Atacama Pathfinder EXperiment

Interface Control Document

APEX-MPI-ICD-0005	
Revision:	1.1
Release:	July 13, 2018
Category:	4
Author:	Dirk Muders

APEX SCPI command and subsystem log message syntax APEX backend data stream format

Heiko Hafok, Dirk Muders, Michael Olberg

Keywords: SCPI, Embedded Systems

Author Signature: Dirk Muders	Date: July 13, 2018
Approved by: L.-Å. Nyman	Signature: L.-Å. Nyman
Institute: APEX	Date: July 13, 2018
Released by: L.-Å. Nyman	Signature: L.-Å. Nyman
Institute: APEX	Date: July 13, 2018

Change Record

Revision	Date	Author	Section/ Page affected	Remarks
1.0	2006-03-29	Heiko Hafok	All	Initial Version
1.1	2018-07-13	Dirk Muders	All	Added detailed explanations. Added subsystem log messages.

Contents

List of Tables

1 Introduction

This document describes the syntax and rules of SCPI (Standard Commands for Programmable Instrumentation) commands and embedded system log messages and the data stream format for backends as they are implemented for the control of APEX instrumentation. For the SCPI commands we basically follow the implementation that was used at the Onsala 20m and SEST telescopes. In addition, we take advantage of the flexibility introduced by the CORBA/component model of the ALMA Common Software (ACS), which is the basis of the APEX control system (APECS).

Physical hardware like receivers, backends, synthesizers etc. is represented as so called CORBA components (CC), i.e. Distributed Objects (DO). The interface of these DOs is defined in IDL (Interface Description Language) file and compiled into stubs and skeletons, which can be accessed by C++, JAVA and Python. There are high level interfaces for e.g. backends, receivers and more low level interfaces describing e.g. the local oscillator, the multiplier, the mixer, etc. (see [1]).

These interfaces are generic and can be applied to any instrument of a given class. They provide a set of building blocks out of which we can construct the logical structure of any instrument such as a receiver. All instruments will have the *same* high-level interface. This makes the setup for the high-level observing software APECS very simple because one just adds a new name but one does not have to worry about adding new features at that level.

The actual detailed structure of e.g. a given receiver or backend system differs, of course, from one to another. This is represented by a naming hierarchy of DOs for the components in the receiver. This way we can find all relevant DOs belonging to one instrument and we determine its structure programmatically by looking at the naming hierarchy. This hierarchy is stored in the ACS Configuration Data Base (CDB) which consists of directories defining the name space in which the generic DOs are started and XML files describing the IDLs of the individual sub-devices like local oscillators, backend bands etc.

One can see that we need only a few IDLs to describe a complex device like a heterodyne frontend (HFE): apexHFE, apexHFE_Mixer, apexHFE_LO, apexHFE_Multi, etc. or a spectral backend (SBE): apexSBE, apexSBE_Band (see [1]). We reuse the same IDL and even the same DO C++ library many times. A new device is merely a new CDB entry. We do not even have to compile any new code unless one needs to add something that is not covered by the existing IDLs. The interfaces have been in use at APEX for almost 15 years without major changes. Only a few optional features were added.

Except for the telescope and wobbler DOs which run on a VxWorks real-time system, the direct hardware control is not implemented as a DO in ACS but via embedded systems. The DOs serve as the communication layer between APECS and the embedded hardware control system. For communication between the CORBA/ACS component and the instruments a socket based ACS "Device IO" (DevIO) class using externally multiplexed UDP communication (DevIOUDPSock) was developed. The connectionless UDP protocol was chosen because of bad experiences with TCP connections getting stuck in previous projects. In addition, it facilitates debugging via a simple client program while the system is running.

For each DO a control host and port and an optional data host and port (for backends) are defined in the CDB to send and receive commands from the instruments and to receive binary data from the backends.

We adopt the hierarchical structure of SCPI, where keywords can be re-used. The command syntax is created by the DevIOUDPSock implementation in APECS from the name space defined in the CDB.

2 SCPI command syntax and rules

The SCPI naming hierarchy uses the colon as the separator between the command elements. The device names are derived from the hierarchical structure stored in the CDB. The device properties and methods as defined in the IDL file are appended to the device name using again a colon as separator:

```
[APEX:]<device name>:<property/method name>
```

In interactive mode unequivocal abbreviations are optionally allowed. The APEX DevIOUDPSock software always uses the fully qualified names with device names in capitals and property and method name in lower camel case capitalization like in the IDLs. The replies may, however, use arbitrary capitalization.

The APEX UDP SCPI communication requires an acknowledgement from the embedded system to ensure that the commands were actually received by the device. The answer must be sent to the originating host and port to ensure that the multiplexing machanism works.

2.1 Transactions

The embedded system must reflect the SCPI command, possibly add the requested value(s) and add a TAI ISO8601 time stamp without time zone suffix. In case of returning a value, the time stamp must be the time of when this value was actually sampled, i.e. if the system is buffering those numbers internally, it may be a past time. Those times are used by the APEX monitoring system.

There are three types of APEX SCPI commands to get/set values or invoke methods:

1. Getting the value of a parameter: `[APEX:]<device name>:<property name>?`
   Answer: `[APEX:]<device name>:<property name> value <ISO8601 time stamp>`
2. Setting the value of a parameter: `[APEX:]<device name>:<property name> value`
   Answer: `[APEX:]<device name>:<property name> value <ISO8601 time stamp>`
3. Invoking a method: `[APEX:]<device name>:<method name>`
   Answer: `[APEX:]<device name>:<method name> <ISO8601 time stamp>`

In case of errors, the embedded system must send the string "ERROR" after the SCPI command. It may add specific information about what went wrong:

```
[APEX:]<device name>:<property/method name> ERROR <error type> <TAI ISO 8601 time
stamp>
```

where <error type> is a string without blanks.

If parts of an interface cannot be implemented by a given device, the SCPI answer should be the string "NOT_AVAILABLE" after the SCPI command:

```
[APEX:]<device name>:<property/method name> NOT_AVAILABLE <TAI ISO 8601 time stamp>
```

The DO will replace this with a default value for the given type.

The embedded system may reply only when the requested action is completed. For low-level settable properties, this means that a new value is accepted. The actual value may already have reached the commanded one, but it does not have to. It is just important to reply within the given timeout (see below). Client software needs to check the actual value to be sure that it has reached the commanded one (within some accuracy limits for numerical values).

For high-level properties that are used for a configuration command, the reply just acknowledges that the new value has been received and stored for later.

For methods, it means that the complete action has finished. If this takes a long time, the embedded system must send the reply only after completion. This must, however, not block any other (monitoring and/or control) SCPI commands in between. The parser programs thus typically need to be multi-threaded. This use case also illustrates the general multi-threaded communication from the CORBA component which can lead to initiating further SCPI transactions before previous ones have been completed. In the DO these transactions are generated from individual C++ objects which have individual host/port coordinates per transaction. The embedded system must keep track of these and send the replies to the correct origin.

The CORBA components implement an independent monitoring loop that samples selected actual values at given rates (typically once per minute) and provides them via a notification channel to monitoring GUIs and to the central data archiver that writes these time stamped monitoring data into a database.

2.2 Property Types and Names

Properties may be of the ACS types long, double, longSeq, doubleSeq, string or enum. Sequences are to be sent with a single blank as separating character. Enums must be sent as the ASCII texts defined in the IDL. There must not be any trailing whitespace after the time stamp.

Whether a parameter can be set or only read is defined by the IDL type. We use a scheme of actual and commanded parameters to be able to see possible configurations differences. The commanded parameters are settable while the actual ones are read-only. Commanded parameter names begin with "cmd", followed by the name of the actual parameter. A typical example is a property pair like `skyFrequency/cmdSkyFrequency`.

2.3 Timeouts

In the IDL interface, we distinguish between synchronous and asynchronous methods. Synchronous methods and synchronous property acesses have a timeout of 4 seconds. Timeouts for asynchronous methods and property accesses can be set by the client software.

Sometimes changing a hardware status may take a long time and also depend on several commanded parameters (e.g. tuning a receiver or configuring a backend). We therefore distinguish between so called low- and high-level properties. Setting low-level properties shall cause an immediate change of the corresponding actual hardware parameter while setting a high-level property shall merely store the value for later use. In that case, there are methods like "tune" for receivers or "configure" for backends which take all commanded high-level values and re-configure the system to that new configuration at once. Note that the commanded high-level values are sent without any specific order.

2.4 Examples of SCPI communication

The names of the devices for a complex 460 GHz receiver with two local oscillator chains could look like this:

```
APEX:HET460
APEX:HET460:CALUNIT
APEX:HET460:MIXER1
APEX:HET460:MIXER1:COLDAMP
APEX:HET460:MIXER2
APEX:HET460:MIXER2:COLDAMP
APEX:HET460:LO1
APEX:HET460:LO1:PLL
APEX:HET460:LO1:MULTI1
APEX:HET460:LO1:MULTI2
APEX:HET460:LO2
APEX:HET460:LO2:PLL
APEX:HET460:LO2:MULTI1
APEX:HET460:LO2:MULTI2
```

Examples of typical SCPI communication are:

Request: `APEX:HET460:LO2:MULTI1:backShort2?`
Answer: `APEX:HET460:LO2:MULTI1:backShort2 2.341 2005-11-05T10:19:38`

Request: `APEX:HET460:LO1:MULTI2:backShort1?`
Answer: `APEX:HET460:LO1:MULTI2:BACKSHORT1 ERROR HARDWARE-FAILURE 2005-11-05T10:19:38`

Request: `APEX:HET460:cmdSkyFrequency 461.018870922`
Answer: APEX:HET460:cmdSkyFrequency 461.018870922 2005-11-05T10:19:38

Request: `APEX:HET460:cmdSideBand USB`
Answer: APEX:HET460:cmdSideBand USB 2005-11-05T10:19:39

Request: `APEX:HET460:tune`
Answer: APEX:HET460:tune 2005-11-05T10:20:51

2.5 Observing and Monitoring Interfaces

In order to allow for the complete remote control of a device by APECS, certain IDLs have to be implemented by the embedded system. The embedded system must react properly to SCPI requests at all times so that the monitoring loops do not run into timeouts which can slow down the CORBA containers if too many of them occur. In addition to the mandatory IDLs, optional IDLs may be implemented to benefit from the automatic monitoring into the APECS monitor database and to access lower level instrument configurations via scripts or GUIs. Table 1 shows the mandatory and optional IDLs for frontends, IF processors and backends.

Device Type	Mandatory Observing IDLs	Optional Monitor and Control IDLs
Heterodyne Frontend	apexHFE, apexHFE_LO, apexCalUnit	apexHFE_Multi, apexHFE_Mixer, apexHFE_ColdAmp apexHFE_PLL, apexHFE_Gunn, apexFE_Derot
Continuum Frontend	apexCFE	apexCFE_Pol, apexCFE_Amp
IF Processor	apexIF, apexIF_Chain	apexIF_Modules
Spectral Backend	apexSBE, apexSBE_Band	–
Continuum Backend	apexCBE	–

Table 1: Mandatory observing and optional monitor and control IDLs.

3 Subsystem log messages

Starting with APECS 3.3 subsystems can submit log messages to APECS to report warnings or errors independent of SCPI transactions. APECS accepts strings with a given format and translates them to ACS based APECS logs. The string must be sent to the host "apecslog" on UDP port 12157. The format of the string is

```
Message="This is a test message." | Priority="Info" | Audience="Observer" |
Source="udp-telnet" | TimeStamp="2017-02-22T11:49:49"
```

Priority can be "Info", "Warning", "Error", "Critical" or "Emergency", Audience can be "Operator", "Observer" or "Operator, Observer", the TimeStamp is in ISO8601 format (TAI). Message and Source are free text except for the `"|"` character, which is the separator between the different parts of the log message. One can omit everything except "Message". In that case the UDP logger assumes defaults for the other parameters (Priority: Info, Audience: Observer, Source: N/A, TimeStamp: current time).

4 Binary backend data format

For APEX we have defined a simple binary backend data format consisting of a header and the actual data. The header allows to distinguish the encoding standard (IEEE or EEEI) since these character are sent as a string. The header contains one time stamp referring to the mid-time of the first package. If data is sent with a blocking factor larger than 1, then the subsequent time stamps are assumed to be equidistantly spaced in steps of the integration time. If this cannot be guaranteed because of varying blanking time (e.g. for wobbler or frequency switched observations), then the data should not be blocked (the latter is the default at APEX). Data for switched observations is sent per phase since the actual integration times may vary. Table 2 summarizes the binary format definition.

Item	Description	Item length
IEEE	Numerical encoding standard	4 bytes ASCII
I F + 3 blanks	Backend data format (longInt or float)	4 bytes ASCII
< length:longInt>	Length of data package	4 bytes Binary
<BEName:char[8]>	Backend identifier	8 bytes ASCII
<timeStamp:char[28]>	Mid time ISO 8601 timestamp: YYYY-mm-ddTHH:MM:SS.ssss[TAI||GPS||UTC] + 1 blank	28 bytes ASCII
<integrationTime:longInt>	Time per integration (micro seconds)	4 bytes Binary
<phaseNumber:longInt>	Phase number (1-based)	4 bytes Binary
<numBESections:longInt>	Number of backend sections	4 bytes Binary
<blocking:longInt>	Blocking factor (integrations per timestamp)	4 bytes Binary
<BESec1:longInt> <numChannels:longInt> <int1 BESec1 ch1:longInt I float> ... <int1 BESec1 chN:longInt I float> ..... <BESecN:longInt> <numChannels:longInt> <int1 BESecN ch1:longInt I float> ... <int1 BESecN chN:longInt I float> ... ... <BESec1:longInt> <numChannels:longInt> <intN BESec1 ch1:longInt I float> ... <intN BESec1 chN:longInt I float> ..... <BESecN:longInt> <numChannels:longInt> <intN BESecN ch1:longInt I float> ... <intN BESecN chN:longInt I float>	Backend section number (1-based) Number of channels for this backend section Data	(4 bytes + 4 bytes + 4 bytes * numChannels) * numBESections * blocking Binary

Table 2: APEX binary backend stream format.

References

[1] Muders, D., 2018, APEX Instruments Generic CORBA IDL Interfaces, APEX Report APEX-MPI-ICD-0004, Rev. 3.4