
| | |
|---|---|
| **Client:** | NRF (National Research Foundation) |
| **Project:** | Telescope Project |
| **Type:** | Subsystem Interface Specification |

# Guidelines for Communication with Devices

| | |
|---|---|
| **Document number:** | NRF-KAT7-6.0-IFCE/002 |
| **Revision:** | 4 |
| **Classification:** | Unrestricted |
| **Author:** | S. Cross, R. Crida, T. Bennett and M. Welz |
| **Date:** | 2009/04/29 |

# Document Approval

| | Name | Designation | Affiliation | Date | Signature |
|---|---|---|---|---|---|
| Submitted by | S. Cross | CSS Developer | SKA SA | | |
| Accepted by | T. Kusel | System Engineer | SKA SA | | |
| Approved by | W. Esterhuyse | Project Manager | SKA SA | | |

# Document History

| Revision | Date of Issue | ECN Number | Comments |
|---|---|---|---|
| A | 2008/06/27 | N/A | Initial version. |
| B | 2008/06/30 | N/A | Added REQUEST Restart. |
| C | 2008/06/30 | N/A | Updates after review of Rev B. |
| D | 2008/07/17 | N/A | Added details for replies plus ref to logging memo. |
| E | 2008/07/21 | N/A | Added static IP configuration and incorporated logging memo. |
| 1 | 2008/07/31 | N/A | Changes described in tags/RevE/NRF-KAT7-6.0-IFCE-002-RevE-COAR.xls including describing simulators. |
| 1A | 2008/08/01 | N/A | Fixed mistakes with Sample Config in diagrams and removed ref to INFORM Config. Added REQUEST Help and INFORM Disconnect. |
| 1B | 2008/10/01 | N/A | Changes and improvements prompted by initial implementations of libraries for the protocol. |
| 2 | 2008/10/10 | N/A | Provide option of using PTP for time synchronization. Describe how Proxies and Devices should handle malformed messages. |
| 2A | 2008/10/23 | N/A | Allow any amount of whitespace between arguments. Introduce underscore and at sign escapes. |
| 3 | 2008/10/24 | N/A | Trim logging options down to those likely to be used. Remove confusing "mandatory" lines from logging table. |
| 3A | 2008/11/20 | N/A | Added document family figure for context. |
| 3B | 2009/02/06 | N/A | Changes resulting from CSS Design Review 1. |
| 4 | 2009/02/06 | KAT-7-ECP-003 | Changes from multi-client ECP. |

# Document Software

| | Package | Version | Filename |
|---|---|---|---|
| Stylesheet | katdoc | 1.1.1 | katdoc.sty |
| Word processor | LaTeX | 3.141592-1.40.3 (Web2C 7.5.6) | NRF-KAT7-6.0-IFCE-002.tex |
| Diagrams | Inkscape | 0.46 | images/*.svg |
| Diagrams | Inkscape | 0.46 | images/*.pdf |
| Diagrams | epstopdf | 2.9.5gw | images/ska_logo.pdf |

# Company Details

| | |
|---|---|
| Name | Karoo Array Telescope Office |
| Physical/Postal Address | Units 12 - 14<br>Lonsdale Office Park<br>Lonsdale Avenue<br>Pinelands<br>7405 |
| Tel. | +27 21 531 7282 |
| Fax | +27 21 531 9761 |
| Website | http://www.ska.ac.za |

# Contents

# List of Figures



# List of Tables



# List of Abbreviations

| | |
|---|---|
| API | Application Programming Interface |
| BNF | Backus-Naur Form [1] |
| DHCP | Dynamic Host Configuration Protocol |
| KAT | Karoo Array Telescope |
| KATCP | KAT Communication Protocol |
| ICD | Interface Control Document |
| IP | Internet Protocol |
| LRU | Line Replaceable Unit |
| NTP | Network Time Protocol |
| RFE | Radio Front End |
| SKA | Square Kilometer Array |
| TCP/IP | Transmission Control Protocol/Internet Protocol |
| UTP | Unshielded Twisted Pair |

# 1 Introduction

The purpose of this document is to describe a communication protocol between hardware devices and the software that controls them. It has been produced by SKA (Square Kilometer Array) South Africa as part of the KAT (Karoo Array Telescope) project and the protocol design has been driven by the KAT project requirements. The protocol has been dubbed KATCP (the KAT Communication Protocol). Note that additional requirements relating to devices being implemented for the KAT project are captured in Appendix A.

Broadly speaking, KATCP consists of newline-separated text messages sent asynchronously over a TCP/IP stream. There are three categories of messages: requests, replies and informs. Request messages expect some sort of acknowledgement. Reply messages acknowledge requests. Inform messages require no acknowledgement. Inform messages are of two types: those sent synchronously as part of a reply and those sent asynchronously.

A summary of the standard requests is provided in Table 1. Table 2 provides a summary of the standard asynchronous informs.

This document is divided into sections describing the modules that make up the protocol. Hopefully this makes the guidelines easier to read and implement. Modules are divided into two types: *optional* (implementations may conform to these at their discretion) and *required* (implementations must conform to these). The multi-client and single-client modules are both optional, but all devices must conform to one of these two.

| Request | Required? | Module | Description |
|---|---|---|---|
| client-list | optional | Multi-client | List the clients connected *[informs]*. |
| configure | optional | Device Configuration | Configure properties on a device. |
| halt | required | Core Messages | Halt a device server. |
| help | required | Core Messages | Return help on the requests supported by a device *[informs]*. |
| log-level | required | Logging | Query or set the logging level. |
| mode | optional | State and Mode | Query or change the mode. |
| restart | required | Core Messages | Restart a device server. |
| sensor-list | required | Sensors | List the sensors a device supplies *[informs]*. |
| sensor-sampling | required | Sensors | Configure reporting of device sensor values. |
| sensor-value | required | Sensors | Request sensor values *[informs]*. |
| watchdog | required | Core Messages | Ping the device. |

Table 1: List of requests covered by this document. The requests that send inform messages as part of their reply are marked with *[informs]* in their description. More detail is provided in the module which covers the request.

| Inform | Required? | Module | Description |
|---|---|---|---|
| build-state | required | Core Messages | Report build information for the device software. |
| client-connected | optional | Multi-client | Inform other clients when a client connects. |
| disconnect | required | Core Messages | Inform a client that its about to be disconnected. |
| log | required | Logging | Report logging information and errors. |
| sensor-status | required | Sensors | Report sensor values. |
| version | required | Core Messages | Report version information for the device interface. |

Table 2: List of informs covered by this document. More detail is provided in the module which covers the inform.

# 2 Messaging Protocol [Required]

The preferred interface to devices is a text-based protocol resembling a command-line interface. It should be accessible over a TCP/IP connection. The purpose of the protocol is to enable control and monitoring of a device. It is not intended for high-volume data transport. The protocol layers are described in Table 3.

| Layer | |
|---|---|
| Application | katcp |
| Transport | TCP/IP |
| Link | 10Mb, 100Mb or 1Gb Ethernet |
| Physical | UTP |

Table 3: Table describing the proposed protocol layers.

Communication consists of a number of messages, each message consisting of a line of text. The protocol supports requests, replies and inform messages. The protocol is symmetrical in that either party may send any type of message. Requests are indicated by "?", replies by "!" and informs by "#". A request should be acknowledged by a reply for synchronous communication. An inform can be sent asynchronously and does not require a reply. Replies should not be sent except in response to a request.

Although the protocol is symmetric, it is envisaged that requests will usually be sent *to* the device and that the device will be the *source* of inform messages. This is true for all messages described in this document.

A reply is necessary for every request, however the nature of the reply may change depending on the request. The reply message should have the same name as the request message, even when the request name does not correspond to a request handled by the device.

The first parameter of a reply message should always be a return code. A return code of `ok` indicates successful processing of the request, while anything else indicates failure. The recommended failure strings are `invalid` (for malformed requests) and `fail` (for valid requests which could not be processed) but devices may return other failure strings. On success, further parameters are specific to the type of request made while in the case of failure a second parameter should describe the failure in more detail and in human-readable form. The standard return codes are listed in Table 4.

| Return Code | Description |
|---|---|
| `ok` | Request successfully processed. Further arguments are request-specific. |
| `invalid` | Request malformed. Second argument is a human-readable description of the error. |
| `fail` | Valid request that could not be processed. Second argument is a human-readable description of the error. |

Table 4: List of standard return codes. Only `ok` indicates success. The codes `invalid`, `fail` and any unlisted return code indicate a failed request.

When a device receives an unparsable message or an unexpected reply, it should respond by sending a `#log error` message back to the client explaining the error. A client which receives a badly formed message or an unexpected reply, should *not* send anything back to the device but should rather pass the error on to an external logging mechanism or other third party. This intentional asymmetry is to ensure that message corruption does

not result in a flood of message between the device and the client. Device servers may ignore unexpected inform messages.

Where message parameters are described as "human-readable" the contents of the parameter should be restricted to plain ASCII text (printable ASCII plus the escape characters for horizontal tab, line feed and carriage return).

| Request and Reply Examples |
|---|
| ?set-rate 5.1<br>!set-rate ok |
| ?set-unknown-parameter 6.1<br>!set-unknown-parameter invalid Unknown_request. |
| ?set-rate 4.1<br>!set-rate fail Hardware_did_not_respond. |

Table 5: Example request and reply messages.

The message grammar is described next.

## 2.1 Message grammar

The message grammar is described in extended BNF [1] where:

- Optional items are enclosed in square brackets.
- Items repeating 0 or more times are suffixed with a *.
- Items repeating 1 or more times are suffixed with a +.
- Set difference is indicated by /. For example $\{1,2,3\}/\{2,3,4\} = \{1\}$.
- Alternative choices in a production are separated by the 'l' symbol.

```
   <message> ::= <type> <name> <arguments> <eol>
      <type> ::= "?" | "!" | "#"
      <name> ::= alpha (alpha | digit | "-")*
<whitespace> ::= (space | tab) [<whitespace>]
       <eol> ::= newline | carriage-return
 <arguments> ::= (<whitespace> <argument> <arguments>) | <whitespace> | ""
  <argument> ::= (<plain> | <escape>)+
    <escape> ::= "\" <escapecode>
<escapecode> ::= "\" | "_" | zero | "n" | "r" | "e" | "t" | "@"
   <special> ::= backslash | space | null | newline | carriage-return | escape | tab
     <plain> ::= character / <special>
```

Note that unlike in some earlier versions of the protocol, tabs are a valid form of whitespace and any amount of whitespace can occur between arguments or before the end of the message. The characters listed in the `<special>` production may not occur as raw characters in arguments but can be represented by a backslash followed by the corresponding character in the `<escapecode>` production above. The escape character pair `\@` unescapes to the empty string and is used to represent empty arguments. For example, the message `#foo \@`

represents an inform message with one parameter whose value is the empty string. Sending the `\@` escape is discouraged except in the case of sending an empty argument but parsers should handle it wherever it appears.

Lines that contain only whitespace should be ignored by devices and device clients even though they do not constitue valid messages.

All string constants used in messages should be in lowercase including message names, log levels (Section 5), sensor types (Section 6), and the `ok`, `fail` and `invalid` return codes.

## 2.2 Informs Associated with Requests

Where a request returns a list of values the standard mechanism for dealing with this is to return a list of inform messages which precede the reply and for the reply itself to include just a success code and the number of items returned. For example,

```
?sensor-list
#sensor-list drive.enable-azim Azimuth\_drive\_enable\_signal\_status \@ boolean
#sensor-list drive.enable-elev Elevation\_drive\_enable\_signal\_status \@ boolean
#sensor-list drive.dc-voltage-elev Drive\_bus\_voltage V float 0.0 900.0
!sensor-list ok 3
```

It is mandatory that informs which form part of a response to a request have the same message name as the request (and reply) and that no other informs should share a name with a request.

The requests defined in this document that use this technique are: `?help` (Section 4), `?sensor-list` and `?sensor-value` (Section 6), and `?client-list` (Section 7).

# 3 Datatypes [Required]

KATCP message arguments are sent as strings. It is often necessary to send other datatypes to and from a device. In order to do so, this data must be encoded into strings before being sent and decoded when it is received. This section defines formats for a number of common datatypes. Table 6 describes how each of the specified types should be formatted. After formatting, parameters will be escaped (as described in the message grammar) when the message string to be sent is constructed.

| Type | Format | Example |
|---|---|---|
| integer | as formatted by `printf("%d", i)` in C99 | `123`, `-546` |
| float | as read by C99's `strtof(s)`, `strtod(s)` and `strtold(s)` functions, but without the optional leading spaces and only in decimal format | `-1.234e-05`, `1.7` |
| boolean | `True` should be formatted as `1` and `False` as `0`. | `1`, `0` |
| lru | one of the values `nominal` or `error` | `nominal`, `error` |
| timestamp | XXXX.YYYY where XXXX is an integer representing milliseconds since the Unix epoch and the optional .YYYY is the remaining fraction of a millisecond. | `1222180721660213`, `1222195721660237.723`, `1222195721660237.85` |
| discrete | one of a defined set of values specific to the type | `initialise`, `operate`, `maintain` |
| string | character bytes (with no implied character encoding) | `abc`, `foo` |

Table 6: Formatting for parameter types.

## Notes

- The `lru` (line replaceable unit) datatype is intended to represent part of a device which may be either operational (`nominal`) or non-operational (`error`).
- Although KATCP supports sending arbitrarily large integers and floats, those implementing devices should note (in any interface documentation) instances where arguments that cannot be represented as 32 bit integers or floats are expected.
- Although timestamps may have arbitrary accuracy, devices are free to store only as much of a timestamp as is relevant to them.
- Representing timestamps in milliseconds will require integers larger than 32 bits, but even those devices that represent timestamps internally in seconds (and merely format them to milliseconds when constructing messages) should be aware that the lifespan of devices may extend past 2038 when the number of seconds since the Unix epoch will exceed the maximum integer that can be represented with 32 bits.

- In older versions of KATCP, timestamps were not allowed to contain the fractional number of milliseconds part.
- If the character data contained in a string type argument should be interpreted with a specific encoding, those implementing the device should note this in any interface documentation for the device. Devices are permitted to return non-character data in string sensors, but this is not encouraged.

# 4 Core Messages [Required]

The requests and informs detailed in this section deal with connecting to a device, halting it or restarting it and querying it for some basic information about itself. KATCP devices are required to implement all of the messages in this section.

## 4.1 Requests

If a request below does not have a corresponding reply message, then the reply message has just one argument (`ok`) if the request was successful and just two arguments (a failure code and an error message) if the request was unsuccessful.

**REQUEST halt** `?halt` should trigger a software halt. It is expected to close the connection and put the software and hardware into a state where it is safe to power down. The reply message should be sent just before the halt occurs.

**REQUEST help** `?help [name]`

**name** is an optional request name

Before sending a reply, the help request will send a number of `#help` inform messages. If no name parameter is sent the help request will return one inform message for each request available on the device. If a name parameter is specified, only an inform message for that request will be sent. On success the first reply parameter after the status code will contain the number of help inform messages generated by this request. If the name parameter does not correspond to a request on the device, a reply with a failure code and message should be sent.

**INFORM help** `#help name description`

**name** the name of a request

**description** a human-readable description of what the request does, its parameters and return values.

Although the description is not intended to be machine readable, the preferred convention for describing the parameters and return values is to use a syntax like that seen on the right-hand side of a BNF production (as commonly seen in the usage strings of UNIX command-line utilities and the synopsis sections of man pages). Brackets ([]) surround optional arguments, vertical bars (|) separate choices, and ellipses (...) can be repeated.

**REPLY help** `!help ok numCommands`

**numCommands** number of inform messages generated in response to the request.

**REQUEST restart** `?restart` should trigger a software reset. It is expected to close the connection, reload the software and begin execution again, preferably without changing the hardware configuration (if possible). It would end with the device being ready to accept new connections again. The reply should be sent before the connection to the current client is closed.

**REQUEST watchdog** `?watchdog` may be sent by the client occasionally to check that the connection to the device is still active. The device should respond with a success reply if it receives the watchdog request.

## 4.2 Asynchronous Informs

The inform messages listed here are not sent in response to a request, but rather in response to events on the device. The events that should trigger the sending of each type of inform are described along with the description of the inform message parameters below.

**INFORM build-state** `#build-state name-major.minor[(a|b|RC)number]`

**name** is the name of the software running on the device

**major.minor[(a|b|RC)number]** is the version number of the device software

A `#build-state` inform should be sent to a client on connection and should define the build version of the device software. E.g. `#build-state antennasimulator-3.5a3`.

**INFORM disconnect** `#disconnect message`

**message** is a message describing the reason for disconnection

Sent to the client by the device shortly before the client is disconnected. In the case where a client is being disconnected because a new client has connected (see Section 8 on single client devices), the message should include the IP number and port of the new client for tracking purposes. E.g. `#disconnect New\_client\_connected\_from\_192.168.1.100:24500`.

**INFORM version** `#version api-major.minor`

**api** is the name of the API implemented by the device, e.g. antenna, dbe

**major and minor** describe a version number for the interface which should be defined in the ICD.

A `#version` inform should be sent to a client on connection and should define the version of the device API. This allows the client to perform a basic sanity check that it and the device are using compatible versions of the API. The minor version number should be incremented when the API changes in a backwards compatible way (including the adding new sensors and requests or altering existing requests to accept wider ranges of options). The major number should be incremented if the API changes in a non-backwards compatible way (including removing commands or any change that would make use of a request from the previous API major version fail). E.g. `#version antenna-1.0`.

# 5 Logging [Required]

Devices should whenever possible send log messages to connected clients using the `#log inform`. Which log messages should be reported is controlled through the log-level request. Devices may also log messages to some other logging mechanism in order to assist in troubleshooting when no client is connected. This secondary logging mechanism might be to a local standard directory (e.g. `/var/log` on Linux), or to a network logging service, or to some configurable language-specific logging library (for example, *log4j* in Java or the standard *logging* module in Python).

Note that even for multi-client devices, it is envisioned that the logging level will be a setting global to the device. If one client sets the logging level, all clients will receive log informs as dictated by the new logging level. Unlike sensor sampling (see Section 6), the logging level is expected to persist when clients disconnect and then later reconnect.

## 5.1 Standard Logging Levels

After an investigation into the logging levels used for standard logging implementations (*log4j*, *logging for Python* and *syslog*) the set of logging levels shown in Table 7 was chosen for the KATCP protocol. Definitions and expected content for each of the logging levels are included in the table as a guideline for developers to decide on what information should be logged at each level. When logging has been set to a particular level, all higher levels will also be reported. For example when the logging level has been set to `INFO`, the logging levels `INFO`, `WARN`, `ERROR` and `FATAL` will all need to be reported. The higher the logging level that has been set, the less information should be reported by the device.

## 5.2 Requests

**REQUEST log-level** `?log-level [level]`

**level** is one of (off > fatal > error > warn > info > debug > trace > all) and all levels greater than or equal to the specified level should be reported. See Table 7 for a full description of the levels.

If the level parameter is omitted, then the log level is left unchanged but the current level is still returned in the reply.

**REPLY log-level** `!log-level ok level` The level returned is the new log level (or the current log level if no level was specified).

## 5.3 Asynchronous Informs

**INFORM log** `#log level timestamp_ms name message`

**level** is the log level of the message

**timestamp_ms** is a count in milliseconds since the Unix epoch (formatted as described in Section 3).

**name** is the name of the logger using a dotted notation. This allows a virtual hierarchy of loggers to be represented.

**message** is the actual message string. Conventions could be used to identify file names and line numbers etc as appropriate.

| | |
|---|---|
| **Log Level** | `OFF` |
| **Definition** | OFF is the highest possible logging level and is intended to turn logging off. |
| **Expected Content** | No information. Devices should never log messages directly to the OFF logging level. |
| **Log Level** | `FATAL` |
| **Definition** | The device has failed. There is no workaround. Recovery is not possible. |
| **Expected Content** | The logged message should capture as much system state information as possible in order to assist with debugging the problem. Logging information at this level should not directly impact the performance of the device. |
| **Log Level** | `ERROR` |
| **Definition** | An error has occurred. A function or operation did not complete successfully. A workaround may be possible. The device can continue, potentially with degraded functionality. Logging information at this level should not directly impact the performance of the device. |
| **Expected Content** | The error message should capture detailed information relating to the event that has occurred. |
| **Log Level** | `WARN` |
| **Definition** | A condition was detected which may lead to functional degradation (e.g. an anomaly threshold has been crossed), but the device is still fully functional. Logging information at this level should not directly impact the performance of the device. |
| **Expected Content** | The warning message should capture the information relating to what functional degradation may occur and list thresholds that have been exceeded. |
| **Log Level** | `INFO` |
| **Definition** | This level of logging should give information about workflow at a coarse-grained level. Information at this level may be considered useful for tracking process flow. Logging information at this level should not directly impact the performance of the device. |
| **Expected Content** | The information message should capture information relating to the operation that has completed. |
| **Log Level** | `DEBUG` |
| **Definition** | Verbose output used for detailed analysis and debugging of a device. Logging information at this level may impact the performance of the device. |
| **Expected Content** | This level of logging should show workflow at a fine-grained level. Information relating to parameters, data values and device states should be reported. |
| **Log Level** | `TRACE` |
| **Definition** | Extremely verbose output for detailed analysis and debugging of a device. Logging information at this level may impact the performance of the device. |
| **Expected Content** | This level of logging should show function call stacks and provide a high level of debug information. |
| **Log Level** | `ALL` |
| **Definition** | ALL is the lowest possible logging level and is intended to turn on all logging. |
| **Expected Content** | Logging will occur at the most detailed level. Devices should never log messages directly to the ALL logging level. |

Table 7: Standard logging level definitions

# 6 Sensors [Required]

Sensors provide a means for a device to send monitoring data to the clients connected to it. Each sensor has a name and a type.

A sensor name should be unique within the context of a particular device and should preferably not contain any reference to the name of the device. For example, "pressure" is preferred to "device3.pressure". Sensor names may use a dotted notation to indicate a hierarchical grouping of sensors. The only purpose served by this dotted notation is to hint to users of the device how sensors might be logically arranged. E.g. "pump.pressure", "pump.voltage", "pump.current".

The sensor type is one of the datatypes listed in Section 3.

The value of a sensor at any given time is conceptually a triple containing the timestamp of the reading, the status of the reading and the value of the reading itself. This double meaning of the word "value" can be confusing but it is usually clear from the context whether the full triple or just the value of the reading is intended. The full list of possible sensor value statuses is given in Table 8.

| Status Name | Description |
|---|---|
| `unknown` | The sensor reading could not be determined. |
| `nominal` | The sensor reading is within the expected range of nominal operating values. |
| `warn` | The sensor reading is outside the nominal operating range. |
| `error` | The sensor reading indicates a critical condition for the device. |
| `failure` | Taking a sensor reading failed and seems unlikely to succeed in future without maintenance. |

Table 8: Sensor status definitions.

A client may obtain the list of sensors using `?sensor-list`, which returns the device name and type and some additional information including the units of measurement, a description of what the sensor records and a few extra parameters which are type-dependent (see the description of `#sensor-list` for details).

A client can poll the current value of a sensor, or of all sensors, using `?sensor-value`. An alternative means for obtaining updates is sensor sampling, which is described below.

## 6.1 Sensor Sampling

Sensor sampling provides a means for each client to request that the device send it updates of a sensor value. A sensor sampling strategy determines the conditions under which updates are sent. The complete list of strategies is given in Table 9. Updates are sent to the client using the `#sensor-status` message.

After a client connects to a device, no `#sensor-status` messages should be sent to it until it requests them using `?sensor-sampling`. This is true for both single-client (see Section 8) and multi-client (see Section 7) devices.

## 6.2 Requests

**REQUEST sensor-list** `?sensor-list [name]`

**name** is an optional sensor name

Before sending a reply, the sensor-list request will send a number of sensor-list inform messages. If no name parameter is sent the sensor-list request will return a sensor-list inform message for each sensor

| Strategy Name | Required? | Parameters | Description |
|---|---|---|---|
| `auto` | required | - | Report the sensor value when convenient for the device. This should never be equivalent to the `none` strategy. |
| `none` | required | - | Do not report the sensor value. |
| `period` | optional | period | Report the value approximately every `period` milliseconds. The `period` will be specified using the timestamp data format. May be implemented for sensors of any type. |
| `event` | optional | - | Report the value whenever it changes. May be implemented for sensors of any type. For `float` sensors the device will have to determine how much of a shift constitutes a real change. |
| `differential` | optional | difference | Report the value when it changes by more than `difference` from the last reported value. Maybe only be implemented for `float` and `integer` sensors. The `difference` is formatted as a float for float sensors and an integer for integer sensors. |

Table 9: Sampling strategy definitions. Required strategies *must* be implemented for all sensors.

available on the device. If a name parameter is specified, only an inform message for that sensor will be sent. On success the first reply parameter after the status code will contain the number of inform messages generated by this request. If the name parameter does not correspond to a sensor on the device, a fail reply should be sent.

**INFORM sensor-list** `#sensor-list name description units type [param [...]]`

**name** is the name of the sensor in dotted notation. This notation allows a virtual hierarchy of sensors to be represented; e.g. a name might be `rfe0.temperature`.

**description** is a human-readable description of the information provided by the sensor.

**units** is a human-readable string containing a short form of the units for the sensor value. May be blank if there are no suitable units. Examples: "kg", "packet count", "m/s". Should be suitable for display next to the value in a user interface.

**type** is the name of one of the datatypes described in Section 3.

**params** are determined by the type:

**integer** `min` and `max` value for range (inclusive)

**float** `min` and `max` values for range (inclusive)

**discrete** list of available options (as multiple arguments)

**boolean, lru, timestamp, string** no additional parameters

**REPLY sensor-list** `!sensor-list ok numSensors` where

**numSensors** is the number of sensor-list informs sent.

**REQUEST sensor-sampling** `?sensor-sampling name [strategy [param ...]]`

**name** is the name of the sensor

**strategy** specifies a sampling strategy and is one of strategies described in Table 9. If no strategy is specified, the current strategy and parameters are left unchanged and just reported in the reply.

**params** are determined by the strategy as described in Table 9.

**REPLY sensor-sampling** `!sensor-sampling ok name strategy [param ...]`

**name** is the name of the sensor

**strategy** is the name of the new sampling strategy (or the current strategy if the strategy was not updated)

**params** are the new sampling strategy parameters (or the current parameters if the strategy was not updated)

**REQUEST sensor-value** `?sensor-value [name]`

**name** an optional sensor name.

Before sending a reply, the sensor-value request will send a number of sensor-value inform messages. If no name parameter is sent the sensor-value request will return a sensor value for each sensor available on the device using a set of sensor-value inform messages. If a name parameter is specified, only an inform message for that sensor will be sent. On success the first reply parameter after the status code will contain the number of inform messages generated by this request. If the name parameter does not correspond to a sensor on the device, a fail reply should be sent.

**INFORM sensor-value** `#sensor-value timestamp_ms numSensors [(name status value) ...]`

**timestamp_ms** is the time at which the sensor value was read (formatted as a timestamp).

**numSensors** is the number of sensors reported in this message, and is followed by a corresponding number of repeats of (name status value).

**name** corresponds to one of the sensors

**status** is one of `unknown`, `nominal`, `warn`, `error`, or `failure`.

**value** is a value appropriate to the sensor's type.

The sensor-value inform message has the same structure as the asynchronous sensor-status inform except for the message name. The message name is used to determine whether the sensor value is being reported in response to a sensor-value request or as a result of sensor sampling. See section 3 for a description of how the value parameters for different types should be formatted.

**REPLY sensor-value** `!sensor-value ok numInforms`

**numInforms** is the number of sensor-value informs sent.

## 6.3 Asynchronous Informs

**INFORM sensor-status** `#sensor-status timestamp_ms numSensors [(name status value) ...]`

**timestamp_ms** is a count in milliseconds for the Unix epoch

**numSensors** is the number of sensors reported in this message, and is followed by a corresponding number of repeats of (name status value).

**name** corresponds to one of the sensors

**status** is one of `unknown`, `nominal`, `warn`, `error`, or `failure`.

**value** is appropriate for the sensor type

A sensor-status inform should be sent whenever the sensor sampling set up by the client dictates. The sensor-status inform message has the same structure as the sensor-value inform except for the message name. The message name is used to determine whether the sensor value is being reported in response to a sensor-value request or as a result of sensor sampling. See section 3 for a description of how the value parameters for different types should be formatted.

# 7 Multi-client [Optional]

KATCP-compliant devices may either support multiple simultaneous clients (in which case they should behave as described in this section) or only a single client (in which case they should follow Section 8). The multi-client option is the preferred option for devices capable of implementing it as it provides a means of monitoring a device while it is being controlled on a separate connection.

Multi-client devices need not make any arrangements to share control – they may simply accept commands from all clients. Clients should arrange to handle shared control among themselves. It is expected that usually a single client will have primary control and that other clients will only monitor the device, although this arrangement is not required by KATCP.

To assist clients in tracking what other clients are connected a `client-list` request is provided so the current list of connected clients can be retrieved. A `client-connected` inform is sent to each connected client when a new client is accepted. Both these messages are described in more detail later in this section.

Replies to requests should be sent only to the client that made the request. Inform messages generated as part of a reply should also only be sent to the client that made the request.

Whether asynchronous informs are sent to multiple clients is determined by the type of inform. Of the messages described in this document only `#log` and `#client-connected` are sent to multiple clients. All others are sent to the single client associated with the event that triggered the inform. For `#build-state`, `#version` and `#disconnect` it is the client connecting or being disconnected. For `#sensor-status` it is the client that configured the sensor sampling strategy. The `#log` informs should be sent to all clients. The `#client-connected` informs should be sent to all clients except the one that has just connected. These behaviours are summarized in Table 10.

Devices should maintain one sensor sampling strategy *per sensor per client* and send sampled values only to the client that set up the sampling strategy.

| Inform | Sent to |
|---|---|
| build-state | Client that is connecting. |
| client-connected | All clients except the one that is connecting. |
| disconnect | Client that is about to be disconnected. |
| log | All clients. |
| sensor-status | Client that configured the relevant sensor sampling. |
| version | Client that is connecting. |

Table 10: Summary of which clients asynchronous informs should be sent to.

## 7.1 Requests

**REQUEST client-list** `?client-list` Before sending a reply, the client-list request will send a client-list inform message containing the address of a client for each client connected to the device, including the client making the request.

**INFORM client-list** `#client-list addr`

**addr** The address and port the client is connected from as a single human-readable string parameter.

**REPLY client-list** `!client-list ok numClients`

**numClients** The number of #client-list inform messages sent by the corresponding request.

## 7.2 Asynchronous Informs

**INFORM client-connected** `#client-connected msg`

**msg** A description of the new client. It should include the address and port the new client connected from.

The #client-connect inform should be sent to all other clients when a new client is accepted.

# 8 Single-client [Optional]

KATCP-compliant devices may either support just a single client (in which case they should behave as described in this section) or multiple simultaneous clients (in which case they should follow Section 7). The multi-client option is the preferred option for devices capable of implementing it. The single-client option is retained for backwards compatibility with earlier versions of the protocol and to support devices implemented on minimal hardware.

The single-client specification is as far as possible intended to follow the behaviour specified for multi-client devices in the case where only a single client is connected at any one time.

If a second client attempts to connect to a single-client device, it must send the *first* client a #disconnect inform and disconnect the first client. The message parameter of the #disconnect inform should explain that a new client has connected and include the address and port of the new client (for debugging and logging purposes).

Single-client devices should not implement any of the requests or informs described in the section on multi-clients (Section 7) in order to allow the two types of devices to be easily distinguished.

For sensor sampling, single-client devices need only maintain one sensor sampling strategy *per sensor*. When a new client connects, all sampling strategies should behave as if set to the `none` strategy.

# 9 Device Configuration [Optional]

It is recommended that should a device provide a means for a client to configure it that this be done using a `configure` request which is outlined below. A client may need to use multiple `configure` requests to set different device parameters.

**REQUEST Configure** `?configure param ...`

**params** are custom for a device. For example, the first parameter might be the name of an option to set and the second might be value of the option, as in `ntp ntpserver.localdomain`.

The reply arguments are either just `ok` if the configuration described by params was accepted or a failure code and a message if setting the configuration failed.

# 10 State and Mode [Optional]

More complex devices may wish to provide information to clients about their current state or mode. It is recommended that this be done using sensors named `state` and `mode`, which should be of the `discrete` type.

If a device wishes to allow clients to explicitly switch between modes, this should be done using a `mode` request as described below.

**REQUEST mode** `?mode name`

**mode** name of the mode to change too. Should be one of the mode values reported by the mode sensor.

This request should trigger a change to the mode of the specified name. The list of modes will be device-specific but should be a subset of the possible values of the mode sensor. The reply is just `ok` if the mode change succeeded or a failure code and message if the request failed.

# A KAT Devices

This appendix applies only to devices being implemented for the KAT project. Others may find it useful background reading.

KAT is a project to build a radio telescope in the Karoo region of South Africa. Such a radio telescope contains many hardware devices that need to be monitored and controlled from a central location. The solution adopted by the KAT project attempts to standardize on the protocol described in this document for the interfaces with devices.

The KAT monitoring and control system includes the concept of device proxies. Proxies are the clients of devices. Their role is to shield the KAT software system from the details of device control. This may be necessary for a number of reasons including:

- a device might not properly support KATCP (for example, it may be a legacy device)
- the device might implement the single-client KATCP option (in which case a proxy provides a means for multiple clients to connect)
- it may be convenient to aggregate several devices into one virtual device
- the level of control provided by the device may be deemed too primitive for use by the rest of the system (in which case the proxy represents a higher-level interface to the device functionality)

Although the proxies provide a level of standardization within the KAT monitoring and control system, there are distinct advantages to having some degree of standardization at the device level too as this will:

- promote reuse of communications libraries
- reduce confusion caused by switching between widely varying protocols
- make it simpler to ensure that the device protocol satisfies KAT system requirements

## A.1 Physical context

Figure 1 shows the physical context for a device within the KAT computing system. It can be seen that the primary communication is between the device and the proxy however there may also be interface/communication with a DHCP server (for obtaining an IP address) and an NTP server (for synchronising the device time).

Details for each of the interfaces are provided in the sections below.

### A.1.1 Device - DHCP Server

It is preferred that devices obtain an IP address from a DHCP server. Where this is not possible, devices should use a static IP address. Devices that use a static IP address are strongly encouraged to make the IP address and network address configurable. The KAT DHCP server will also provide devices that use it with the address of the NTP server.

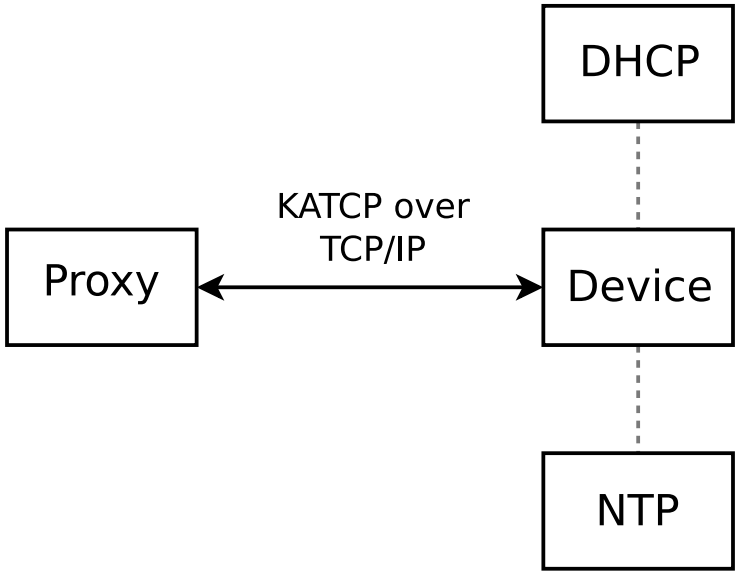


Figure 1: Context diagram showing relationship between the device and other system components.

### A.1.2 Device - NTP Server

If a device uses an NTP server, it should receive the NTP server IP address either through DHCP or alternatively, by allowing the NTP server IP to be configured by a client using the `?configure` request (see the Device Configuration module, section 9).

Devices are strongly encouraged to synchronise their local time from the NTP server to ensure that logging timestamps are accurate.

Should a particular device require more accurate time synchronization than is available through NTP, the provision of a Precision Time Protocol (PTP) master or similar mechanism may be requested.

### A.1.3 Device - Proxy

Communication between the proxy and the device uses the KATCP protocol described in the main part of this document.

An example telnet session to an antenna device is shown below to illustrate the interaction between the antenna proxy and an antenna device. Note that all requests (commands that begin with a question mark) are sent from the proxy to the device. Reply (exclamation mark) and inform (hash) messages are sent from the device to the proxy. Each block in the example corresponds either to a request (and its reply and associated informs) or to a set of asynchronous inform messages sent by the device (such as the inform messages sent on connect or the sensor status informs). Where the text is indented, line breaks have been introduced for readability in this document. Ellipses indicate where lines have been left out of the transcript for brevity.

```
#build-state acs-1.0
#version acs-1.0
#mode idle
#state operate remote braked
```

```
?configure ntp-server 192.168.1.21
!configure ok

?help
#help configure Configure\_NTP\_server\_IP\_address.\nParameters:
  \_ntp-server\_ip-address\nReturn:\_success\_ntp-server\_|\_message
#help halt Request\_antenna\_to\_prepare\_system\_for\_shutdown.
  \nReturn:\_success\_[message]
...
!help ok 8

?sensor-list
#sensor-list acs.desired-azim Desired\_azimuth\_position Deg float -230.0 230.0
#sensor-list acs.mode ACS\_operating\_mode \@ discrete idle remote-point stow
  timeout-stow local-drive access-feed error
...
!sensor-list ok 52

?sensor-sampling acs.mode period 2000
!sensor-sampling ok acs.mode period 2000

#sensor-status 514229978 1 acs.mode nominal idle
#sensor-status 514231988 1 acs.mode nominal idle
#sensor-status 514233998 1 acs.mode nominal idle
#sensor-status 514236007 1 acs.mode nominal idle

?sensor-sampling acs.mode none
!sensor-sampling ok acs.mode none

?watchdog
!watchdog ok
```

## A.2 Device Start-up and Configuration

Regardless of the mechanism used to allocate an IP address to the device, a central configuration server will be aware of what IP address a device will use. This information is used to ensure that a suitable proxy is running and configured with the address and port for connecting to the device. The proxy will attempt to retry connecting to the device periodically until the connection succeeds. Therefore, it is unimportant which component is started first.

A device may wait until the proxy configures it (see the Device Configuration module, section 9) before completing initialisation and changing to an operating state. The exact configuration of the start-up parameters is specific to each type of device. Note that a device should be responsive on the connection even prior to being configured.

## A.3 Timestamps and Leap Seconds

It is suggested that devices implement timestamps internally using 64 bit integers counting the number of milliseconds since the Unix epoch. Representing the time using a 32 bit integers counting the number of seconds since the epoch risks overflowing the count in 2038.

NTP provides a mechanism for distributing details of leap seconds in the 24 hours preceding the time of taking effect. What is required is that the NTP server which is slaved to a time source is aware of the leap seconds so that it can distribute them. Our NTP server will get absolute time from GPS which does provide a mechanism for taking leap seconds into account. We just need to ensure that each NTP client honours the leap second and inserts it into Unix time.

## A.4 Timed Command Execution

Some devices may wish to include a timestamp in some request parameters indicating that the action requested be carried out at some future time. In these cases the request should be processed and a reply sent immediately, even though the action required is yet to complete. Further requests should be processed and replied to even while the action from the earlier command is awaiting execution. Such devices should be able to queue multiple timestamped commands. The device developer and the KAT computing team should discuss and agree on whether or not timed execution and queues are necessary for each particular device. Queues at devices are generally discouraged because of the complexity of managing and debugging them.

The device queue size must be large enough so as to not impose any real- time requirements on the proxy.

It is proposed that, in order to simplify matters for such devices, the proxy may be required only to send timestamped requests in increasing time order so that it is not necessary to perform any sorting in the device.

The device should provide a means of flushing the queue so that it can be returned to a known state if errors occur.

## A.5 Logging

When a proxy receives log messages from a device, the proxy is responsible for ensuring that these log messages are forwarded to KAT's system-wide logging mechanism for storage and easy retrieval. As described in Section 5, it may still be useful for the device to also log messages to a local logging mechanism to assist debugging when a proxy or other device client is not connected.

If a language-specific logging library is used as a secondary logging mechanism and the language is Java, then the *log4j* library is preferred. If the language is Python, the standard Python *logging* module is preferred.

Note that all log messages from `INFO` level up are expected to be made visible to the telescope operator and as such should make sense in that context.

## A.6 Software Simulators

KAT device providers are also required to provide a software simulator which can be used during testing to represent the device. The simulator will serve a number of purposes:

- Allows for early integration with the proxy. The advantage of this is that it highlights where there are misunderstandings regarding the interpretation of the interface. It results in considerably smoother integration between the proxy and real device.
- Provides part of a test framework for verifying that the proxy software works correctly. As such it is a vital part of the quality assurance process for the telescope software, particularly the proxy, and allows more elaborate simulation systems to be built up.

- The provision of test hooks into the simulator provides a mechanism which supports the testing of the proxy, but is potentially also useful for testing of the device software as it can be used to simulate failure cases and the software response to them without having to generate real failures in hardware or wait for them to occur in practice which may be some time after the system has been deployed.

The benefits described above are best achieved if as much of the code base as possible used for the real device is present in the simulator. One way of achieving this is to ensure that in software all references to hardware components are encapsulated behind functional APIs. As little code as possible should reside behind these interfaces to interact with hardware. In the simulator implementation the simulation code only resides behind these interfaces. All other code is common with the hardware device.

To increase the power of the simulator one can create test hooks within the stubbed sections to introduce various effects such as out of range values (e.g. high temperatures), hardware failures (e.g. communications failures to components) as well as potentially operator interactions if appropriate (e.g. the antenna has a local operator control interface).

The simulator should be able to simulate the following functionality of the device:

| Category | Example Test Hooks |
|---|---|
| All device-specific commands | |
| High priority device-specific sensors | Normal values, out of range values |
| Initialisation sequence: build, version and configure messages | Normal behaviour, hardware configuration errors |
| States and Modes behaviours | Trigger error states and modes |
| Watchdog | Trigger failure to make software unable to respond, e.g. stuck in synchronous reads |
| Halt and Restart | |
| Logging | |

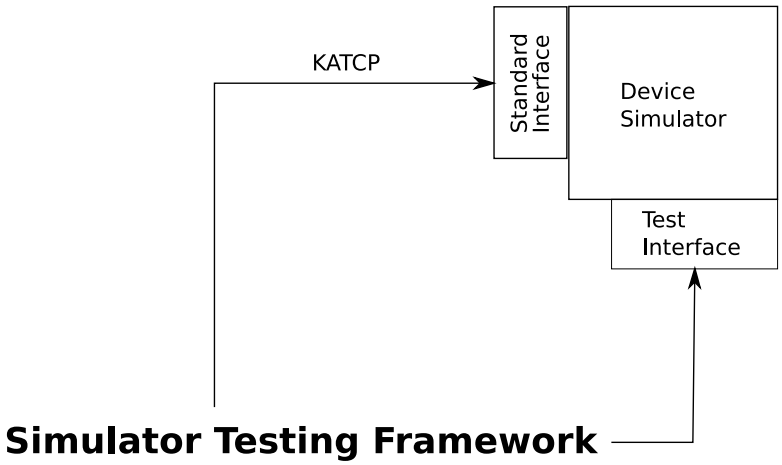


Figure 2: The testing framework connecting both to the standard interface and the test interface of the device simulator.

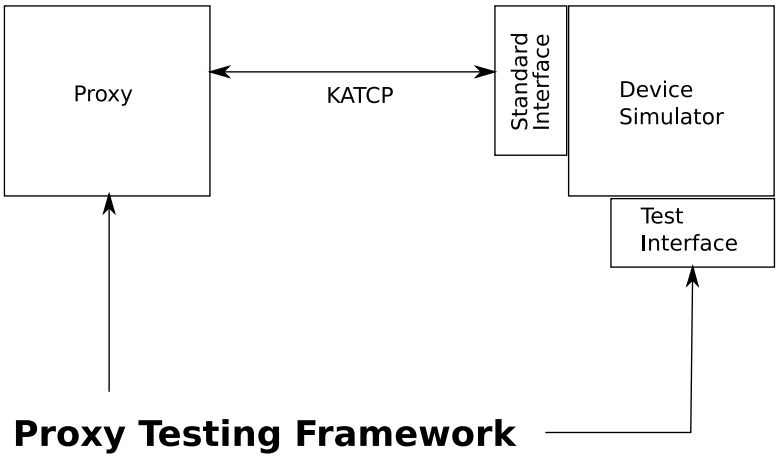


Figure 3: Testing the proxy using the device simulator.

# B Applicable and Reference Documents

## B.1 Applicable Documents

The following documents are applicable to the extent stated herein. In the event of conflict between the contents of the applicable documents and this document, the applicable documents shall take precedence.

*No applicable documents*

## B.2 Related Documents

The following documents are referenced in this document. In the event of conflict between the contents of the referenced documents and this document, this document shall take precedence.

[1] `http://en.wikipedia.org/wiki/Backus-Naur_form`.

[2] Marc Welz. CSS_DBE_Control. Technical Report NRF-XDM-6.2-ICD/001 Rev C, SKA/KAT, November 2007.