RTS2: a powerful robotic observatory manager

Petr Kubánek[a,b], Martin Jelínek[c], Stanislav Vítek[c], Antonio de Ugarte Postigo[c], Martin Nekola[a,d] and John French[e]

[a]Astronomický ústav Akademie věd České republiky, Fričova, Ondřejov, Czech Republic;
[b]INTEGRAL Science Data Center, Chemin d'Ecogia 16, Versoix, Switzerland;
[c]Instituto de Astrofísica de Andalucía (IAA-CSIC), Granada, Spain;
[d]České vysoké učení technické, Fakulta elektrotechnická, Praha, Czech Republic;
[e]University College Dublin, Dublin, Ireland

Copyright 2006 Society of Photo-Optical Instrumentation Engineers.

This paper was published in Advanced Software and Control for Astronomy, Edited by H. Lewis and A. Bridger, Proceedings of the SPIE, Volume 6274 and is made available as an electronic reprint with permission of SPIE. One print or electronic copy may be made for personal use only. Systematic or multiple reproduction, distribution to multiple locations via electronic or other means, duplication of any material in this paper for a fee or for commercial purposes, or modification of the content of the paper are prohibited.

ABSTRACT

RTS2, or Remote Telescope System, 2nd Version, is an integrated package for remote telescope control under the Linux operating system. It is designed to run in fully autonomous mode, picking targets from a database table, storing image meta data to the database, processing images and storing their WCS coordinates in the database and offering Virtual-Observatory enabled access to them. It is currently running on various telescope setups world-wide. For control of devices from various manufacturers we developed an abstract device layer, enabling control of all possible combinations of mounts, CCDs, photometers, roof and cupola controllers.

We describe the evolution of RTS2 from Python-based RTS to C and later C++ based RTS2, focusing on the problems we faced during development. The internal structure of RTS2, focusing on object layering, which is used to uniformly control various devices and provides uniform reporting layer, is also discussed.

Keywords: Robotic telescopes, Telescope Software

1. INTRODUCTION

The idea of a fully robotic telescope, which will be able to perform tasks at night when the observer is asleep, is not new. It emerged as soon as the first microprocessors which were capable of such tasks become available on the market.[1] The advanced scheduling makes a better use of telescope time. Fully autonomous operation of telescope enables real-time follow-ups, allows the unattended operations on remote sites and, in consequence, brings the costs savings.

2. EVOLUTION OF RTS2

RTS2, as the 2 in the name suggests, evolved from RTS.[2] RTS was written by four computer science students of Charles University in Prague, Faculty of Mathematics and Physics (MFF UK), as part of compulsory team work required for completion of their University degrees.

We decided to write the RTS control system in the Python language, as it seemed to be suitable for such a task. Image processing was written partly in Python, but mostly in Matlab. After half a year of designing, coding and testing we were able to control the telescope and process the images which were acquired. The project was finished as expected in June 2000, and a successful final presentation was given.

Further author information: Send correspondence to Petr Kubánek : E–mail: pkubanek@asu.cas.cz

The telescope was then put into routine operation, which revealed bottlenecks and bugs hidden in the control code. PK then decided to completely redesign the control code,[3] taking into account the experience gained during RTS development.

The primary reason for this redesign was to make the code more portable, so it could be used on other telescopes, with mounts and CCD detectors from other manufacturers. Python was abandoned in favour of the C language. Python was only capable of throwing exceptions in real-time, which is fine for some prototyping work but makes it completely unsuitable for a system which has to control a telescope, where most of such exceptions are encountered at 3 am. And it was quite hard to interface Python to low-level, mostly C, libraries used to control devices. From the beginning, RTS2 was designed to retrieve targets and log target observations to a PostgreSQL database, a major change from RTS which used text files for the same purpose.

RTS2 was initially designed in pure C, e.g. without use of object oriented programming (OOP) techniques. After two years of development and active use of RTS2 on *BART* and *BOOTES* telescopes (and beta-testing version for *FRAM* telescope), we decided to abandon C-purism in favour of OOP design in the C++ language. That decision has enabled us to produce code which is easily maintainable, and after a year of active use, we are confident that it has paid off. It may have been more appropriate to rename the new code RTS3, since there is not much remaining from the original C design, but the name RTS2 was retained.

The evolution of both the late RTS and the whole RTS2 package can be tracked in Concurrent Versions System (CVS), which we use for version control.

3. COMMUNICATION LIBRARY

Before we started to implement RTS2, possible means of communication were investigated. For communication we considered CORBA and some other, mostly GPL licensed, libraries. We did not choose CORBA, because we found it to be quite complex and difficult to understand, and not targeted at real-time systems. Furthermore at the time we made our investigation, only Java-based CORBA implementations were available. Java means trouble when one needs to access devices on system levels. Other libraries (RPC, XPA, various bus-based industrial control standards) did not exactly suit our requirements.

We finally decided to develop our own communication library, which will fit exactly to our specific needs. This was originally coded in C, but then recoded in C++. It is currently text-based, with option to make it binary-based. It supports execution of both one-time actions (set camera cooling temperature) as well as execution and reporting of actions which take some time (camera exposure, mount move).

3.1. C communication library

In the old C library, each device daemon has a master process, which creates a listening TCP/IP socket during its startup. When the listening TCP/IP socket receives a request for a new connection, the master process forks a child process, which handles all the communication with the client and with the device. That process can create threads which control long-running operations – e.g. camera exposures and mount movements. Information about the progress of long-running device operations is distributed through so-called “states”, which are in fact named integer variables.

Code is divided into three levels – generic device-independent TCP/IP communication code, device-type (mount, CCD camera, roof) specific library, which handles device commands (start of mount movement, start of exposure) transmitted over TCP/IP, and finally native drivers for every kind of device (CCD type, mount type). Using this approach it was quite easy to control devices which communicate with Linux user-space programs via standard IO lines, e.g. serial port for mount modules. But it is almost impossible to use this approach for devices with proprietary driver libraries. The problem is in non-reentrancy code, which is usually used in device drivers, since it is not possible to use non-reentrant code in a multithread environment.

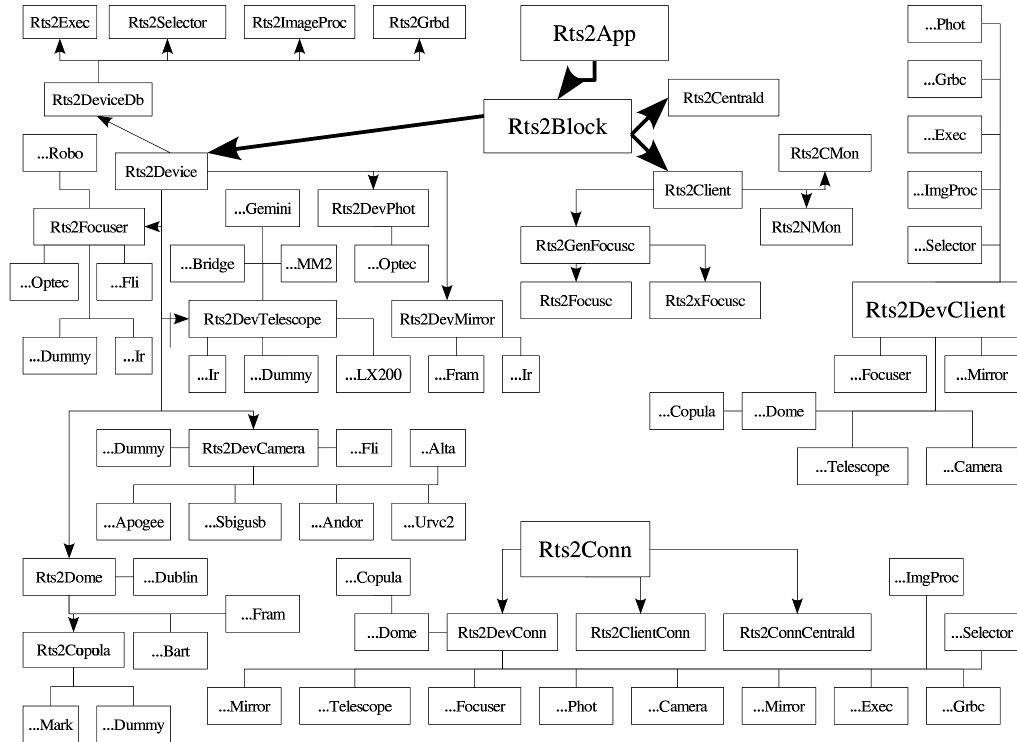

Figure 1. C++ class hierarchy

3.2. New C++ based design

This design uses C++ as the primary language, using the advantages of object oriented programming – encapsulation for assembling in one place data and operations performed on the data, polymorphism for device-type specific code, and heritage for moving from generic connection code to specific device code. It also makes use of a master device class, which processes command line arguments given to executables.

The advantages of this design can be seen when new devices are added – it is quite easy to interface a CCD camera to RTS2 in a few hours of coding if the camera kernel drivers and/or library for Linux are available. Such rapid development was almost impossible with the old multi-process approach.

4. C++ CLASS HIERARCHY

The class hierarchy is depicted in figure 1. At the beginning we wanted to avoid event distribution mechanisms, as we regarded these as being worse than direct calls of required functions. But during development of client applications, it became evident that some event distribution mechanism were required – this was the only way to avoid the need to keep and properly delete many references to one object, which can be deleted at any time. Dereferencing a deleted object would result in core dump, which was something we wanted to avoid.

So we introduced the Rts2Event class, which can transmit various content to all objects in a RTS2 executable. It is up to the postEvent method of an object what it will do with an incoming event.

The following types of classes are used:

- classes which form executables – they have init, run and sometimes idle methods. They process the command line arguments, and their run method runs the actual application code.
- connection classes, which process network data streams from other RTS2 executables.

- scripting support classes, which are used (mostly in the executor) to store and execute observation scripts.
- target classes, which are used to store the information about possible observation targets.
- other DB-bound classes, which are used to store either rows or full tables. They are used to faciliate communication with database.

Object layering in a typical device executable can be seen in figure 1 – the path can be traced going from Rts2App to Rts2DevCameraApogee, which is the executable class for Apogee CCD cameras.

5. REPORTING

For each table in the database, we created two classes – one representing one row in the table, the other representing subset of table rows. For these DB classes, we overloaded << and >> operators for output or input of class data from C++ streams.

Reporting and database manipulation applications uses those classes to provide common output and input of the table rows and table subsets.

6. EXECUTABLES IN RTS2

The RTS2 system consists of various executables. All share the same codebase for processing command line arguments and calling other system-level functions. Executables can be divided into the following groups:

rts2-centrald the name resolver and observatory housekeeper – this executable enables the finding of devices in the observatory setup and keeps track of the observatory state, e.g. whether observatory is in *off*, *standby* or *on* state, and if it is *day* or *night*.

device daemons one executable serving one device. They share common code for processing of TCP/IP commands (and registering to centrald), and implement the HW interacting layer (either through their own code or through some library). Those include daemons to drive cameras, mounts, roofs etc.

executing daemons which interact with the database and either select the next target (*rts2-selector*), execute an observation (*rts2-executor*), process images (*rts2-imgproc*), or wait and process a GCN[4] or other incoming messages (*rts2-grbd* and *rts2-auger-shooter*).

client-side monitoring programs ncurses based *rts2-mon*, console based *rts2-cmon*, X-Window image grabber *rts2-xfocusc*, console grabber *rts2-focusc*, and *rts2-soapd*, which enables access to the RTS2 system through the *Simple Object Access Protocol* (SOAP).

database querying and updating tools – various editing (*rts2-newtarget*, *rts2-target*, *rts2-plan*), modelling (*rts2-tpm* and *rts2-telmodeltest*) and reporting (*rts2-nightreport*, *rts2-targetlist*, *rts2-nightmails*) programs.

Along with drivers for real devices also the dummy drivers, which create virtual devices in computer memory, are provided. With dummy devices, a virtual RTS2 system can be created in computer memory which does not control any real devices. We use these virtual setups to test new features before they are installed on production telescope systems.

7. FAULT TOLERANCE

All RTS2 programs are designed as fault tolerant. The failure of one device does not affect other devices executing daemons. It is possible to remove and add devices during observation without affecting whole observation. For example, the computer controlling one of the cameras can be rebooted during the night, and after reboot the executor will command the camera to take images. It is even possible to restart *rts2-centrald*.

8. OBSERVATION SCRIPTING

In order to fulfil the requirements of the various setups, the observation system must enable customisable scripting of observations, so that images using various filters can be obtained.

Scripting offers commands for exposing (E command), filter change (F command), and acquisition[†] of a star or a ground calibration target (star, A and HAM commands). Because we run some setups with multiple devices, commands for synchronisation among different devices are provided: various signalling commands (SS for sending signal, SW for waiting for signal) and commands to wait for acquisition performed by guiding camera (Aw command).

Scripting attempts to make the best use of the available observing time – all commands that can be executed during mount moves or camera readouts are executed at that time. RTS2 is designed to allow early observations of possible GRB OTs. It supports a continuous mode of observing (useful for mounts with multiple detectors). When a target remains unchanged, scripts are executed in a loop on all detectors until a new target is selected.

Scripting is quite a complex issue, so we give an example in the following subsection.

8.1. Example of script execution

Suppose we have a mount with two cameras – called C0 and C1. To make the example a bit more complex, suppose we also have a photometer, called PHOT. Both cameras and photometer are equipped with filter wheels, C0 (W0[*]) with UBVRz filters, C1 (W1) with RB and photometer with UBVRizcd filters. In table 1 are scripts for target 1001.

Table 1. Example observation script

C0	F R A 0.01 20 SW 1 ifacq {loops 2 {E R 2 E U 3 E R 2 E B 3 E R 2 E z 4} } else { E 10 } }
C1	F R Aw star 10 2 SS 1 ifacq {guide 2 2}
PHOT	F R Aw SW 1 ifacq {P 10 2 R P 10 2 V P 10 2 i P 10 2 z P 10 2 d P 10 2 d P 20 2 R F d} SS 2

In figure 2 we present in a simplified form a sequence of operations as they will be executed. The executor, the central component of the RTS2 autonomous system, executes target observations. The executor is coupled with a selector, a standalone program[†], which selects the next targets and queues them to an executor queue. Next or immediate observations can be posted to an executor by real-time observing programs, such as *rts2-grbd*.

The executor waits for the end of the last exposure on the CCD in the previous target, then takes control over the system. Beside commanding the mount to move to the new target position, it also sends commands to move the filters to a requested positions. It queues filter moves to cameras and photometer, so that they can happen while the mount is moving to the target destination. The cameras drivers communicates with filter wheel devices and handle filter moves. The photometer handles the filter move on its own, as the filter wheel is controlled by the same driver.

After the mount reaches its destination and the previous image is readout from the CCD, it is saved to disk and queued to the image processing daemon *rts2-imgp*[†], the executor begins the execution of image acquisition. It commands C0, on which the acquisition is performed, to take an image. After the image is taken, the executor reads it out, saves it to a hard disk, and puts it to the immediate processing using *rts2-imgp*. After it receives the astrometry solution from JIBARO,[5] it compares the actual coordinates with the requested coordinates.

[†] acquisition refers here to the process of getting the optical axis of an instrument to a target position using the images obtained from camera

[*] W0 and W1 are physically separated filter wheel devices, controlled by camera through RTS2 filter wheel server. PHOT has an integrated filter wheel which is controlled directly through the same driver as the photometer

[†] not shown in figure

[†] not shown in figure

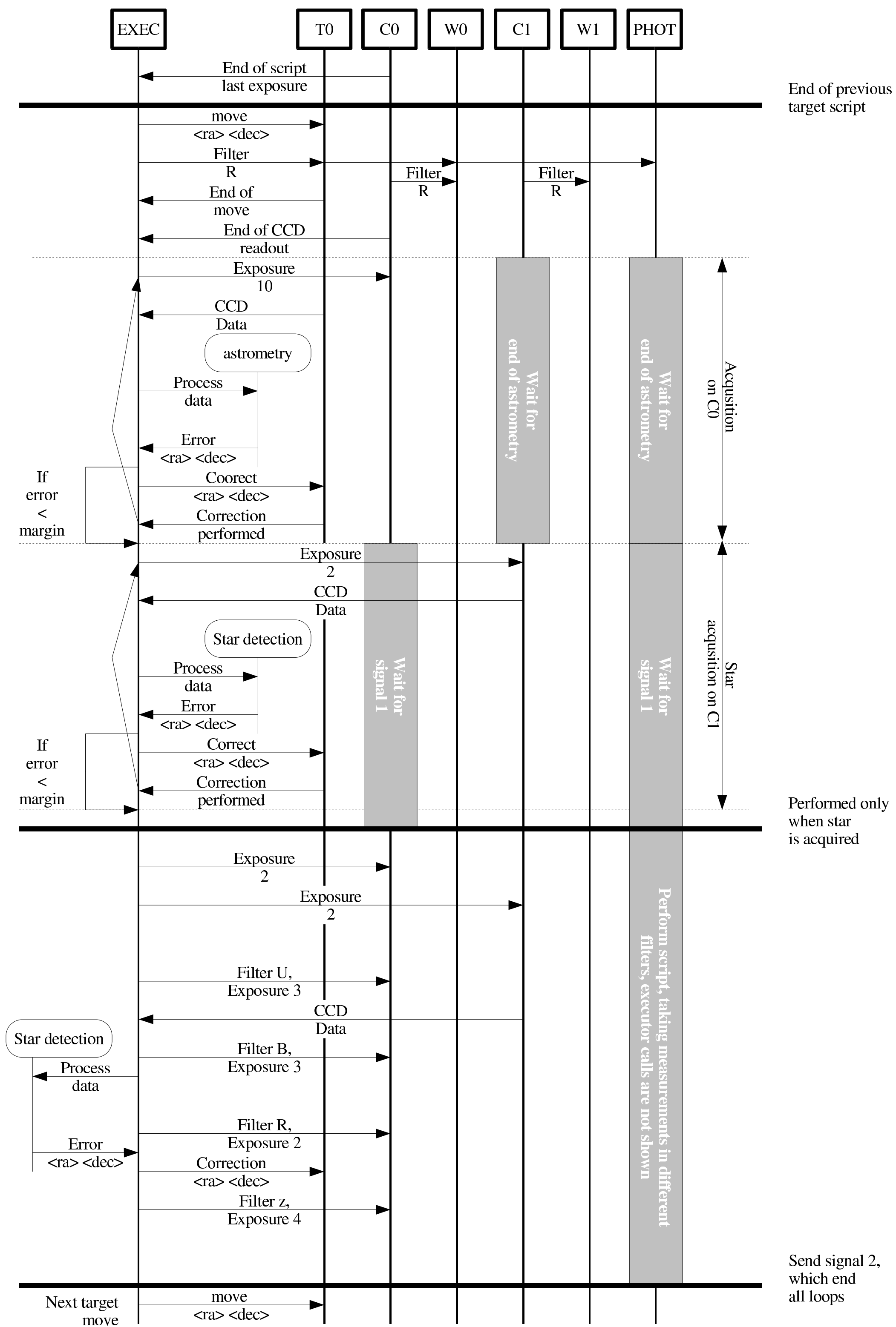

Figure 2. Execution of observing script

If the actual position of the telescope is further from the target coordinates than a specified margin, the executor sends updated coordinates to the mount, waits for the mount to finish the correction move, then tries imaging again. All other devices remain in a waiting state until the end of acquisition.

If astrometry fails and does not give a position, or if the correction sent to the telescope results in an error of more then half of the previous error, execution of the script is cancelled and the next target is executed.

If the telescope is close enough to the target position, execution of the script continues. C0 and PHOT wait for signal 1 to be sent, while C1 performs acquisition based on the brightest star in the image. Executor takes an exposure on C1, stores the data to a file on disk, runs source extraction on the data,[6] and seeks the brightest star in the obtained list. If the executor finds a bright source, it tries to align the mount to it. If it succeeds, the next command from the C1 script is executed – this sends signal 1, which unblocks the waiting devices.

After signal 1 is received, C0 and PHOT exit from their waiting state. The next execution depends on whether the last acquisition, in this case acquisition of a bright source, was successful or not. If it was successful, the executor starts to execute a loop on C0, taking images while signal 2 is received. On C1 CCD, the guide command is executed, which keeps the mount aligned to the bright source found in the previous step. Guiding is performed until signal 2 is received. The photometer starts to take measurements in different filters. At the end of the photometer measurements, signal 2 is sent, which ends the imaging loop on C0 and guiding loop on C1.

If the last acquisition was unsuccessful, a 10 second exposure is taken on C0 to determine where the mount drifted during star acquisition, and the next target is executed.

9. TARGET SCHEDULING

RTS2 enables two basic modes of operation – dispatch scheduling[7] and queue scheduling. Queue scheduling allows the observer to put a list of targets which he/she would like to observe to the database. It is convenient when the observer is at the telescope and can check local weather conditions. Dispatch scheduling is ideal for an autonomous system where there is no human intervention. We think that by facilitating both approaches users have a wider range of options at their disposal. Also, given the separation of the selection and execution components, it will be quite easy to implement other types of scheduling.

9.1. Dispatch scheduling

Dispatch scheduling uses a target merit function to calculate merit for all currently observable targets, and then selects the target with the highest merit. The advantage of this approach is its self-drivenness. Its major disadvantage is its inability to reasonably predict what will be observed during the night.

9.2. Queue scheduling

The plan is stored in a separate database table, which lists the target and starting time of the observation. The queue scheduling is actually implemented as a subclass of target which has its own merit function based on whether a plan is entered or not. The selector selects the queue scheduling based on merit as with every other observation, and puts it to the executor, which calls plan routines to select the actual observation.

As queue scheduling is still merit-based, when a target with higher merit appears [†], queue scheduling will not be selected by selector and the GRB will be properly observed.

10. TARGET HIERARCHY

To enable observation of different targets, a target hierarchy was created. Its parent object is Target, which provides the interface to various target-related calculations. It is depicted in figure 3.

There are various subclasses of Target. For example, there is a subclass for targets in solar orbit (comets and minor planets).

[†] usually a GRB

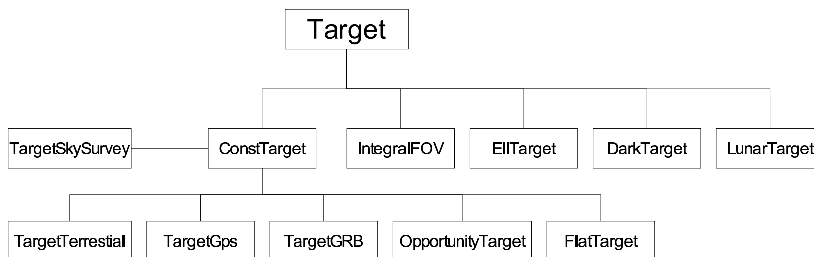

Figure 3. C++ class hierarchy for target

10.1. Target chaining

Targets sometimes use "chaining". This is when one master target holds a list of possible targets, from which it selects the actual observation to be performed.

Chaining was developed for observations of the *INTEGRAL*[8] targets. The GCN sends information on the next *INTEGRAL* pointing to *rts2-grbd*. *Rts2-grbd* stores this information to a database table. When an instance of class IntegralFOV is created, it will hold coordinates of the last *INTEGRAL* target. When such an instance is created in selector, its merit function will be high when the last IntegralFOV is above the local horizon and low when it's below the local horizon.

When IntegralFOV is created in the executor, its methods will give as a target the coordinates of the last *INTEGRAL* FOV. When *rts2-grbd* inserts new target coordinates to the database, the IntegralFOV object will retrieve them and use them for its coordinates. When the next target is below the horizon, the selector will select another target.

Target chaining is also used for other special targets. The plan target is coded using chaining – it holds a reference to the actual plan target, and provides its coordinates in overwritten methods for plan access. The plan target also holds information about the next plan entry, so queue targets get changed on time.

Another use of chaining is for the airmass monitoring target. Here, the master target selects calibration stars at various airmass ranges in order to provide information about atmospheric conditions at the observation site.

11. CURRENT SETUPS

RTS2 is controlling telescopes on three continents: four telescopes in Europe (BART,[9] *BOOTES 1A*, *BOOTES 1B*[10]), and *BOOTES-IR*[11] one in South America (*FRAM* at the southern site of the Pierre Auger observatory), and one in Africa (*Watcher*[12]). In the following paragraphs, the current setups of all instruments is described.

BOOTES-IR is 60 cm alt-az mount, located on Observatorio de Sierra Nevada (IAA-CSIC), Spain.

Two other *BOOTES* setups – 1A and 1B – are small-class telescopes, located at El Arrenosillo (CEDEA-INTA) and La Mayora (EELM-CSIC) observing stations in southern Spain.

BART is a 25 cm telescope running on a Losmandy Titan mount (controlled by Gemini interface), located at AsÚ AV ČR, Ondřejov, Czech Republic.

FRAM is operating in a remote part of Argentina at the Pierre Auger south observatory. Its main target is the measuring of atmospherics conditions for Pierre Auger observatory.

Watcher is a Paramount ME based instrument, located at the University of Free State's Boyden Observatory, South Africa.

Hardware, which is currently supported by RTS2, is described in table 2.

Table 2. RTS2 supported hardware

Devices	Manufacturer	Notes
Mounts	Software Bisque	Paramount ME with proprietary serial port driver
	Gemini	L3 and L4 of firmware, Losmandy mounts, Mountain Instrument mounts and custom mounts
	Meade	all LX 200 protocol models
	Astelco	*BOOTES-IR* mount, OpenTPL interface
	Dynostar	LX 200 protocol based mount, no longer in use
Cameras	Apogee	all models, including Alta series (with patched drivers)
	FLI	all models
	SBIG	all models, including new USB models
	Starlight Xpress	all models
Photometers	Optec	ISA I/O card based photometer, with own kernel driver
Filter wheels	FLI	all models
	Optec	Intelligent Filter Wheel (IFW)
	SBIG	filter head integrated to SBIG cameras
Focusers	Robofocuser	all models
	FLI focuser	all models
	Astelco	OpenTPL controlled focuser
	Optec	TCF-S and TCF-S3
Mirrors	Astelco	OpenTPL controlled mirror
	own design	serial controlled mirrors
Meteo stations	Davis	all models supported by modified Meteo (http://meteo.othello.ch) package
	own design	stations communicating thought TI I/O cards
Roof and cupola	own controllers	own design serial I/O board (*FRAM*, *BART*, *Watcher*), TI I/O card controlled roofs
	cupola	Profibus-based serial interface

12. FURTHER DEVELOPMENT

We will continue to develop RTS2, bringing to the code even more advanced features. We believe that RTS2 has reached the level of maturity where we are confident that it can control a 1m+ class telescope. We think RTS2 is, given the possibility to perform both manual and autonomous observations, fully suitable for the control of larger setups. And given RTS2 proof history with a lot of supported, sometimes obscure, hardware, we are not afraid that we will hit the limit and shall not be able to make one or zero-keyboard observations possible on those large setups.

ACKNOWLEDGMENTS

PK and MN would like to thank to GA AV ČR for the support via grant A3003206 and ESA PECS Project 98023. MJ would like to thank to the Spanish Ministry of Education and Science for the support via FPU grant AP2003-1407. We would like to acknowledge INTA for the hospitality provided at the INTA-CEDEA BOOTES-1 observing station.

REFERENCES

1. M. Trueblood and R. M. Genet, *Telescope control – 2nd ed.*, Willmann-Bell, 1997.
2. T. Jílek, P. Kubánek, F. Krolluper, and F. Kvapil, *Detekce astronomickch objektu s proměnnou intenzitou za pomoci robotického teleskopu*, Katedra softwarového inženýrství MFF UK, May 2000.

3. P. Kubánek, "Zdokonalení řídícího softwaru pro dalekohled BART (Improvement of the controlling software for BART telescope)," Master's thesis, Katedra softwarového inženýrství MFF UK, Oct. 2003.
4. S. D. Barthelmy, "GRB Coordinates Network (GCN): A Status Report," *American Astronomical Society Meeting Abstracts* **202**, May 2003.
5. de Ugarte Postigo *et al.*, "JIBARO: Un conjunto de utilidades para la reducción y análisis automatizado de imágenes," in *Astrofísica Robótica en España*, A. J. Castro-Tirado, B. de la Morena, and J. Torresi, eds., pp. 35–50, Ed. Sirius, Madrid, 2005.
6. E. Bertin and S. Arnouts, "SExtractor: Software for source extraction.," *Astronomy & Astrophysisc* **117**, pp. 393–404, June 1996.
7. T. Granzer, "What makes an automated telescope robotic?," *Astronomische Nachrichten* **325**, pp. 513–518, Oct. 2004.
8. C. Winkler, T. J.-L. Courvoisier, G. Di Cocco, N. Gehrels, A. Giménez, S. Grebenev, W. Hermsen, J. M. Mas-Hesse, F. Lebrun, N. Lund, G. G. C. Palumbo, J. Paul, J.-P. Roques, H. Schnopper, V. Schönfelder, R. Sunyaev, B. Teegarden, P. Ubertini, G. Vedrenne, and A. J. Dean, "The INTEGRAL mission," *Astronomy and Astrophysics* **411**, pp. L1–L6, Nov. 2003.
9. M. Jelínek, P. Kubánek, M. Nekola, and R. Hudec, "BART: an intelligent GRB and sky monitoring telescope (2000-2004)," *Astronomische Nachrichten* **325**, pp. 678–678, Oct. 2004.
10. A. J. Castro-Tirado *et al.*, "BOOTES: A Stereoscopic and Robotic Ground-Support Facility for the INTEGRAL Era," in *ESA SP-552: 5th INTEGRAL Workshop on the INTEGRAL Universe*, V. Schoenfelder, G. Lichti, and C. Winkler, eds., pp. 637–+, Oct. 2004.
11. A. J. Castro-Tirado *et al.*, "BOOTES-IR: near IR observations by a robotic system in Southern Spain," **this volume**, SPIE, 2006.
12. J. French, L. Hanlon, B. McBreen, S. McBreen, L. Moran, N. Smith, A. Giltinan, P. Meintjes, and M. Hoffman, "Watcher: A Telescope for Rapid Gamma-Ray Burst Follow-Up Observations," in *AIP Conf. Proc. 727: Gamma-Ray Bursts: 30 Years of Discovery*, E. Fenimore and M. Galassi, eds., pp. 741–744, Sept. 2004.