Applying Standard Commands for Programmable Instruments (SCPI) to CubeSats

Shaun Houlihan, Andrew Kalman

Pumpkin Inc.

744 Naples Street, San Francisco, CA 94112; (415) 548-6360

shaun@pumpkininc.com, aek@pumpkininc.com

ABSTRACT

A variety of different command and data protocols over UARTs, SPI and I^{2}C exist within subsystems designed for CubeSats. We present our implementation of the Standard Commands for Programmable Instruments (SCPI, sometimes pronounced "skippy") as applied to Pumpkin's evolving line of Supervisor MCU-based modules for nanosatellites. SCPI includes human-readability and standardized error reporting, which can be beneficial in a variety of circumstances, including preproduction testing and runtime monitoring. We present an overview of the software and hardware required to support SCPI, and its runtime performance in a production CubeSat.

INTRODUCTION

The SCPI protocol was established in the early 1990s as a way to standardize communication between instruments and controllers in a way that is extensible and easy to learn.[1] It defines a command syntax and structure that can be applied to a wide variety of test and measurement devices for control and data query.

Pumpkin has implemented SCPI as the standard command interface for its line of PIC24E Supervisor MCU-based (SupMCU) CubeSat-compatible modules, enabling configuration and telemetry capability across an entire family of CubeSat components using a common set of ASCII based instructions. The SupMCU SCPI implementation is built around an open source SCPI parser library written by Jan Breuer. [2] One of the strengths of the SCPI protocol is that it uses a command tree hierarchy which allows maximum reuse of common commands across devices, minimizing the need to extensively rewrite code to interface with different instruments. Adopting a common electrical architecture and command interface across key flight components will greatly reduce time required for bus integration, testing, and flight software development.

SupMCU modules communicate with a flight computer as a slave device over an I^{2}C bus. I^{2}C's 2-wire common electrical bus and multi-node architecture make it the ideal data bus choice for the SupMCU family of CubeSat devices. The high data overhead and strict timing requirements for I^{2}C message transmission presented some implementation challenges but these were overcome by developing a well defined send/receive sequence, implementing an efficient interrupt driven I^{2}C driver, and taking advantage of the clock stretching capability of I^{2}C slave devices.

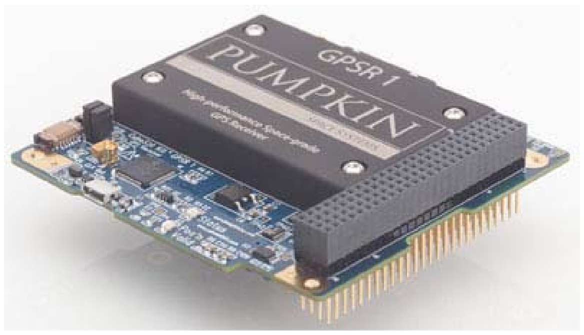

Figure 1: GPSRM 1 Module

The first SupMCU module that has been released is the GPSRM 1, which provides the GPS functionality of NovAtel® OEM615-series GPS receivers in a CubeSat-ready form factor.[3] SCPI over I^{2}C has been extensively tested on this module and has performed without error under conditions simulating the constant demands of a flight C&DH system. Pumpkin currently fields an additional SupMCU-based module (the SIM 1) in its MISC 3 CubeSats, and more are planned.

SUPERVISOR MCU ARCHITECTURE

At the core of a SupMCU module is a PIC24E 16-bit microcontroller running firmware built upon the Salvo ™ real-time operating system. Each module contains a set of hardware & software components for common functions such as power management, I^{2}C bus isolation, programming & debugging support and status indication, as well as unique application-specific features (e.g., control of a GPS module).

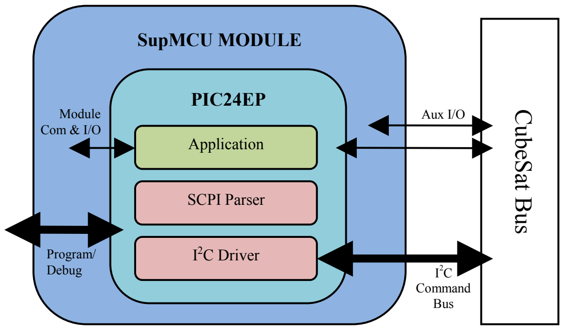

Figure 2: SupMCU simplified

Command and telemetry data is transferred between module and C&DH over the system bus via an I^{2}C connection to the PIC24E. Depending on the module type, a number of auxiliary I/O lines from the microcontroller and application hardware may also be brought out to the main bus. The minimum hardware connection between the SupMCU and the CubeSat requires only power and I^{2}C lines.

Each module's firmware is built from a common set of code that handles RTOS, I^{2}C drivers, SCPI Parsing and implementation of SupMCU common commands and tasks. The generous memory allocation of the chosen SupMCU microcontroller ensures that plenty of program and data space remain available for application-specific functionality. SCPI parser and telemetry functions occupy about 19kB of program memory – less than 8% of the MCU's available 256k. Likewise, less than 2kB, or 6%, of RAM is required for SCPI and telemetry data.

SCPI SYNTAX AND STRUCTURE

SCPI command strings are formed by colon-separated keywords that trace out a command hierarchy. For example, a command to measure a voltage would be written:

MEASure:VOLTage

SCPI commands are case-insensitive but have both a long form and a short form. The short form is typically expressed in capital letters in documentation. The SCPI parser will treat full short form and full long form commands equally but will generate errors for partial or undefined commands. These errors are logged and can be retrieved and handled by the commanding device.

Depending on implementation, certain commands can be changed to queries by appending a '?' to the command string or passed one or more parameters separated by commas. The commands to query and then set the frequency of an instrument might look like: [1]

SYSTem:FREQuency?

SYSTem:FREQuency 8000000

Multiple commands can be sent as a single string separated by semicolon. The previous commands could be shortened to:

SYST:FREQ?;SYST:FREQ 8000000

SCPI's efficient yet human-readable command format makes it suitable for in-flight command & telemetry as well as manual or batch debug and test.

SCPI ON THE SupMCU

Every SupMCU-based module will support a common set of SCPI commands. An outline some of the commands in the SupMCU command hierarchy is shown in Table 1.

Keyword	Parameters	Notes
SUPervisor		*Top level keyword*
:LED	{OFF|ON|FLASh|APPlication}	*Status LED behavior*
:RESet		*Reset MCU*
:I2C		*I2C functions keyword*
:RESet		*Reset I2C driver*
:PASSthrough	{OFF|ON}	*Set I2C isolator*
:CLOCk	{OFF|ON}, <divider_value>	*Configure clock output*
:SELFtest		*Run PIC24E self test*
:TELemetry?	<index_value>	*Request telemetry data*
…	…	…

Table 1: SupMCU Common SCPI Commands

This set of commands covers some of the basic functionality common to all SupMCU modules. It is likely to be expanded and modified as development continues.

In this representation, curly brackets { } enclose keyword parameters and angle brackets < > enclose numeric parameters. As an example, turning on the clock signal output with a divider of 2 would be accomplished with:

SUP:CLOC ON,2

One or more additional top-level keywords are defined for application specific functionality. The GPSRM 1 module defines the keyword *GPS* as well as commands such as: POWer, *:LOG*, and *:RESet*. The command string *GPS:LOG GGA*, for example, will initialize logging of NMEA GGA data from the NovAtel® OEM615 GPS receiver.

The behavior for each SCPI command is defined in callback functions that are called by the SCPI parser when a command match is recognized.

I^{2}C CHALLENGES

Though the I^{2}C-bus has many features that make it desirable as the primary SupMCU command bus, such as a two-wire interface and the ability to address up to 112 slave devices, the high processing overhead as compared to the UART or SPI bus required careful consideration of how to implement data handling on an 8MHz MCU to avoid data loss.[4]

To send a command, an I^{2}C master – typically the C&DH processor – first sends out an address byte, with one bit indicating read or write, over the bus. Each slave device receives this byte and if it matches the device address, the device prepares to receive or transmit additional bytes.[4] Since the dedicated I^{2}C buffer on the PIC24E is only one byte long, it was important that incoming messages were processed efficiently and quickly stored in a RAM ring-buffer for parsing at a later time.

As a first measure to try and accomplish this, the I^{2}C handling routine was made interrupt-driven and optimized to require a minimal amount of MCU operations. This worked at lower I^{2}C clock speeds, but bytes of data were still being missed at I^{2}C speeds over 100 kbit/s or when other interrupt routines or application processes were putting excessive demand on the MCU. We found this result somewhat surprising, as simple and inexpensive I^{2}C devices (e.g., discrete latches) have no problem keeping up with 400kHz I^{2}C clock rates.

Implementing clock stretching on the SupMCU's I^{2}C slave driver solved this problem. Clock stretching is a method by which a slave device can delay the master from sending additional data by holding the I^{2}C clock (SCL) line low during the last bit of an I^{2}C byte until it is ready to receive and process another byte. This allows the SupMCU device to process and store each I^{2}C byte, as well as handle other high priority tasks such as timer interrupts or UART messages, without any chance of missing bytes that arrive over I^{2}C. Figure 3 shows the bit timing of a SCPI message sent over I^{2}C.

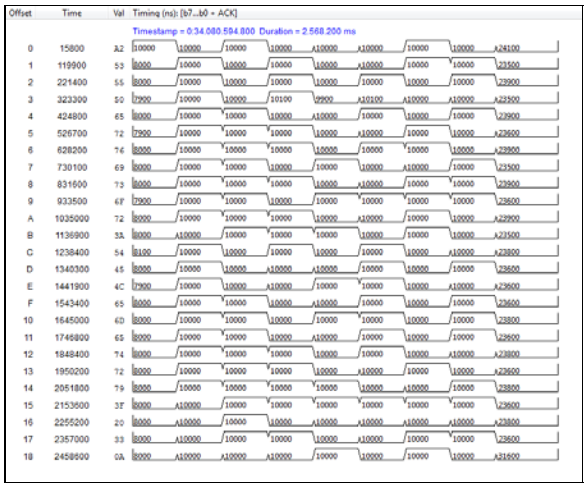
```
Offset   Time     Val  Timing (ns): [b7...b0 + ACK]
                       Timestamp = 0:34.080.594.800  Duration = 2.568.200 ms
0        15800    A2   10000 10000 10000 10000 10000 10000 10000 10000 24100
1        119900   53   8000  10000 10000 10000 10000 10000 10000 10000 23500
2        221400   55   8000  10000 10000 10000 10000 10000 10000 10000 23900
3        323300   50   7900  10000 10000 10100 9900  10100 10000 10000 23500
4        424800   65   8000  10000 10000 10000 10000 10000 10000 10000 23900
5        526700   72   7900  10000 10000 10000 10000 10000 10000 10000 23600
6        628200   76   8000  10000 10000 10000 10000 10000 10000 10000 23900
7        730100   69   8000  10000 10000 10000 10000 10000 10000 10000 23500
8        831600   73   8000  10000 10000 10000 10000 10000 10000 10000 23900
9        933500   6F   7900  10000 10000 10000 10000 10000 10000 10000 23600
A        1035000  72   8000  10000 10000 10000 10000 10000 10000 10000 23900
B        1136900  3A   8000  10000 10000 10000 10000 10000 10000 10000 23500
C        1238400  54   8100  10000 10000 10000 10000 10000 10000 10000 23800
D        1340300  45   8000  10000 10000 10000 10000 10000 10000 10000 23600
E        1441900  4C   7900  10000 10000 10000 10000 10000 10000 10000 23600
F        1543400  65   8000  10000 10000 10000 10000 10000 10000 10000 23600
10       1645000  6D   8000  10000 10000 10000 10000 10000 10000 10000 23800
11       1746800  65   8000  10000 10000 10000 10000 10000 10000 10000 23600
12       1848400  74   8000  10000 10000 10000 10000 10000 10000 10000 23800
13       1950200  72   8000  10000 10000 10000 10000 10000 10000 10000 23600
14       2051800  79   8000  10000 10000 10000 10000 10000 10000 10000 23800
15       2153600  3F   8000  10000 10000 10000 10000 10000 10000 10000 23600
16       2255200  20   8000  10000 10000 10000 10000 10000 10000 10000 23800
17       2357000  33   8000  10000 10000 10000 10000 10000 10000 10000 23600
18       2458600  0A   8000  10000 10000 10000 10000 10000 10000 10000 31600
```

Figure 3: I^{2}C Clock Stretching

Clock stretching is evident on the 9th bit (the acknowledge bit) of each character transmission as the slave device holds the clock low until ready to process the next byte. In this example the clock was typically held for about 24 µs on the last bit instead of the 10 µs that would be required if clock stretching was not enabled. The exact amount of clock stretching required depends on application loads placed on the MCU.

It's important to point out that clock-stretching is an optional feature in the I^{2}C specification that is supported by many, but not all, I^{2}C master devices. *master I^{2}C devices that support clock stretching can communicate with SupMCU-based modules.*

With clock stretching implemented, a SupMCU was shown to be able to receive and parse I^{2}C messages flawlessly at all standard I^{2}C clock rates while running multiple application tasks and sending and receiving debug messages over UART.

TELEMETRY DATA

In order to get telemetry and operational data from a SupMCU module, a telemetry request system has been set up that enables reliable access to information from the SupMCU module.

Similar to the way SCPI commands are divided between commands that are common to SupMCU-based modules and those that are application-specific, the telemetry table is made up of fields common to the SupMCU and ones specific to individual modules. Table 2 shows a sampling of telemetry fields available on the GPSRM 1. This data structure is initialized in RAM at module start up and is periodically updated. Each field can be queried via SCPI by passing the corresponding index parameter.

Table 2: SupMCU Telemetry Table

Index	Name	Data	Data Length (Bytes)	Read Buffer
1	Clock Ticks	<1…>	10	1
2	I2C Messages	<2…>	10	1
3	SCPI Messages Parsed	<3…>	10	1
…	…	…	…	…
10	GPS NMEA GGA	<10…>	88	1
…	…	…	…	…

Fields in the table are updated at different frequencies by dedicated tasks depending on the nature of the data. To avoid conflicts between write and read operations to the telemetry table, a triple ring buffer update system is used. For each data field there are three buffer arrays that get updated in a circular manner. The latest buffer that has been finished updating is tracked and this is where the master is pointed to when it requests data. This ensures that there is always a complete data string available to be read out over I^2C and that there is an exceedingly low chance of an update trying to write while data is being read.

Table 3 shows the way that telemetry data is formatted byte-by-byte in the 'data' field array.

Table 3: Telemetry Data Field

Index	Time Stamp	Telemetry String	CRC Value
<1 Byte>	<4 Bytes>	<Depends on Field>	<1 Byte>

Each data array begins with the (non-zero) index of the field in the telemetry table followed by a timestamp, a fixed length data field, and an 8-bit check value. A cyclic redundancy check is automatically performed and recorded for each telemetry field in order to confirm data integrity.

The I^2C specification does not allow slave devices to initiate transmission of data to a master device. Instead the master device must request data from the slave device byte-by-byte. For this reason, a telemetry request protocol has been established for SupMCU devices that allows the master to request specific data from the SupMCU. The request sequence looks like this:

<u>*C&DH (Master):*</u> Send SCPI message requesting a particular telemetry field – eg. *SUPervisor:TELemetry? 3*

<u>*SupMCU (Slave):*</u> Begin parsing SCPI message

<u>*C&DH:*</u> Begin reading bytes of data at intervals from the slave. The master will read only NULL until requested data is ready to be read.

<u>*SupMCU:*</u> Once request message is parsed and handled, read pointer is set to start of data array.

<u>*C&DH:*</u> Non-zero index byte will be read indicating that data is available. The known length of the requested telemetry message can be read in rapid succession from SupMCU.

<u>*SupMCU:*</u> Reset pointer to NULL when last character of telemetry string is read. Subsequent reads will return NULL until another telemetry request is received.

Table 4 shows what this looks like from the point of view of the master device.

Table 4: Sample Telemetry Read

Operation	Data	Meaning
Write	53 55 50 3a 54 45 4c 3f 20 33	SUP:TEL? 3
Read <1 Byte>	00	0
Read <1 Byte>	00	0
Read <1 Byte>	03	3
Read <9 Bytes>	64 0E 00 00 01 00 00 00 61	3684 (s) 1 (SCPI message parsed) 0x61 (CRCV)

This method requires that the master device know the structure of the telemetry table in order to make requests for the correct field and data string length. A telemetry query function will be implemented so that the master can acquire the structure and metadata of the SupMCU device it is communicating with.

CONCLUSION

Applying SCPI standards to CubeSat modules provides a robust method for command and telemetry that is easy for operators and programmers to learn while being powerful enough to use for in orbit operations. Careful attention to hardware and software architecture ensures that there is maximum interoperability between modules and a minimal amount of effort required to add support for new modules in flight software.

Use of the I^2C bus is the simplest method for electrically connecting an extendable multi-node system. Implementation of clock stretching and telemetry checksums help ensure the reliability of data sent over I^2C, though it adds the requirement that a master device talking to a SupMCU module must support clock stretching.

Further development of the Supervisor MCU family of CubeSat modules will add to their existing functionality. The number of different types of Supervisor MCU modules available is expected to grow beyond the GPSR and SIM that have currently been designed. Adoption of SCPI and common command interface standards across CubeSat modules should significantly decrease time required for test, integration, and software development for future CubeSat missions.

ACKNOWLEDGMENT

A special thanks to Jan Breuer who developed the SCPI parser libraries that were used on the Supervisor MCU firmware and posted them to GitHub with an open licence[2].

REFERENCES

1. SCPI Consortium. “Standard Commands for Programmable Instruments (SCPI).” Version 1999.0.

2. Breuer, Jan. “SCPI parser library.” Internet: https://github.com/j123b567/scpi-parser, [Jun. 13, 2014.]

3. Pumpkin Inc. “GPSRM 1 GPS Receiver Module Datasheet.” Rev C. February 2014.

4. NXP Semiconductors. “I^{2}C-bus specification and user manual.” Rev. 6. 4 April 2014.