTree = FALSE;
} while (Err == SCPI_ERROR_NONE);
```

B.2 SCPI_ParamType()

```
UCHAR SCPI_ParamType (struct strParam *psParam, enum
enParamType *pePType, UCHAR *pNumSubtype);
```

B.2.1 Description

Returns the type of a parameter returned by **`SCPI_Parse()`**. If parameter is type Numeric Value, then also returns its sub-type attributes.

B.2.2 Parameters

Parameter	In/Out-Bound?	Description
`psParam`	In	Pointer to parameter returned by **`SCPI_Parse()`** (must not be null)
`pePType`	Out	Pointer to returned type of parameter
`pNumSubtype`	Out	Pointer to returned parameter's sub-type attributes, if parameter is type *Numeric Value*:

Bit Number	Use
7-2	*Not Used*
1	1=Real number, 0=Integer
0	1=Negative number, 0=Positive

B.2.3 Return Value

Value	Meaning
`SCPI_ERR_NONE`	OK (always returned by this function)

B.2.4 Example Code

```
UCHAR Err;
enum enParamType ePType;
UCHAR NumSubtype;
Err = SCPI_ParamType (&(sParams[0]), &ePType, &NumSubtype)
if (Err == SCPI_ERR_NONE)
{
  if (ePType == P_NUM)// Numeric Value
  {
    if (NumSubType & SCPI_NUM_ATTR_NEG)
      // Value is negative
    else
      // Value is positive
    if (NumSubType & SCPI_NUM_ATTR_REAL)
      // Value is real
    else
      // Value is an integer
  }
}
```

where **`sParam[0]`** is the first parameter returned by **`SCPI_Parse()`**.

B.3 SCPI_ParamUnits()

```
UCHAR SCPI_ParamUnits (struct strParam *psParam, enum enUnits
*peUnits);
```

B.3.1 Description

Returns the units of a parameter of type Numeric Value.

B.3.2 Parameters

Parameter	In/Out-Bound?	Description
psParam	In	Pointer to parameter returned by **SCPI_Parse()** (must not be null)
peUnits	Out	Pointer to returned type of units of the parameter

B.3.3 Return Value

Value	Meaning
SCPI_ERR_NONE	OK
SCPI_ERR_PARAM_TYPE	Error. Parameter is not of type Numeric Value

B.3.4 Example Code

```
UCHAR Err;
enum enUnits eUnits;
    :
Err = SCPI_ParamUnits (&(sParam[0]), &eUnits);
```

where **sParam[0]** is the first parameter returned by **SCPI_Parse()**.

B.4 SCPI_ParamToCharDataItem()

```
UCHAR SCPI_ParamToCharDataItem (struct strParam *psParam, UCHAR
*pItemNum);
```

B.4.1 Description

Converts a parameter of type Character Data into its Character Data Item Number.

B.4.2 Parameters

Parameter	In/Out-Bound?	Description
psParam	In	Pointer to parameter returned by **SCPI_Parse()** (must not be null)
pItemNum	Out	Pointer to returned parameter's Character Data Item Number. Item Number is 0 for first item in Character Data Sequence.

B.4.3 Return Values

Value	Meaning
SCPI_ERR_NONE	OK
SCPI_ERR_PARAM_TYPE	Error. Parameter is not of type Character Data

B.4.4 Example Code

```
UCHAR Err;
UCHAR ItemNum;
    :
Err = SCPI_ParamToCharDataItem (&(sParam[0]), &ItemNum);
if (Err == SCPI_ERR_NONE)
{
  // ItemNum contains number of item entered
}
```

where **sParam[0]** is the first parameter returned by **SCPI_Parse()**.

B.5 SCPI_ParamToBOOL()

```
UCHAR SCPI_ParamToBOOL (struct strParam *psParam, BOOL *pbVal);
```

B.5.1 Description

Converts a parameter of type Boolean into a BOOL

B.5.2 Parameters

Parameter	In/Out-Bound?	Description
psParam	In	Pointer to parameter returned by **SCPI_Parse()** (must not be null)
pbVal	Out	Pointer to returned BOOL value

B.5.3 Return Value

Value	Meaning
SCPI_ERR_NONE	OK
SCPI_ERR_PARAM_TYPE	Error. Parameter is not of type Boolean

B.5.4 Example Code

```
UCHAR Err;
BOOL bVal;
    :
Err = SCPI_ParamToBool (&(sParam[0]), &bVal);
if (Err == SCPI_ERR_NONE)
{
  // bVal contains Boolean value of parameter
}
```

where **sParam[0]** is the first parameter returned by **SCPI_Parse()**.

B.6 SCPI_ParamToUnsignedInt()

```
UCHAR SCPI_ParamToUnsignedInt (struct strParam *psParam,
unsigned int *puiVal);
```

B.6.1 Description

Converts a parameter of type Numeric Value into an unsigned integer. If parameter's value is negative, then sign is ignored. If parameter's value is real (non-integer) then digits after the decimal point are ignored.

B.6.2 Parameters

Parameter	In/Out-Bound?	Description
`psParam`	In	Pointer to parameter returned by `SCPI_Parse()` (must not be null)
`puiVal`	Out	Pointer to returned unsigned int value

B.6.3 Return Value

Value	Meaning
`SCPI_ERR_NONE`	OK
`SCPI_ERR_PARAM_TYPE`	Error. Parameter is not of type Numeric Value
`SCPI_ERR_PARAM_OVERFLOW`	Error. Value cannot be stored in a variable of type *unsigned int*

B.6.4 Example Code

```
UCHAR Err;
unsigned int uiVal;
    :
Err = SCPI_ParamToUnsignedInt (&(sParam[1]), &uiVal);
if (Err == SCPI_ERR_NONE)
{
  // uiVal contains unsigned integer value of parameter
}
```

where `sParam[1]` is the second parameter returned by `SCPI_Parse()`.

B.7 SCPI_ParamToInt()

```
UCHAR SCPI_ParamToInt (struct strParam *psParam, int *piVal);
```

B.7.1 Description

Converts a parameter of type Numeric Value into a signed integer. If parameter's value is real (non-integer) then digits after the decimal point are ignored.

B.7.2 Parameters

Parameter	In/Out-Bound?	Description
psParam	In	Pointer to parameter returned by **SCPI_Parse()** (must not be null)
piVal	Out	Pointer to returned signed int value

B.7.3 Return Value

Value	Meaning
SCPI_ERR_NONE	OK
SCPI_ERR_PARAM_TYPE	Error. Parameter is not of type Numeric Value
SCPI_ERR_PARAM_OVERFLOW	Error. Value cannot be stored in a variable of type *int*

B.7.4 Example Code

```
UCHAR Err;
int iVal;
    :
Err = SCPI_ParamToInt (&(sParam[0]), &iVal);
if (Err == SCPI_ERR_NONE)
{
  // iVal contains integer value of parameter
}
```

where **sParam[0]** is the first parameter returned by **SCPI_Parse()**.

B.8 SCPI_ParamToUnsignedLong()

```
UCHAR SCPI_ParamToUnsignedLong (struct strParam *psParam,
unsigned long *pulVal);
```

B.8.1 Description

Converts a parameter of type Numeric Value into an unsigned long. If parameter's value is negative, then sign is ignored. If parameter's value is real (non-integer) then digits after the decimal point are ignored.

B.8.2 Parameters

Parameter	In/Out-Bound?	Description
`psParam`	In	Pointer to parameter returned by `SCPI_Parse()` (must not be null)
`pulVal`	Out	Pointer to returned unsigned long value

B.8.3 Return Value

Value	Meaning
`SCPI_ERR_NONE`	OK
`SCPI_ERR_PARAM_TYPE`	Error. Parameter is not of type Numeric Value
`SCPI_ERR_PARAM_OVERFLOW`	Error: Value cannot be stored in a variable of type *unsigned long*

B.8.4 Example Code

```
UCHAR Err;
unsigned long ulVal;
    :
Err = SCPI_ParamToUnsignedLong (&(sParam[1]), &ulVal);
if (Err == SCPI_ERR_NONE)
{
  // ulVal contains unsigned long integer value of parameter
}
```

where `sParam[1]` is the second parameter returned by `SCPI_Parse()`.

B.9 SCPI_ParamToLong()

```
UCHAR SCPI_ParamToLong (struct strParam *psParam, long *plVal);
```

B.9.1 Description

Converts a parameter of type Numeric Value into a signed long. If parameter's value is real (non-integer) then digits after the decimal point are ignored.

B.9.2 Parameters

Parameter	In/Out-Bound?	Description
`psParam`	In	Pointer to parameter returned by `SCPI_Parse()` (must not be null)
`plVal`	Out	Pointer to returned signed long value

B.9.3 Return Value

Value	Meaning
`SCPI_ERR_NONE`	OK
`SCPI_ERR_PARAM_TYPE`	Error. Parameter is not of type *Numeric Value*
`SCPI_ERR_PARAM_OVERFLOW`	Error. Value cannot be stored in variable of type *long*

B.9.4 Example Code

```
UCHAR Err;
long lVal;
    :
Err = SCPI_ParamToLong (&(sParam[0]), &lVal);
if (Err == SCPI_ERR_NONE)
{
  // lVal contains long integer value of parameter
}
```

where `sParam[0]` is the first parameter returned by `SCPI_Parse()`.

B.10 SCPI_ParamToDouble()

```
UCHAR SCPI_ParamToDouble (struct strParam *psParam, double
*pfdVal);
```

B.10.1 Description

Converts a parameter of type Numeric Value into a double-precision float.

B.10.2 Parameters

Parameter	In/Out-Bound?	Description
psParam	In	Pointer to parameter returned by **SCPI_Parse()** (must not be null)
pfdVal	Out	Pointer to returned double value

B.10.3 Return Value

Value	Meaning
SCPI_ERR_NONE	OK
SCPI_ERR_PARAM_TYPE	Error. Parameter is not of type Numeric Value

B.10.4 Example Code

```
UCHAR Err;
double fdVal;
    :
Err = SCPI_ParamToDouble (&(sParam[0]), &fdVal);
if (Err == SCPI_ERR_NONE)
{
  // fdVal contains double-precision floating-point
  // value of parameter
}
```

where **sParam[0]** is the first parameter returned by **SCPI_Parse()**.

B.11 SCPI_ParamToString()

```
UCHAR SCPI_ParamToString (struct strParam *psParam, char
**pSString, SCPI_CHAR_IDX *pLen, char *pDelimiter);
```

B.11.1 Description

Converts a parameter of type String, Unquoted String, Expression, Numeric List or Channel List into a pointer to a string of characters, and a character count.

B.11.2 Parameters

Parameter	In/Out-Bound?	Description
`psParam`	In	Pointer to parameter returned by `SCPI_Parse()` (must not be null)
`pSString`	Out	Returned pointer to an array of characters containing the returned string. Note: The array of characters pointed to is always within the command line string that contained the command parameter; the command line string must therefore still be valid when calling this function.
`pLen`	Out	Pointer to number of characters within returned string
`pDelimiter`	Out	Pointer to character containing the symbol used to delimit the string. Only applies to parameter of type String (quoted).

B.11.3 Return Values

Value	Meaning
`SCPI_ERR_NONE`	OK
`SCPI_ERR_PARAM_TYPE`	Error. Parameter was not type String, Unquoted String or Expression.

B.11.4 Example Code

```
char *SString;
UCHAR Err;
SCPI_CHAR_IDX Len;
char Delimiter;
char MyString[256];
    :
Err = SCPI_ParamToString (&(sParams[0]), &SString, &Len,
                          &Delimiter);
if (Err == SCPI_ERR_NONE)
{
  strncpy (MyString, SString, Len);  // Copy into MyString
  MyString[Len] = '\0';              // Null terminate MyString
}
```

where `sParam[0]` is the first parameter returned by `SCPI_Parse()`.

B.12 SCPI_GetNumListEntry()

This function is only implemented if SUPPORT_NUM_LIST is #defined

```
UCHAR SCPI_GetNumListEntry (struct strParam *psParam, UCHAR
Index, BOOL *pbRange, struct strParam *psFirst, struct
strParam *psLast);
```

B.12.1 Description

Returns an entry from a Numeric List parameter in the form of one or two (if it is a range) numeric value parameters. The numeric value parameters returned can be converted into C variables using other Access Functions – **`SCPI_ParamToDouble()`**, **`SCPI_ParamToUnsignedInt()`**, etc.

B.12.2 Parameters

Parameter	In/Out-Bound?	Description
`psParam`	In	Pointer to parameter returned by **`SCPI_Parse()`** (must not be null)
`pbRange`	Out	Pointer to returned flag: TRUE means entry is a range of values; FALSE means entry is a single value.
`psFirst`	Out	Pointer to returned parameter containing entry's value (or first value in range if *pbRange==TRUE).
`psLast`	Out	Pointer to returned parameter containing entry's last value in range - only used if *pbRange==TRUE.

B.12.3 Return Values

Value	Meaning
`SCPI_ERR_NONE`	OK
`SCPI_ERR_NO_ENTRY`	Error. There was no entry to get - the index was beyond the end of the entries.
`SCPI_ERR_PARAM_TYPE`	Error. Parameter is not of type Numeric List

B.12.4 Example Code

```
UCHAR Err;
BOOL bRange;
struct strParam sFirst, sLast;
    :
Err = SCPI_GetNumListEntry (&(sParams[1]), 0, &bRange,
                            &sFirst, &sLast);
```

where **`sParam[1]`** is the second parameter returned by **`SCPI_Parse()`**.

B.13 SCPI_GetChanListEntry()

This function is only implemented if SUPPORT_CHAN_LIST is #defined

```
UCHAR SCPI_GetChanListEntry (struct strParam *psParam, UCHAR
Index, UCHAR *pDims, BOOL *pbRange, struct strParam sFirst[],
struct strParam sLast[]);
```

B.13.1 Description

Returns an entry from a Channel List parameter in the form of one or two (if it is a range) arrays of numeric value parameters. The numeric value parameters returned can be converted into C variables using the other Access Functions – **`SCPI_ParamToDouble()`**, **`SCPI_ParamToUnsignedInt()`**, etc.

B.13.2 Parameters

Parameter	In/Out-Bound?	Description
`psParam`	In	Pointer to parameter returned by **`SCPI_Parse()`** (must not be null)
`pbRange`	Out	Pointer to returned flag: TRUE means entry is a range of values; FALSE means entry is a single value.
`pDims`	In/Out	Inwards: Pointer to maximum dimensions possible in an entry; returned as the number of dimensions in the entry.
`sFirst[]`	Out	Array [0..*pDims-1] of returned parameters containing the entry's value (or its first value in the range if *pbRange == TRUE)
`sLast[]`	Out	Array [0..Dims-1] of returned parameters containing entry's last value in range - only used if *pbRange == TRUE

B.13.3 Return Values

Value	Meaning
`SCPI_ERR_NONE`	OK
`SCPI_ERR_NO_ENTRY`	Error. There was no entry to get - the index was beyond the end of the entries.
`SCPI_ERR_TOO_MANY_DIMS`	Error. Too many dimensions in the entry to be returned in the parameters.
`SCPI_ERR_PARAM_TYPE`	Error. Parameter is not of type Channel List

B.13.4 Example Code

```
UCHAR Err = SCPI_ERR_NONE;

UCHAR Index = 0;

UCHAR DimCnt = MAX_DIMS;

UCHAR Dim;

BOOL bRange;

struct strParam sFirst[MAX_DIMS], sLast[MAX_DIMS];

Err = SCPI_GetChanListEntry (&(sParams[0]), Index, &DimCnt,
       &bRange, sFirst, sLast);
```

where **`sParam[0]`** is the first parameter returned by **`SCPI_Parse()`**.

Appendix C – SCPI Instrument Class Templates

This appendix lists the commands supported by each of the supplied SCPI Instrument Class templates. For more information see *"6.1 SCPI Instrument Classes Introduced"*. For information on each command, refer to the SCPI Standard.

Note:

As well as the commands listed in this appendix, every instrument class template also includes the commands of the SCPI Base Class template.

C.1 DC Voltmeter

Template Location: ***code**{format}**template**dcvmet***

C.1.1 Command Set

ABORt

CONFigure
[:SCALar]:VOLTage:DC [<range>|MIN|MAX [,<resolution>|MIN|MAX]]

CONFigure?

FETCh
[:SCALar]:VOLTage:DC? [<range>|MIN|MAX [,<resolution>|MIN|MAX]]

INITiate
[:IMMediate][:ALL]

MEASure
[:SCALar]:VOLTage:DC? [<range>|MIN|MAX [,<resolution>|MIN|MAX]]

READ
[:SCALar]:VOLTage:DC? [<range>|MIN|MAX [,<resolution>|MIN|MAX]]

SENSe
:FUNCtion[:ON] {"VOLTage:DC"}
:FUNCtion[:ON]?
:VOLTage:DC:RANGe[:UPPer] {<range>|MIN|MAX}
:VOLTage:DC:RANGe[:UPPer]?
:VOLTage:DC:RANGe:AUTO {ON|OFF}
:VOLTage:DC:RANGe:AUTO?
:VOLTage:DC:RESolution {<resolution>|MIN|MAX}
:VOLTage:DC:RESolution?

TRIGger
[:SEQuence]:COUNt {<value>|MIN|MAX}
[:SEQuence]:COUNt?
[:SEQuence]:DELay {<period>|MIN|MAX}
[:SEQuence]:DELay?
[:SEQuence]:SOURce {BUS|IMMediate|EXTernal}
[:SEQuence]:SOURce?

***TRG**

SYSTem
:CAPability?

Notes

1. <range> and <resolution> are numeric values with units defined as Volts.
2. <value> is a numeric value with no units.
3. <period> is a numeric value with units defined as seconds.
4. You may wish to make the **SENSe** subsystem the default node. Do this by enclosing it in square brackets in the command specification's command keywords, i.e. **[SENSe:]**.
5. You may wish to make all occurrences of **:VOLTage:DC** optional keywords if your instrument only supports DC voltage measurements. Do this by enclosing it in square brackets in the command spec's command keywords, i.e. **[:VOLTage:DC]**

C.2 AC RMS Voltmeter

Template Location: ***\code\{format}\template\acvmet***

C.2.1 Command Set

```
ABORt
CONFigure
  [:SCALar]:VOLTage:AC [<range>|MIN|MAX [,<resolution>|MIN|MAX]]
CONFigure?
FETCh
  [:SCALar]:VOLTage:AC? [<range>|MIN|MAX [,<resolution>|MIN|MAX]]
INITiate
  [:IMMediate][:ALL]
MEASure
  [:SCALar]:VOLTage:AC? [<range>|MIN|MAX [,<resolution>|MIN|MAX]]
READ
  [:SCALar]:VOLTage:AC? [<range>|MIN|MAX [,<resolution>|MIN|MAX]]
SENSe
  :FUNCtion[:ON] {"VOLTage:AC"}
  :FUNCtion[:ON]?
  :VOLTage:AC:RANGe[:UPPer] {<range>|MIN|MAX}
  :VOLTage:AC:RANGe[:UPPer]?
  :VOLTage:AC:RANGe:AUTO {ON|OFF}
  :VOLTage:AC:RANGe:AUTO?
  :VOLTage:AC:RESolution {<resolution>|MIN|MAX}
  :VOLTage:AC:RESolution?
TRIGger
  [:SEQuence]:COUNt {<value>|MIN|MAX}
  [:SEQuence]:COUNt?
  [:SEQuence]:DELay {<period>|MIN|MAX}
  [:SEQuence]:DELay?
  [:SEQuence]:SOURce {BUS|IMMediate|EXTernal}
  [:SEQuence]:SOURce?
*TRG
SYSTem
  :CAPability?
```

Notes

1. <range> and <resolution> are numeric values with units defined as Volts.
2. <value> is a numeric value with no units.
3. <period> is a numeric value with units defined as seconds.
4. You may wish to make the **SENSe** subsystem the default node. Do this by enclosing it in square brackets in the command specification's command keywords, i.e. **[SENSe:]**.
5. You may wish to make all occurrences of **:VOLTage:AC** optional keywords if your instrument only supports AC voltage measurements. Do this by enclosing it in square brackets in the command specification's command keywords, i.e. **[:VOLTage:AC]**

C.3 DC Ammeter

Template Location: ***code**{format}**template**dcimet***

C.3.1 Command Set

ABORt

CONFigure
[:SCALar]:CURRent:DC [<range>|MIN|MAX [,<resolution>|MIN|MAX]]

CONFigure?

FETCh
[:SCALar]:CURRent:DC? [<range>|MIN|MAX [,<resolution>|MIN|MAX]]

INITiate
[:IMMediate][:ALL]

MEASure
[:SCALar]:CURRent:DC? [<range>|MIN|MAX [,<resolution>|MIN|MAX]]

READ
[:SCALar]:CURRent:DC? [<range>|MIN|MAX [,<resolution>|MIN|MAX]]

SENSe
:FUNCtion[:ON] {"CURRent:DC"}
:FUNCtion[:ON]?
:CURRent:DC:RANGe[:UPPer] {<range>|MIN|MAX}
:CURRent:DC:RANGe[:UPPer]?
:CURRent:DC:RANGe:AUTO {ON|OFF}
:CURRent:DC:RANGe:AUTO?
:CURRent:DC:RESolution {<resolution>|MIN|MAX}
:CURRent:DC:RESolution?

TRIGger
[:SEQuence]:COUNt {<value>|MIN|MAX}
[:SEQuence]:COUNt?
[:SEQuence]:DELay {<period>|MIN|MAX}
[:SEQuence]:DELay?
[:SEQuence]:SOURce {BUS|IMMediate|EXTernal}
[:SEQuence]:SOURce?

***TRG**

SYSTem
:CAPability?

Notes

1. <range> and <resolution> are numeric values with units defined as Amps.
2. <value> is a numeric value with no units.
3. <period> is a numeric value with units defined as seconds.
4. You may wish to make the **SENSe** subsystem the default node. Do this by enclosing it in square brackets in the command specification's command keywords, i.e. **[SENSe:]**.
5. You may wish to make all occurrences of **:CURRent:DC** optional keywords if your instrument only supports DC current measurements. Do this by enclosing it in square brackets in the command specification's command keywords, i.e. **[:CURRent:DC]**

C.4 AC RMS Ammeter

Template Location: ***\code\{format}\template\acimet***

C.4.1 Command Set

```
ABORt
CONFigure
  [:SCALar]:CURRent:AC [<range>|MIN|MAX [,<resolution>|MIN|MAX]]
CONFigure?
FETCh
  [:SCALar]:CURRent:AC? [<range>|MIN|MAX [,<resolution>|MIN|MAX]]
INITiate
  [:IMMediate][:ALL]
MEASure
  [:SCALar]:CURRent:AC? [<range>|MIN|MAX [,<resolution>|MIN|MAX]]
READ
  [:SCALar]:CURRent:AC? [<range>|MIN|MAX [,<resolution>|MIN|MAX]]
SENSe
  :FUNCtion[:ON] {"CURRent:AC"}
  :FUNCtion[:ON]?
  :CURRent:AC:RANGe[:UPPer] {<range>|MIN|MAX}
  :CURRent:AC:RANGe[:UPPer]?
  :CURRent:AC:RANGe:AUTO {ON|OFF}
  :CURRent:AC:RANGe:AUTO?
  :CURRent:AC:RESolution {<resolution>|MIN|MAX}
  :CURRent:AC:RESolution?
TRIGger
  [:SEQuence]:COUNt {<value>|MIN|MAX}
  [:SEQuence]:COUNt?
  [:SEQuence]:DELay {<period>|MIN|MAX}
  [:SEQuence]:DELay?
  [:SEQuence]:SOURce {BUS|IMMediate|EXTernal}
  [:SEQuence]:SOURce?
*TRG
SYSTem
  :CAPability?
```

Notes

1. <range> and <resolution> are numeric values with units defined as Amps.
2. <value> is a numeric value with no units.
3. <period> is a numeric value with units defined as seconds.
4. You may wish to make the **SENSe** subsystem the default node. Do this by enclosing it in square brackets in the command specification's command keywords, i.e. **[SENSe:]**.
5. You may wish to make all occurrences of **:CURRent:AC** optional keywords if your instrument only supports AC current measurements. Do this by enclosing it in square brackets in the command specification's command keywords, i.e. **[:CURRent:AC]**

C.5 Ohmmeter

Template Location: ***code**{format}**template**ohmmet***

C.5.1 Command Set

```
ABORt
CONFigure
  [:SCALar]:RESistance [<range>|MIN|MAX [,<resolution>|MIN|MAX]]
CONFigure?
FETCh
  [:SCALar]:RESistance? [<range>|MIN|MAX [,<resolution>|MIN|MAX]]
INITiate
  [:IMMediate][:ALL]
MEASure
  [:SCALar]:RESistance? [<range>|MIN|MAX [,<resolution>|MIN|MAX]]
READ
  [:SCALar]:RESistance? [<range>|MIN|MAX [,<resolution>|MIN|MAX]]
SENSe
  :FUNCtion[:ON] {"RESistance"}
  :FUNCtion[:ON]?
  :RESistance:RANGe[:UPPer] {<range>|MIN|MAX}
  :RESistance:RANGe[:UPPer]?
  :RESistance:RANGe:AUTO {ON|OFF}
  :RESistance:RANGe:AUTO?
  :RESistance:RESolution {<resolution>|MIN|MAX}
  :RESistance:RESolution?
TRIGger
  [:SEQuence]:COUNt {<value>|MIN|MAX}
  [:SEQuence]:COUNt?
  [:SEQuence]:DELay {<period>|MIN|MAX}
  [:SEQuence]:DELay?
  [:SEQuence]:SOURce {BUS|IMMediate|EXTernal}
  [:SEQuence]:SOURce?
*TRG
SYSTem
  :CAPability?
```

Notes

1. <range> and <resolution> are numeric values with units defined as Ohms.
2. <value> is a numericl value with no units.
3. <period> is a numeric value with units defined as seconds.
4. You may wish to make the **SENSe** subsystem the default node. Do this by enclosing it in square brackets in the command specification's command keywords, i.e. **[SENSe:]**.
5. You may wish to make all occurrences of **:RESistance** an optional keyword if your instrument only supports 2-wire resistance measurements. Do this by enclosing it in square brackets in the command specification's command keywords, i.e. **[:RESistance]**

C.6 4-wire Ohmmeter

Template Location: ***\code\{format}\template\4wohmmet***

C.6.1 Command Set

```
ABORt
CONFigure
  [:SCALar]:FRESistance [<range>|MIN|MAX [,<resolution>|MIN|MAX]]
CONFigure?
FETCh
  [:SCALar]:FRESistance? [<range>|MIN|MAX [,<resolution>|MIN|MAX]]
INITiate
  [:IMMediate][:ALL]
MEASure
  [:SCALar]:FRESistance? [<range>|MIN|MAX [,<resolution>|MIN|MAX]]
READ
  [:SCALar]:FRESistance? [<range>|MIN|MAX [,<resolution>|MIN|MAX]]
SENSe
  :FUNCtion[:ON] {"FRESistance"}
  :FUNCtion[:ON]?
  :FRESistance:RANGe[:UPPer] {<range>|MIN|MAX}
  :FRESistance:RANGe[:UPPer]?
  :FRESistance:RANGe:AUTO {ON|OFF}
  :FRESistance:RANGe:AUTO?
  :FRESistance:RESolution {<resolution>|MIN|MAX}
  :FRESistance:RESolution?
TRIGger
  [:SEQuence]:COUNt {<value>|MIN|MAX}
  [:SEQuence]:COUNt?
  [:SEQuence]:DELay {<period>|MIN|MAX}
  [:SEQuence]:DELay?
  [:SEQuence]:SOURce {BUS|IMMediate|EXTernal}
  [:SEQuence]:SOURce?
*TRG
SYSTem
  :CAPability?
```

Notes

1. <range> and <resolution> are numeric values with units defined as Ohms.
2. <value> is a numeric value with no units.
3. <period> is a numeric value with units defined as seconds.
4. You may wish to make the **SENSe** subsystem the default node. Do this by enclosing it in square brackets in the command specification's command keywords, i.e. **[SENSe:]**.
5. You may wish to make all occurrences of **:FRESistance** an optional keyword if your instrument only supports 4-wire resistance measurements. Do this by enclosing it in square brackets in the command specification's command keywords, i.e. **[:FRESistance]**

C.7 Power Supply

Template Location: ***\code\{format}\template\powersup***

C.7.1 Command Set

```
OUTPut
  [:STATe] {ON|OFF}

[SOURce:]
  CURRent[:LEVel][:IMMediate][:AMPLitude] {<current>|MIN|MAX}
  VOLTage[:LEVel][:IMMediate][:AMPLitude] {<voltage>|MIN|MAX}

SYSTem
  :CAPability?
```

Notes

1. <current> is a numeric value with units defined as Amps.
2. <voltage> is a numeric value with units defined as Volts.

C.8 Digitizer

Template Location: ***\code\{format}\template\digitizr***

C.8.1 Command Set

```
INPut#
  :COUPling {AC|DC|GND}
  :COUPling?

[SENSe:]
  VOLTage#[:DC]:RANGe:LOWer {<voltage>|MIN|MAX}
  VOLTage#[:DC]:RANGe:LOWer?
  VOLTage#[:DC]:RANGe:OFFSet {<voltage>|MIN|MAX}
  VOLTage#[:DC]:RANGe:OFFSet?
  VOLTage#[:DC]:RANGe:PTPeak {<voltage>|MIN|MAX}
  VOLTage#[:DC]:RANGe:PTPeak?
  VOLTage#[:DC]:RANGe[:UPPER] {<voltage>|MIN|MAX}
  VOLTage#[:DC]:RANGe[:UPPER]?
  SWEep:POINts {<number>|MIN|MAX}
  SWEep:POINts?
  SWEep:TIME {<period>|MIN|MAX}
  SWEep:TIME?
  SWEep:TINTerval {<period>|MIN|MAX}
  SWEep:TINTerval?
  DATA? ["XTIMe:VOLTage#[:DC]"]
  FUNCtion:CONCurrent {ON|OFF}
  FUNCtion:CONCurrent?
  FUNCtion:OFF {"XTIMe:VOLTage#[:DC]"}
  FUNCtion:OFF?
  FUNCtion[:ON] {"XTIMe:VOLTage#[:DC]"}
  FUNCtion[:ON]?
  FUNCtion:STATe {"XTIMe:VOLTage#[:DC]"}
  FUNCtion:STATe?

FORMat
  [:DATA] {ASCii}[,<length>]
  [:DATA]?

INITiate
  [:IMMediate][:ALL]

ABORt

TRIGger
  [:SEQuence]:COUPling {AC|DC}
  [:SEQuence]:COUPling?
  [:SEQuence]:LEVel {<voltage>|MIN|MAX}
  [:SEQuence]:LEVel?
  [:SEQuence]:SLOPe {POSitive|NEGative|EITHer}
  [:SEQuence]:SLOPe?
  [:SEQuence]:SOURce {INTernal#}
  [:SEQuence]:SOURce?
```

```
SYSTem
  :CAPability?
```

Notes

1. <voltage> is a numeric value with units defined as Volts.
2. <period> is a numeric value with units defined as seconds.
3. <number> and <length> are numeric values without units.

C.9 Signal Switcher

Template Location: ***code**{format}**template**switcher***

C.9.1 Command Set

```
[ROUTe:]
  CLOSe <channel list>
  CLOSe? <channel list>
  CLOSe:STATe?
  OPEN <channel list>
  OPEN? <channel list>
  OPEN:ALL
SYSTem
  :CAPability?
```

Notes

1. <channel list> is a Channel List parameter that uses the **sCL1DimInts** channel list type defined in *cmds.c*, thus allowing entry of a single dimensional channel list of integer values. If this is unsuitable then select another channel list type or define your own.

C.10 RF and Microwave Source

Template Location: ***\code\{format}\template\rfmicsrc***

C.10.1 Command Set

```
[SOURce:]
  FREQuency[:CW] {<frequency>|MIN|MAX}
  FREQuecny[:CW]?
  FREQuency:FIXed {<frequency>|MIN|MAX}
  FREQuency:FIXed?
  POWer:ALC[:STATe] {ON|OFF}
  POWer:ALC[:STATe]?
  POWer[:LEVel][:IMMediate][:AMPLitude] {<power>|MIN|MAX}
  POWer[:LEVel][:IMMediate][:AMPLitude]?
OUTPut
  [:STATe] {ON|OFF}
  [:STATe]?
UNIT
  :POWer {W|V|DBNW|DBUW|DBM|DBMW|DBW}
SYSTem
  :CAPability?
```

Notes

1. <frequency> is a numeric value with units defined as Hertz.
2. The **UNIT:POWer** command allows the user to specify the default units when sending a command with the <power> parameter. Add any other units supported to the options list.
3. <power> is a numeric value with no default units, but accepting units of Watts, Volts and Decibel Watts.

   If <power> is entered with no units, your code should assume the units as specified by the most recent **UNIT:POWer** command. Initially, the default units are DBM, as specified in the SCPI Standard (in section describing the RF and Microwave Source Instrument Class.)

C.11 SCPI Base Class

Template Location: ***\code\{format}\templates\base***

The SCPI Base Class template includes the set of commands that must be supported by every instrument in order to claim SCPI-compliance.

Note: All the commands in the SCPI Base Class template are also included in every SCPI Instrument Class template.

C.11.1 Command Set

```
SYSTem
  :ERRor[:NEXT]?
  :VERSion?

STATus
  :OPERation[:EVENt]?
  :OPERation:CONDition?
  :OPERation:ENABle <value>
  :OPERation:ENABle?
  :QUEStionable[:EVENt]?
  :QUEStionable:CONDition?
  :QUEStionable:ENABle <value>
  :QUEStionable:ENABle?
  :PRESet

*CLS

*ESE <enable value>

*ESE?

*ESR?

*IDN?

*OPC

*OPC?

*RST

*SRE <enable value>

*SRE?

*STB?

*TST?

*WAI
```

Notes

1. <value> and <enable value> are both numeric values without units.

Appendix D – Sample Command Specifications

It is often useful to examine examples of source code that implement command specifications similar to your requirements for your instrument.

The JPA-SCPI Parser Demo Application (available from the JPA Consulting website: http:\\www.jpacsoft.com) demonstrates a variety of types of command specifications.

The *cmds.c* and *cmds.h* files used by the Demo Application (version 1.2) are supplied with JPA-SCPI Parser. They are located in folder: **\code\{format}\sample**

Here is the list of command specifications implemented by the *cmds.c* and *cmds.h* files of the Demo Application. Take a look through the command specifications here for one similar to your requirements, e.g. it has a similar type of parameter or uses default keywords in a similar way. Now open the *cmds.c* file used by the Demo Application to see how that command specification is implemented in code.

Note, to save space, some standard character data choices are only listed here in short form. For example, **MIN|MAX|DEF** actually means that the choices are **MINimum|MAXimum|DEFault**.

Cmd #	Command Specification
	IEEE488.2 Commands required for SCPI Compliancy
0	***CLS**
1	***ESE** <enable value (no units)>
2	***ESE?**
3	***ESR?**
4	***IDN?**
5	***OPC**
6	***OPC?**
7	***RST**
8	***SRE** <enable value (no units)>
9	***SRE?**
10	***STB?**
11	***TST?**
12	***WAI**
	Other Commands required for SCPI Compliancy
13	**SYSTem:ERRor[:NEXT]?**
14	**SYSTem:VERSion?**
15	**STATus:OPERation[:EVENt]?**
16	**STATus:OPERation:CONDition?**
17	**STATus:OPERation:ENABle** <value>
18	**STATus:OPERation:ENABle?**
19	**STATus:QUEStionable[:EVENt]?**
20	**STATus:QUEStionable:CONDition?**
21	**STATus:QUEStionable:ENABle** <value>
22	**STATus:QUEStionable:ENABle?**
23	**STATus:PRESet**
	Example Commands used by a DMM
24	**MEASure:VOLTage:DC?** **{**<range (V)>**|MIN|MAX|DEF}**, **{**<resolution (V)>**|MIN|MAX|DEF}**
25	**MEASure:VOLTage:DC:RATio?** **{**<range (V)>**|MIN|MAX|DEF}**, **{**<resolution (V)>**|MIN|MAX|DEF}**
26	**MEASure:VOLTage:AC?** **{**<range (V)>**|MIN|MAX|DEF}**, **{**<resolution (V)>**|MIN|MAX|DEF}**
27	**MEASure:CURRent:DC?** **{**<range (A)>**|MIN|MAX|DEF}**, **{**<resolution (A)>**|MIN|MAX|DEF}**
28	**MEASure:CURRent:AC?** **{**<range (A)>**|MIN|MAX|DEF}**, **{**<resolution (A)>**|MIN|MAX|DEF}**
29	**MEASure:RESistance?** **{**<range (Ω)>**|MIN|MAX|DEF}**, **{**<resolution (Ω)>**|MIN|MAX|DEF}**
30	**MEASure:FRESistance?** **{**<range (Ω)>**|MIN|MAX|DEF}**, **{**<resolution (Ω)>**|MIN|MAX|DEF}**
31	**MEASure:FREQuency?** **{**<range (Hz)>**|MIN|MAX|DEF}**, **{**<resolution (Hz)>**|MIN|MAX|DEF}**

Cmd #	Command Specification
32	**MEASure:PERiod? {**<range (s)>**|MIN|MAX|DEF}, {**<resolution (s)>**|MIN|MAX|DEF}**
33	**CONFigure:CURRent:DC {**<range (A)>**|MINimum|MAXimum|DEFault}**
34	**CONFigure:FREQuency {**<range (Hz)>**|MINimum|MAXimum|DEFault}**
35	**[SENSe:]RESistance:RANGe {**<range (Ω)>**|MINimum|MAXimum}**
36	**[SENSe:]RESistance:RANGe? [MINimum|MAXimum]**
37	**[SENSe:]RESistance:RANGe:AUTO {OFF|ON}**
38	**[SENSe:]VOLTage:DC:NPLCycles {0.02|0.2|1|10|100|MINimum|MAXimum}**
39	**[SENSe:]FUNCtion {"VOLTage:DC"|"VOLTage:DC:RATio"|"VOLTage:AC"|"CURRent:DC"|"CURRent:AC" |"RESistance"|"FRESistance"|"FREQuency"|"PERiod"|"CONTinuity"|"DIODe"}**
40	**[SENSe:]FUNCtion?**
41	**INPut:IMPedance:AUTO {OFF|ON}**
42	**INPut:IMPedance:AUTO?**
43	**CALCulate:STATe {OFF|ON}**
44	**CALCulate:FUNCtion {NULL|DB|DBM|AVERage|LIMit}**
45	**CALCulate:AVERage:MINimum?**
46	**CALCulate:AVERage:MAXimum?**
47	**READ?**
	Example Commands used by a Power Source
48	**APPLy {**<voltage (V)>**|MIN|MAX|DEF}, {**<current (A)>**|MIN|MAX|DEF}**
49	**APPLy[:SOURce]:CURRent[:LEVel][:IMMediate][:AMPLitude] {**<level (A)>**|MINimum|MAXimum|UP|DOWN}**
50	**APPLy[:SOURce]:CURRent[:LEVel][:IMMediate][:AMPLitude]? [MINimum|MAXimum]**
51	**APPLy[:SOURce]:VOLTage[:LEVel][:IMMediate][:AMPLitude] {**<level (V)>**|MINimum|MAXimum|UP|DOWN}**
52	**APPLy[:SOURce]:VOLTage[:LEVel][:IMMediate][:AMPLitude]? [MINimum|MAXimum]**
53	**APPLy[:SOURce]:INDuctance** <inductance (µH)>
54	**APPLy[:SOURce]:TEMPerature** <temperature (°K (default), °C, or °F)>
55	**INITiate[:IMMediate]**
56	**OUTPut:RELay[:STATe] {OFF|ON}**
57	**OUTPut:RELay[:STATe]?**
	Example Commands applicable to many different types of instrument
58	**TRIGger:SOURce {BUS|IMMediate|EXTernal}**
59	**TRIGger:DELay? [MINimum|MAXimum]**
60	**DISPlay:TEXT** <message string>
61	**CALibration:SECure:CODE** <passcode (unquoted string)>
62	**CALibration:SECure:STATe {OFF|ON},** <code (unquoted string)>
63	**CALibration:SECure:STATe?**
64	**CALibration:CURRent[:DATA]** <numeric value (no units)>
65	**CALibration:CURRent:LEVel {MINimum|MIDdle|MAXimum}**
66	**SYSTem:LOCal**
67	**SYSTem:REMote**
68	**STEP[:INCRement]:AUTO {OFF|ON|ONCE}**
	Miscellaneous Examples
69	**ROUTe:OPEN** <channel list>
70	**SYSTem:ERRor:ENABle[:LIST]** <numeric list>
71	**OUTPut:TTLTrg#**
72	**OUTPut#:MOD#FM#** <numeric value>
73	**TRACe:FEED:OCONdition** <expression>

Appendix E – Upgrading from a Previous Version

The current version of JPA-SCPI Parser is V1.3.1. If you are using a previous version of JPA-SCPI Parser, it is recommended that you upgrade to this latest version.

Locate the section below related to the version of JPA-SCPI Parser that you are currently using. Follow the procedure described to upgrade to the next more recent version.

E.1 Upgrading from V1.3.0

E.1.1 Summary of New Features in V1.3.1

This is a maintenance release, hence there are no new features.

E.1.2 Bug Fixes in V1.3.1

- Possible compilation error or erroneous behaviour relating to function *SCPI_ParamToString()* if not using the default definition of *SCPI_CHAR_INDEX*.

  Problem

  Incorrect definition of parameter *pLen* in function declaration *SCPI_ParamToString()*.

  Fix

  Function declaration of *SCPI_ParamToString()* corrected in scpi.h (see *Design Notes* document for more details).

- Unused parameter causing compilation size to be larger than necessary.

  Problem

  Function *TranslateParameters()* includes parameter *InpParamCnt* that is not used or required.

  Fix

  Parameter *InpParamCnt* removed from *TranslateParameters()* function and calls to the function.

- Possible compilation error or erroneous behaviour relating to structure *strAttrString* if using non-default definition of *SCPI_CHAR_INDEX*.

  Problem

  Incorrect definition of element *Len* in structure *strAttrString*.

  Fix

  Definition of element *Len* in structure *strAttrString* changed to *SCPI_CHAR_INDEX* (was *unsigned char*) in *scpi.h*.

E.1.3 Changes to Documentation in V1.3.1

E.1.3.1 User Manual

- Pages 23 & 107 – Corrected address of SCPI Consortium website to be www.scpiconsortium.org.

- Page 92 - Corrected code to be *SCPI_ParamToBOOL()*. Was *SCPI_ParamToBool()*.
- Page 96 – Explain *DimCnt* parameter's inwards requirements when calling the *SCPI_GetChanEntry()* function.

E.1.3.2 Design Notes

- Page 34 – Removed entry for parameter *InpParamCnt* from information table on function *TranslateParameters()*.

E.1.4 Procedure for Upgrading to V1.3.1

a) Replace your existing files with the new versions of *scpi.h* and *scpi.c*.

b) If you have previously made any changes to *scpi.h* or *scpi.c* then you will need to repeat those changes to the new files.

E.2 Upgrading from Older Versions

If you are using an older version than V1.3.0, you will need to perform a two step upgrade – first upgrading to V1.3.0, and then following the procedure in the previous section to upgrade to V1.3.1. For instructions of how to upgrade to V1.3.0, please refer to the *User Manual* for JPA-SCPI Parser V1.3.0. If you require further information please contact us via email: support@jpacsoft.com.

E.3 Revision History of Previous Versions

E.3.1 V1.3.0

E.3.1.1 New Features

- **Option to allow more than 255 characters in the input command line**
- **Option to allow more than 255 command definitions**
- **Enhancement to numeric suffix option.** An option to discriminate between a command entered without a numeric suffix and the same command entered with a numeric suffix of 1.

E.3.1.2 Bug Fixes

- Possible invalid value returned by function *SCPI_ParamToUnsignedLong()* in parameter **pulVal*.

  Problem

  Incorrect casting in function where parameter is assigned.

  Fix

  This incorrect line:

  ```
  *pulVal = (unsigned int)(psParam->unAttr.sNumericVal.ulSigFigs);
  ```

  is now replaced by this:

  ```
  *pulVal = psParam->unAttr.sNumericVal.ulSigFigs;
  ```

- Error in each instrument class template *cmds.c* file causing possible compilation errors.

  Problem

  Inappropriate closing bracket character at the end of three `#ifdef` lines.

  Fix

  Closing bracket characters removed, for instance:

  ```
  #ifdef SUPPORT_NUM_LIST)
  ```

  is now replaced by this:

  ```
  #ifdef SUPPORT_NUM_LIST
  ```

- Error in a command definition of the Digitizer instrument class template.

  Problem

  Command `TRIGger[:SEQuence]:SLOPe` incorrectly accepts `EIT` as the short form of valid parameter `EITHER`, whereas it should be `EITH`.

  Fix

  Correction to definition of *SeqPosNegEit[]* in Digitizer template's *cmds.c*.

E.3.2 V1.2.1

E.3.2.1 New Features

- None

E.3.2.2 Bug Fixes

- Rejection of some valid compound SCPI command strings that contain common commands. Note: non-common commands are parsed correctly.

  Problem

  Previous versions of the parser reject valid compound command strings that contain common commands (e.g.***rst**, ***wai**, ***cls**, etc.) after the first command in the string, for example:

  configure:current:dc max;*cls

  This is a valid SCPI string. The parser correctly parses the first command (**configure:current:dc max**), but parser function **SCPI_Parse()** rejects the second command (***cls**), returning **SCPI_ERR_INVALID_CMD**.

  The parser incorrectly considers the command tree reached by the previous command (**configure:current:dc max**), and rejects the common command (***cls**) since it is not present within the command tree at that level.

  This is incorrect behaviour, since the IEEE488.2 specification states that the level of the command tree reached should not be considered when a common command is parsed.

  A user workaround is to either (a) send separate commands, i.e.:

  configure:current:dc max
  ***cls**

  or (b) insert a colon (**:**) immediately before the common command:

  configure:current:dc max;:*cls

The colon returns the command tree to the root and the parser then recognises the common command (***cls**).

Fix

Parser V1.2.1 now correctly parses compound command strings that contain common commands after the first command in the string.

Note also that the place in the command tree reached before the common command is maintained for any subsequent command. For instance, this SCPI command string is valid, and is parsed correctly by the parser:

configure:current:dc max;*cls;dc min

E.3.3 V1.2.0

E.3.3.1 New Features

- **New parameter type: Numeric List.** Allows entry of a list of numeric values and numeric ranges. Provides full support for the Numeric List type as defined in the SCPI Standard.
- **New parameter type: Channel List.** Allows entry of lists of electrical ports and ranges of ports. Single and multi-dimensional entries are allowed. Full support for the Channel List type as defined in the SCPI Standard, with the exception that alphanumeric entries, such as path names and module identifiers are not supported. If these are required, then the new parameter type Expression (see below) may be used.
- **New parameter type: Expression.** Allows entry of any valid SCPI expression. The expression is validated for the correct levels of bracketing and returned as a pointer to a string. Parsing of the returned string is performed by the user's code.
- **Support for Numeric Suffices** in both command keywords and character data items. Allows commands to be defined for multi-channel instruments without the need for duplicating commands. The numeric suffices entered in the command are returned as an array of unsigned integers for use by the user's code.
- **Optional Support Features.** Using a small set of **`#define`** statements, the above support features can be individually enabled or disabled. By disabling features that are not needed, you can save ROM and RAM.
- Ability to include **optional characters in the specifications of Character Data items**. Character Data item specification can now include optional text within square brackets (**[,]**), in the same way that command keywords can.
- **Better parsing of quoted strings.** Entries can now include embedded quotes of the same kind as the delimiting quotes by use of 2 adjacent quotes (as defined in SCPI Standard). The type of quote used to delimit the string is available via a new return parameter in the **`SCPI_ParamToString()`** Access Function.
- **Parameters are now validated for the correct use of round brackets ((,)).** I.e. there must be the same number of opening and closing brackets and there must never be a negative nesting level (where there is a greater number of closing brackets to the left of any opening brackets). Note, brackets inside quotes (single or double) are not counted.
- **Better support for C++ compilers.** The *scpi.h* and *cmds.h* header files now include pre-processor directives to tell a C++ compiler that the functions are C format. This should allow easier integration of the JPA-SCPI Parser modules into a C++ project.

- **All references to *numerical values* in the code and the documentation have been changed to *numeric values*.** This is in line with the SCPI Standard document.

E.3.3.2 Bug Fixes

- None

In addition, details of the changes and additions made to the source code for this version of JPA-SCPI Parser are given in the accompanying *JPA-SCPI Parser Design Notes* document.

```
:IMMediat
asure[:SCALar][:VOLTage
"READ[:SCALar][:VOLTage
"SENSe:FUNCtion[:ON]",
"SENSe:FUNCtion[:ON]?",
"SENSe:VOLTage:DC:RANGe[:UPP
SENSe:VOLTage:DC:RANGe[:UPPe
NSe:VOLTage:DC:RANGe:AUTO",
se:VOLTage:DC:RANGe:AUTO?",
e:VOLTage:DC:RESolution",
:VOLTage:DC:RESolution?",
[:SEQuence[:COUNt",
:SEQuence[:COUNt",
SEQuence[:COUNt",
EQuence[:COUNt",
```