Atacama Pathfinder EXperiment

Interface Control Document

APEX-MPI-ICD-0005
Revision: 1.0
Release: March 29, 2006
Category: 4
Author: Heiko Hafok

APEX SCPI socket command syntax and backend data stream format

Heiko Hafok, Dirk Muders, Michael Olberg

Keywords: SCPI

Author Signature: Heiko Hafok	Date: March 29, 2006
Approved by: R. Güsten Institute: MPI für Radioastronomie	Signature: R. Güsten Date: April 1, 2006
Released by: R. Güsten Institute: MPI für Radioastronomie	Signature: R. Güsten Date: April 1, 2006

1 Introduction

This document describes the syntax and rules of SCPI (Standard Commands for Programmable Instrumentation) commands as they are implemented for the control of APEX instrumentation. 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 software.

Physical hardware like receivers, backends, synthesizers etc. is represented as so called CORBA components, i.e. Distributed Objects (DOs). 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 APEX-MPI-IFD-0004).

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 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 very simple because ones just adds a new name but one does not have to worry about adding new features at that level.

The actual detailed setup of e.g. a given receiver 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 can see its structure 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 front-end HFE : apexHFE, apexHFE Mixer, apexHFE ColdAmp, apexHFE_LO, apexHFE_Gunn, apexHFE PLL, apexHFE Multi and apexHFE HCal. We reuse the same IDL and even the same DO library many times. A new device is merely a new CDB entry. We don't even have to compile any new code unless one adds something that is not covered by the existing IDL's.

Except for the Telescope DOs themselves and the wobbler DO which run on a VxWorks embedded systems the direct hardware control is not implemented as a DO in ACS but via embedded systems. The DOs are serving as a communication layer between the APEX control system and the embedded hardware control system. For communication between the CORBA/ACS control system and the instruments a socket based ACS "Device IO" (DevIO) class using externally multiplexed UDP connection (DevIOUDPSock) was developed. The connectionless UDP protocol was chosen because of very bad experiences with TCP connections getting stuck in previous projects.

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 inside the control software from the name space defined in the CDB. The standard APEX SCPI parser used for many embedded systems is based on the original software used at Onsala and at the SEST and uses `lex`(1) and `yacc`(1) or `bison`.

2 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 allowed. The APEX DevIOUDPSock software always uses the fully qualified names. 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.

The embedded system must reflect the SCPI command, possibly add the requested value and add an ISO8601 time stamp. 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> <ISO 8601 time
stamp>
```

where <error type> is a string without blanks.

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. A typical example is a property pair like `skyFrequency/cmdSkyFrequency`.

The embedded system may reply only when the requested action is completed. For settable properties, this means that a new value is accepted, but the actual value does not have to be the same yet. For methods, it means that the complete action is finished. If this takes a long time, the embedded system must not block other (monitoring) SCPI commands in between. The parser programs thus typically need to be multi-threaded.

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.

3 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:GUNN
APEX:HET460:LO1:PLL
APEX:HET460:LO1:MULTI1
APEX:HET460:LO1:MULTI2
APEX:HET460:LO2
APEX:HET460:LO2:GUNN
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

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. The following table 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.0000[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