

---

# BRAK Software Abstraction Layer (BSAL)

Version 0.1

written by

**Ken Hughes**

Electrical and Computer Engineering  
University of the Pacific  
Stockton, CA 95219

October 24, 2000

<http://www1.uop.edu/eng/research/brak>

---

---

Copyright © 2000 Ken Hughes <khughes@uop.edu>

This book is free; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This book is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

For a copy of the GNU General Public License, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

---



# Contents

<b>1</b>	<b>Introduction to the BSAL</b>	<b>1</b>
1.1	Component Processes . . . . .	2
1.2	Interface Protocols . . . . .	4
<b>2</b>	<b>Organization of the Control Process</b>	<b>7</b>
2.1	Control Infrastructure . . . . .	8
2.2	Process and Signal Control . . . . .	9
2.3	RCI Handler . . . . .	9
2.4	Configuration Database . . . . .	9
2.5	Device and Board Objects . . . . .	9
2.5.1	Model for Device and Board Objects . . . . .	10
2.6	Real-time Support . . . . .	10
2.7	RSI Handler . . . . .	11
<b>3</b>	<b>The rci-client interface</b>	<b>13</b>
3.1	Data types . . . . .	13
3.1.1	GLib data types . . . . .	13
3.1.2	BSAL-defined data types . . . . .	14
3.2	Interface procedure . . . . .	15
3.2.1	connectToServer() . . . . .	15
3.2.2	getServerTermination() . . . . .	16
3.2.3	resetRobot() . . . . .	16
3.2.4	homeRobot() . . . . .	17
3.2.5	setSteerMotor() . . . . .	17
3.2.6	setDriveMotor() . . . . .	18
3.2.7	getSteerMotor() . . . . .	19
3.2.8	getDriveMotor() . . . . .	19
3.2.9	startUltrasonics() and startUltrasonics() . . . . .	20
3.2.10	readUltrasonics() . . . . .	20
3.2.11	startEncoders() and stopEncoders() . . . . .	21
3.2.12	clearEncoders() . . . . .	22
3.2.13	readEncoders() . . . . .	22
3.3	Example program – client-app.cc . . . . .	22

<b>4</b>	<b>Robot Control Interface (RCI) Specification</b>	<b>27</b>
4.1	Introduction . . . . .	27
4.2	Revision History . . . . .	27
4.3	General interface overview . . . . .	28
4.4	Interface packet formats . . . . .	28
4.4.1	General packets . . . . .	28
4.4.2	Poly packets (RCI_POLY bit) . . . . .	28
4.4.3	Group packets (RCI_GROUP bit) . . . . .	29
4.5	Initialization phase . . . . .	29
4.5.1	Overview . . . . .	29
4.5.2	Identification (RCI-IIId) packet . . . . .	30
4.5.3	Initialization Completed (RCI-IIInCp) packet . . . . .	31
4.5.4	Capability Inquiry (RCI-ICaIq) packet . . . . .	31
4.5.5	Capability Reply (RCI-ICaRp) packet . . . . .	31
4.6	Control phase packet formats . . . . .	32
4.6.1	Overview . . . . .	33
4.6.2	Status Inquiry (RCI-CStIq) packet . . . . .	33
4.6.3	Status Reply (RCI-CStRp) packet . . . . .	33
4.6.4	Server Termination (RCI-CSvTm) packet . . . . .	34
4.6.5	Control Acknowledge (RCI-CCnAc) packet . . . . .	34
4.6.6	Reset (RCI-CRs) packet . . . . .	35
4.6.7	Activate Motor (RCI-CAcMt) packet . . . . .	35
4.6.8	Set Motor (RCI-CStMt) packet . . . . .	36
4.6.9	Get Motor Inquiry (RCI-CGtMtIq) packet . . . . .	37
4.6.10	Get Motor Reply (RCI-CGtMtRp) packet . . . . .	37
4.6.11	Activate Sensor (RCI-CAcSn) packet . . . . .	38
4.6.12	Set Sensor (RCI-CStSn) packet . . . . .	39
4.6.13	Get Sensor Inquiry (RCI-CGtSnIq) packet . . . . .	39
4.6.14	Get Sensor Reply (RCI-CGtSnRp) packet . . . . .	40
4.6.15	Base Time Inquiry (RCI-CBsTmIq) packet . . . . .	41
4.6.16	Base Time (RCI-CBsTm) packet . . . . .	41
4.7	Data type descriptions . . . . .	42
4.7.1	mposval . . . . .	42
4.7.2	timeval . . . . .	42
4.7.3	statval . . . . .	43
4.7.4	speedval . . . . .	43
4.7.5	General data types . . . . .	44
4.8	Constant definitions . . . . .	44
4.8.1	Packet codes . . . . .	44
4.8.2	Status codes, Device subcodes, Device type command codes . . . . .	45
<b>5</b>	<b>Robot Display Interface (RDI) Specification</b>	<b>47</b>
5.1	Introduction . . . . .	47
5.2	Revision History . . . . .	47

5.3	Overall packet format . . . . .	48
5.3.1	General packets . . . . .	48
5.3.2	Poly packet (RDI_POLY bit) . . . . .	48
5.3.3	Group packet (RDI_GROUP bit) . . . . .	49
5.4	Sensor (RDI-Sn) packets . . . . .	49
5.4.1	Ultrasonic sensor (RDI_SENSOR_ULTRASONIC) packets . . . . .	49
5.4.2	Relative encoder sensor (RDI_SENSOR_RELENCODER) packets . . . . .	50
5.4.3	Absolute encoder sensor (RDI_SENSOR_ABSENCODER) packets . . . . .	50
5.4.4	Accelerometer sensor (RDI_SENSOR_ACCEL) packets . . . . .	50
5.5	Motor (RDI-Mt) packets . . . . .	51
5.5.1	DC motor (RDI_MOTOR_DC) packets . . . . .	51
5.6	Actuator (RDI-Ac) packets . . . . .	51
5.6.1	Steering wheel (RDI_ACTUATOR_STEER) packets . . . . .	52
5.6.2	Driving wheel (RDI_ACTUATOR_DRIVE) packets . . . . .	52
5.7	Robot Orientation (RDI-RbOr) packets . . . . .	52
5.7.1	Ego-centric orientation (RDI_ORIENT_EGO) packets . . . . .	53
5.7.2	Omni-centric orientation (RDI_ORIENT_OMNI) packets . . . . .	53
5.8	Miscellaneous (RDI-Ms) packets . . . . .	53
5.8.1	Base Time data (RDI_MISC_BASETIME) packets . . . . .	53
5.9	Data type descriptions . . . . .	54
5.9.1	timeval . . . . .	54
5.9.2	statval . . . . .	54
5.9.3	General data types . . . . .	54
5.10	Constant definitions . . . . .	55
5.10.1	Packet codes . . . . .	55
5.10.2	Status codes, Device subcodes, Device type command codes . . . . .	55
	Index . . . . .	57

# Chapter 1

## Introduction to the BSAL

What is the BRAK Software Abstraction Layer (or BSAL for short)? The following excerpts, taken from Brak's web pages<sup>1</sup>, give a succinct explanation:

This software package provides an abstract layer between control algorithms for Brak and the low-level interface with the hardware. In addition, it allows the use of a simulator for the robot's hardware for the purpose of testing algorithms, and also provides a mechanism by which the status of the robot (either real or simulated) can be observed through a graphical user interface (GUI). We are implementing this abstraction layer using shared memory interfaces and BSD-style sockets for interprocess communications.

... (The project) originated in Spring 1999 from two senior projects using Brak. One, a VRML/Java interface for Brak, used sockets to communicate between a web page and the robot. The other, a robot simulator, needed a way to easily allow control algorithms to be tested without the need to recompile or relink the algorithm with either the simulator or hardware interface libraries. The original model for the abstraction was conceived in the Kangaroo Chair at Alpine Meadows Ski Area (not sure if this was before or after the concussion) in November 1999. It was later modified (after more brilliant ideas came to me on Jingle Bells at Sugar Bowl) and a preliminary version tested in April 2000. Further changes began in Summer 2000 and release 0.1.0 was made public in October 2000.

Hence the impetus behind developing this software was two-fold:

- To easily develop and test applications, by providing a means of switching from “simulation mode” to “real world, move that puppy, bang your shins mode” without needing to change anything internal to the application

---

<sup>1</sup><http://www1.uop.edu/eng/research/brak/soft-bsal.html>, Oct 20, 2000

- To easily observe of the robot’s state (i.e., its sensors, motors, etc) while the robot (or simulator) is active

## 1.1 Component Processes

Figure 1.1 shows an overview of the complete BSAL system, as it is envisioned to someday exist. A brief description of each component follows.

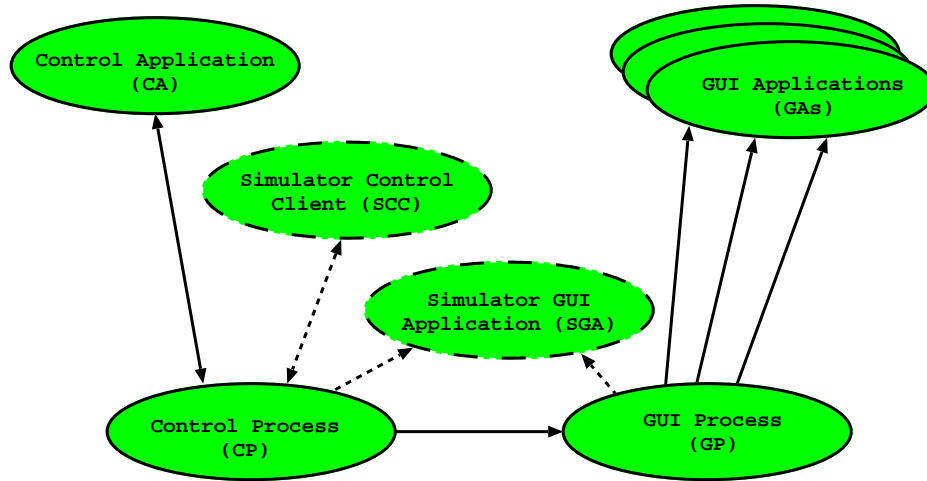


Figure 1.1: Model of BSAL

- **Control process (CP)** : the “robot hardware”. This process runs on the robot and controls access to all the robot’s physical devices (sensors and actuators). It has no intelligence per se; all major decisions are left to the control application (CA) It is, however, conceivable that this process might have some primitive survival instincts; for example, to ignore motor commands which might cause it to run into a nearby obstacle.

The control process can be implemented to work with either a virtual robot (the “simulation” mode) or a physical robot (the “real world, move that puppy, bang your shins” or “RWMTPBYS” mode). Since the interfaces between the CP and both the CA and GUI process (GP) are defined to be the only means of interaction, the ability for the CA to change seamlessly between virtual and physical robots is assured.

- **Control application (CA)** : the “robot algorithms”. This process runs (usually) on the robot and contains all the decision-making processes which control the robot. It can be something as simple as an application which shows the environment as sensed by the ultrasonics or a complex as a complete deliberative-reactive architecture (although there are would undoubtedly be those who argue that the sensing/acting is too far separated to be reactive).

- **GUI process (GP)** : the “robot omnipotentate” (for lack of a better word, I’m make one up out of two perfectly harmless ones). This process (actually two processes) runs on the robot and maintains a connection between the CP and one or more GUI applications (GAs). Its purpose in life is to gather status information about the robot from the CP and pass this information on to GUI applications (technically, they don’t have to be GUI) which will display this status information. An example would be to monitor the readings from the ultrasonic sensors.

During the initial design of the BSAL, it was realized that this status information can be viewed from two different perspectives: the perspectives of the CA and the CP. For example, once the CA activates a sensor, information begins to “flow into” the CP from that sensor in real time. This information is not transmitted back to the CA until explicitly requested. Therefore, two separate states of the robot system can co-exist: the “real-time” state (on the CP) and the “control application” state. For this reason, a separate GP exists in the system for each possible state.

The three entities above can be considered the “basic” components of a fully functional system. The remaining three entities are not required (or even possible) for all systems, but are nevertheless important in the overall scheme.

- **GUI Applications (GAs) indexGUI Applications!description indexGAs—seeGUI Applications:** the “robot view”. These are any applications, usually not running on the robot, which display information about the robot’s state based on information received from one of the GPs. The specifications for the data passed through this interface is contained elsewhere in this book (see “Robot Display Interface Specification”). This specification does not state how the data is exchanged, but the implementation of the GP in BRAK/SAL Release 0.1.0 (and most probably all future releases) communicates using TCP/IP sockets.
- **Simulator control client (SCC)** : the “virtual world manager”. This process, which would likely not run on the robot, works in conjunction with a simulator CP; in fact, it could be viewed as a part of the simulator CP for the reasons which follow. In a virtual world, the parameters of that world need to be defined so that the CP can function. One example is the definition of what objects make up the world and the robot’s location in relation to them. While this could be done initially when the CP begins running, is could (and probably would) be extremely desirable that such information could be changed in a more interactive way. This is what the SCC provides. It is an entity whose existence is hidden from other entities (such as the GP and CA), but which can be crucial if the simulator CP is to be useful.
- **Simulator GUI application (SGA) indexSGA—seeSimulator GUI application indexSimulator GUI application!description:** the “virtual world omnipotentate”. Similar in function to the GAs discussed earlier, this process allows a user to visualize the robot’s state in a simulated environment. Unlike the other GAs, this process communicates with the simulator CP to gather information (known only in the simulation) which might

be useful in interpreting the robot's state. An example would be a GUI client which shows the simulated robot's progress through a simulated environment. Unlike the SCC, this process is not necessary for the system to function; its main contribution to the system, however, is as a tool for examining the simulated performance of a CA.

## 1.2 Interface Protocols

There are two important pieces for the BSAL puzzle: the component processes described in the previous section and then interface protocols which connect the component processes. The inner workings of the components are largely an implementation detail (provided each component does what you think it does). However, just as the components need to be defined so that we know who's supposed to be going what, the interface protocols also need to be defined so we know what type of information is being transferred between the components. The biggest difference between these two pieces is that the details of the interface protocols must be very clearly defined.

Later chapters will go into the details of the two (well, three) currently defined protocols. This section merely explains the "where and why" of each protocol. Five such protocols have been identified and as of this writing three of those have been formally specified. The specifications may not be complete – they may evolve as the needs of the BSAL change – but at least they're written down.

First a word of warning on terminology. Three words are used (unfortunately somewhat interchangeably) in this document. The best definitions of each from resources on the net<sup>2</sup> are:

- *protocol*: rules determining the format and transmission of data
- *interface*: the point of interaction or communication between a computer and any other entity, such as a printer or human operator
- *specification*: a detailed description of design criteria for a piece of work

I will try to use these words with their appropriate meanings. In particular, by "interface protocol" I mean rules determining the format and transmission of data across the point of communication between two components, and by "interface protocol specification" I mean the detailed description of design criteria for an interface protocol.

- **Robot Control Interface (RCI)** : This interface connects the CA and the CP (virtual and real). It defines what commands can be sent to the robot, how status information is returned, and how the capabilities of the CP can be determined.
- **Robot Status Interface (RSI)** : This interface connects the CP (virtual and real) and the GP. It defines what status information is sent to the GP.

---

<sup>2</sup><http://www.dictionary.com>

- **Robot Display Interface (RDI)** : This interface connects the GP and the GAs or SGA. It defines what display information is sent to the GAs and how the GAs can configure the types of display information which they want to receive.
- **Simulator Control Interface (SCI)** : This interface connects the virtual CP and the SCC. It defines what commands can be sent to the virtual CP and how status information is returned.
- **Simulator Status Interface (SSI)** index SSI—see **Simulator Status Interface**: This interface connects the virtual CP and the SGA. It defines what particular simulation information is sent to the SGA.



## Chapter 2

# Organization of the Control Process

The CP (`bsal-cp`) is the central program in the BSAL. Its function is to allow a CA access to the resources of the robot, while also giving some level of diagnostic feedback. It is the most complex process within the BSAL, containing over twenty modules. The hierarchy of the CP is approximated below (it is also shown in Figure 2.1).

```
bsal-cp:
  rbp.cc
  child-routines.cc
  robot-shm.cc
  rci-server.cc
  librci.a
  server-interface.cc
    dcmotordevice.cc dcmotorboard.cc
      device.cc *
        robotconfig.cc *
          board.cc *
    psrvmotordevice.cc psrvmotorboard.cc
    absencdevice.cc absencboard.cc
      timestuff.cc
    acceldevice.cc accelboard.cc
      timestuff.cc
    relencdevice.cc relencboard.cc
      timestuff.cc
    ultradevice.cc ultraboard.cc
      timestuff.cc
```

The CP can be divided into seven different areas, depending on your definition and how arbitrary you're feeling on any given morning. Figure 2.1 below shows these areas, along with their relationship to the GUI Process and Control Application. The areas as numbered in the figure refer to (1) control infrastructure, (2) process and signal control, (3) RCI handler,

(4) configuration database, (5) device and board objects, (6) real-time support, and (7) RSI handler.

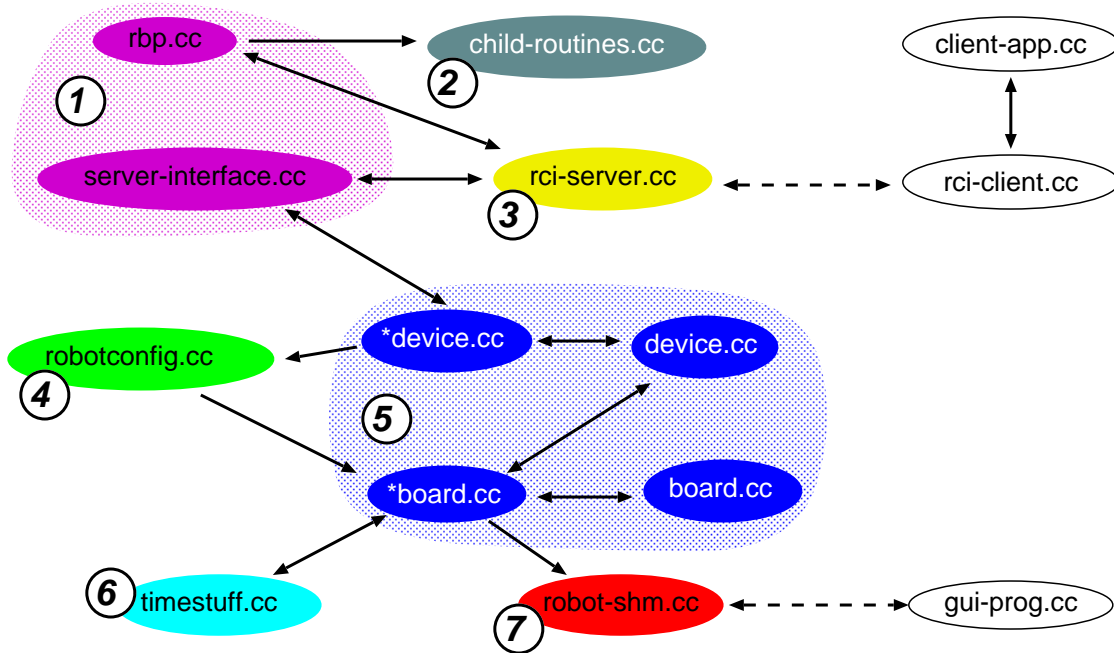


Figure 2.1: Overall organization of the Control Process

## 2.1 Control Infrastructure

The control infrastructure is the “heart” of the Control Process; it is in essence the main procedures which control and coordinate the activities in the CP. Its purpose is to translate “RCI formatted” information (Section 2.3) into the appropriate actions for the robot’s device and board objects (Section 2.5). It consists of two modules: `rbp.cc` and `server-interface.cc`. `rbp.cc` is responsible for setting up the CP (Section 2.2), containing the “main idle loop”, then servicing requests from the CA. `server-interface.cc` is a “callback” module; it contains procedures, corresponding to RCI commands, which are invoked using the robot’s device and board objects. It also maintains internal data structures such as the robot model.

The intent of the control infrastructure is to separate as much as possible the implementation details of the RCI and the device/board interface. This admittedly might not be the best idea (it seemed like a good idea at the time, although implementing it “felt icky” and that should have given a clue). In the future, it might be cleaner to integrate `server-interface.cc` and `rci-server.cc` (Section 2.3) into one module or to move `rci-server.cc` from the BSAL library `libbsalint.a` into the BSAL program source tree.

## 2.2 Process and Signal Control

The process and signal control handles additional operations needed by the CP. These include making sure the GUI Processes (GP) are alive (restarting them should they terminate), monitoring the state of sockets used for interprocess communication with the CA, and initialization of handlers for various signals used within the CP. Some of these operations are controlled within `rbp.cc` while the rest reside in `child-routines.cc`.

## 2.3 RCI Handler

The RCI handler implements the RCI specification using the BSAL library `librci.a` and the SocketMan socket library `libsm.a`. Its sole responsibility is reading and writing “RCI format” packets over a socket connection to the CA and passing information to/from `server-interface.cc` (Section 2.1) to be handled within the system. Much of this work is done by the procedure `processCmd()` within `rci-server.cc`. This module is also responsible for initially establishing the socket connection to the CA and for notifying the control infrastructure when this socket connection is terminated.

## 2.4 Configuration Database

The configuration database maintains a description of the robot for the CP. This information is contained in a text file on the system typically named `config.robot`.<sup>1</sup> `robotconfig.cc` contains all the procedures used to access the database. The purpose of the database is (eventually) to allow the system to be easily reconfigured without requiring changes to the CP. The database information is automatically accessed when a device object is created (Section 2.5) by mapping a logical device identifier to the physical board on which that device resides, as well as the actual physical device identifier on the board.

Future plans (whoever is reading this, would you like a job?) call for extension of the RCI specification so that the CA can query attributes of each device (for example, the orientation of an ultrasonic sensor). These extensions would require additions to the database for storage and retrieval of the information. A less radical addition to the system would be for the CP to automatically create the robot model (see Section 2.1) by querying the database to determine the number and type of each device.

## 2.5 Device and Board Objects

If the control infrastructure is the heart of the system, then the device and board objects must be the backbone. Maybe the control infrastructure should be the brains; then the devices/boards

---

<sup>1</sup>Remember to actually put `config.robot` somewhere on the system where it's easy to find. Also, try to figure out where it is right now in the package!

could be the stomach. Never mind. The devices and boards are responsible for the actual input and output on the system; these objects interact directly with the hardware. Board objects abstract a “system” view of the hardware while Device objects abstract a “user” view. Specific devices and boards are derived from a base class of devices and boards. The control infrastructure (Section 2.1) accesses devices only in this level; it does not access board objects. When a device object is constructed, the robot configuration database (Section 2.4) is consulted to match the device to the appropriate board and, if necessary, to construct the board object as well.

The management of the devices and boards, and their relationship to each other, is complicated (or at the very least it’s not obvious). The following section attempts to shed light on their relationship and the motivation behind their development.

### 2.5.1 Model for Device and Board Objects

The device/board model used in the BSAL considers the hardware from two different perspectives. The “device” perspective considers the hardware from the perspective of the CA; its interest is in reading/writing to a particular device without regard to how the device is physically attached to the robot. From this perspective, it can be assumed that each device has hardware dedicated to it only. The “board” perspective considers the hardware from the perspective of the (operating) system; boards may be responsible for controlling multiple devices, these devices may or may not be related, devices may not be individually accessible, etc. From this perspective, it can be assumed that the actual input/output will be done in a manner which is efficient, organized and (to some extent) “safe”.

This model is implemented in the BSAL using a device and board object for each type of device on the robot. Each of these devices and boards are derived from a base class or board and device (defined in the modules `device.cc` and `board.cc`). In the simplest case, the derived board class will define its own method for device I/O and nothing else; when device calls the board’s I/O methods the board immediately transfers the data. This would only work for boards which can support individual device I/O. More complex devices and boards (which cannot support individual device I/O) would allow the board to buffer input and/or output from the devices, or use real-time support (Section 2.6) to periodically read or write data to/from the board.

## 2.6 Real-time Support

The real-time support for the system is provided by the module `timestuff.cc`, which manipulates a list of structures of type `Time_Queue`. Each item in this list contains a pointer to a “callback procedure” in one of the board objects, as well as the time when this callback should be active (currently unused). When a particular board object (Section 2.5) is activated by its `powerOn()` method, it adds an entry to the list which guarantees that it will be “called back” in the future. The callback procedure within the board’s module is executed when the time comes, and the board performs its activity at this time. A subsequent call to the board’s `powerOff()`

method lets the board remove itself from the list.

An example of the motivation for the real-time support is the ultrasonic device. The ultrasonic sensors sense their surroundings approximately eight times a second. It is possible that the CA could try reading from the devices at a much higher rate, which would cause unnecessary transfers across the Isup2C bus (on Brak's current hardware implementation). Also, since each Isup2C bus transaction involves numerous interrupts for the microcontroller, this could potentially impact the accuracy of the sensor readings themselves since each ultrasonic ranging module also generates interrupts. Finally, the ultrasonic boards are design to transfer all sensor readings at one time. The real-time support for ultrasonic boards causes the data from all sensors to be transferred, with a frequency slightly below the board's update frequency, and stored internally within the board object. Requests for sensor data from the control infrastructure then access the internal data.

Another motivation for the real-time support (for all devices/boards, in fact) is to provide a "real-time picture" of the system for real-time GUI clients. The board's callback procedure transfers calls procedures within `robot-shm.cc` (Section 2.7) which sends the data as it is obtained. This allows a GUI client to view the actual status of the robot, which is independent of the CA's current knowledge of the robot's status.

## 2.7 RSI Handler

The Robot Status Interface (RSI) handler transfers data from the CP to the GP. The procedures used to format and send the data are contained in the module `robot-shm.cc`. This module uses the RSI library `libbsalint.a` to place items into the shared memory queues (the module `shmqueue.cc` specifically), which is the same library the GP uses to retrieve the items from the queues. As previously mentioned in Section 2.5, there two separate queues are maintained within the system for the real-time and control application GPs.



## Chapter 3

# The `rci-client` interface

The module `rci-client.cc` is designed to be the interface between a CA and the rest of the BSAL. It is intended to hide as much of the implementation details of the system from the CA as possible so that the user can focus on the task of writing algorithms. However, if everything was hidden then there would be no procedure whatsoever for the user to call; given this glaring fact, this chapter tries to explain the necessary details of the `rci-client` interface.

One note to advanced users: since the details of the RCI specification are available elsewhere, it is certainly possible to write a CA which bypasses this module and communicates “directly” with the CP. To do this, examine how `rci-client.cc` works and then use the modules found in the BSAL libraries `libbsaling.a` and `librci.a` to implement your own interface.

### 3.1 Data types

The BSAL uses three categories of data types; the standard C/C++ data types (and those included from certain C/C++ header files), Glib data types, and BSAL-specific data types. It is assumed that the user is familiar with the standard C/C++ data types and so they are not discussed here. An extremely helpful guide to C++ data objects is the “Standard Template Library Programmer’s Guide”, which can be found on-line at:

<http://www.sgi.com/Technology/STL/>.

The entire set of pages can also be found in archived format on the web; you just have to look hard enough.

#### 3.1.1 Glib data types

BSAL uses data types defined by the Glib libraries, which can be found by including `<glib.h>` in your source code. In fact, if you include any of the include files supplied in the `bsal-libraries` package, chances are this file is included already and you don’t need to do it yourself.

The Glib types used in BSAL are for integer encoding, since these are non-standard across

operating systems (mainly between x86 architectures and others). These include `guint8`, `gint8`, `guint16`, `gint16`, `guint32`, `gint32`, `guint64` and `gint64`. It's a simple pattern to remember: `g`, followed by an option `u` if the data is unsigned, followed by the number of bits (8, 16, 32 or 64).

### 3.1.2 BSAL-defined data types

There are very few data types defined by BSAL which are necessary in a control application. Currently two data types are defined for passing information back to the control application; `encoderData` and `ultraData`

```
typedef struct
{
    guint8      id;                // sensor id
    guint16     delay;             // time delay measurement (in  $\mu$ s)
    RobotTime   time;             // time/status of the last measurement
} ultraData;
```

```
typedef struct
{
    guint8      id;                // sensor id
    mposit_type position;         // position information
    RobotTime   time;             // time/status of the last measurement
} encoderData;
```

These data types use two other BSAL-defined data types; `mposit_type` and `RobotTime`. `mposit_type` is defined in `<mposval.hh>`:

```
enum mposvalType { ANGULAR, LINEAR };
enum mposvalUnit { METRIC, STANDARD, RADIAN, DEGREE };

typedef struct {
    enum mposvalType type;        // angular or linear?
    enum mposvalUnit unit;       // kind of units
    float displacement;          // position
} mposit_type;
```

This data type is an alternative to the `mposval` data field describes in the Robot Control Interface specifications. For an encoder or a motor which uses the `mposit_type` data type will return either angular values in degrees or radians, or a linear “distance” values in millimeters (metric) or tenths of an inch (standard). The range and resolution of values for each combination is:

type	unit	range	resolution
ANGULAR	RADIANS	$\pm 3.141593$	4.7683716E-7
ANGULAR	DEGREES	$\pm 180.000000$	3.0517578E-5
LINEAR	METRIC	$\pm 500000.000$	6.2500000E-2
LINEAR	STANDARD	$\pm 50000.0000$	7.8125000E-3

RobotTime is an object class defined in `<robot-time.hh>`. It uses the C data type `struct timeval` which can be found in `<sys/time.h>`. Briefly, `timeval` represents time using seconds and microseconds since a fixed reference time (January 1, 1970). Other C library routines are used to convert to and from this data type. The public interface to the `RobotTime` class includes the following procedures:

```
#include <sys/time.h>
#include <glib.h>

class RobotTime {
public:
    RobotTime ( gint32 sec = 0, gint32 usec = 0 );
    void setSec ( gint32 sec );           // set seconds
    void setUsec ( gint32 usec );        // set microseconds
    gint32 getSec ( void ) const;        // get seconds
    gint32 getUsec ( void ) const;       // get microseconds
    void changeUsec ( gint32 nusec );    // increase/decrease sec
    void changeSec ( gint32 nsec );      // increase/decrease usec
    void setTime ( const struct timeval& time ); // time conversion
    struct timeval getTime () const;     // time conversion
    void grabTime ( void );              // set to system time
};
```

Most users will use the `getTime()` operation to read the time sent from the robot, then convert it using other C library functions.<sup>1</sup> The time information which is passed through the Robot Control Interface protocol (and Robot Display/Status Interfaces protocols) is an abbreviated relative time. Internally the `rci-client` interface reconstructs the absolute time.

## 3.2 Interface procedure

### 3.2.1 connectToServer()

```
guint8 connectToServer ( string user_id,
                        guint8 const auth_key[16],
                        guint8 const hostname[] );
```

<sup>1</sup>What is not obvious at the moment is *which* C functions can be used to convert this data into a useful format.

`connectToServer()` is the first procedure which every control application must call. This procedure attempts to contact the robot control program on the host machine given in `hostname`, using the user identification `user_id` and authentication key `auth_key`.

`connectToServer()` will return one of the following status codes:

- `NO_ERROR`: a connection was established with the control process
- `ERR_CONNECT_FAILED`: a connection could not be established
- `ERR_SERVER_TERMINATE`: a connection was established, but terminated by the server during initialization
- `ERR_UNKNOWN_CAPABILITY`: a requested server capability was not available or was denied

## Notes

In Revision 0.1.0 of the BSAL software (libraries and programs), `user_id` and `auth_key` are transmitted but not checked.

### 3.2.2 `getServerTermination()`

```
guint16 getServerTermination ( void );
```

`getServerTermination()` returns the status code returned when the control program sends a Server Termination `RCI_CSvTm` packet to the control application. The values of the status code are defined in `<bsal-protocols.h>`:

- `RCI_CSvTM_ERROR`: system error
- `RCI_CSvTM_EMERGENCY`: system emergency
- `RCI_CSvTM_USER`: unknown user id
- `RCI_CSvTM_AUTHENTICATE`: bad authentication
- `RCI_CSvTM_ACCESS`: attempt to access denied capability
- `RCI_CSvTM_NOACK`: CA has not acknowledged CP queries

Three status codes (`RCI_CSvTM_USER`, `RCI_CSvTM_AUTHENTICATE` and `RCI_CSvTM_ACCESS`) cause specific codes to be returned by `connectToServer` (Section 3.2.1). Most of the other status codes may occur at any time during communication.

This procedure should only be called when a server termination condition is detected by a procedure returning a `ERR_SERVER_TERMINATE` code.

### 3.2.3 `resetRobot()`

```
guint8 resetRobot ( bool ack );
```

`resetRobot()` sends a `RCI_CRs` packet to the control process, which causes the robot to be put into a reset state. This procedure should be called shortly after a connection is established

using `connectToServer()`, and at any time that the control program decides the robot may be in an unknown state.

The return code from `resetRobot()` will be

- `RCI_CSTRP_BUSY`: reset command sent (returned when `ack` is `false`)
- `RCI_CSTRP_IDLE`: reset completed (returned when `ack` is `true`)
- `RCI_CSTRP_NO_SUPPORT`: reset not completed (returned when `ack` is `true`)

### 3.2.4 `homeRobot()`

```
guint8 homeRobot ( bool ack );
```

`homeRobot()` is designed to perform a “home” operation on various devices. Specifically, if an actuator has the ability to move to a particular position with respect to the robot’s frame of reference, or if an actuator needs to be moved in order to determine its position with respect to the robot’s frame of reference, then this device will (or should) be included in those which respond to the `homeRobot()` procedure.

The return code from `homeRobot()` will be

- `ERR_SERVER_TERMINATE`: server has terminated
- `ERR_BAD_ARGUMENT`: unexpected packet received from control process
- `RCI_CSTRP_BUSY`: home command sent (returned when `ack` is `false`)
- `RCI_CSTRP_IDLE`: home completed (returned when `ack` is `true`)
- `RCI_CSTRP_NO_SUPPORT`: home not completed (returned when `ack` is `true`)

## Notes

In Revision 0.1.0 of the BSAL libraries, `homeRobot()` performs a home operation on the steering motor only. If `ack` is `false`, the control program will not respond to other requests until either (a) the steering motor is detected at the home position, (b) the motor controller board aborts the home operation (caused when the board detects that the motor is not turning), or (c) a communication error occurs between the robot and the motor controller board.

### 3.2.5 `setSteerMotor()`

```
guint8 setSteerMotor ( const guint8 mode,
                      const gint8 dir,
                      const float& duty_cycle,
                      const float& angular_disp );
```

`setSteerMotor()` tells the control process the speed and direction to set for the steering motor. The `dir` setting is either `MOTOR_FORWARD` or `MOTOR_BACKWARD`; `MOTOR_FORWARD` sets the rotation counter-clockwise and `MOTOR_BACKWARD` sets it clockwise (`MOTOR_FORWARD` and `MOTOR_BACKWARD`

correspond to positive and negative rotation as used in sine and cosine functions). `mode` is a zero or non-zero value, where zero means use the `angular_disp` to set the speed to radians per seconds and non-zero means use the `duty_cycle` to set the speed to a particular percentage duty cycle.

The return code from `setSteerMotor()` will be

- `ERR_SERVER_TERMINATE`: server has terminated
- `ERR_BAD_ARGUMENT`: unexpected packet received from control process, or argument disagreement
- `RCI_CSTRP_ABORT`: `RCI_CStMt` command aborted
- `RCI_CSTRP_BUSY`: set motor command sent
- `RCI_CSTRP_IDLE`: set motor completed
- `RCI_CSTRP_NO_SUPPORT`: set motor not completed
- `RCI_CSTRP_INVALID_ID`: set motor refers to unknown motor

`ERR_BAD_ARGUMENT` will be returned if the procedure detects the `dir` parameter is not `MOTOR_FORWARD` or `MOTOR_BACKWARD`.

## Notes

In Revision 0.1.0 of the BSAL software (libraries and programs), only duty cycle is supported. Additionally, the `ERR_BAD_ARGUMENT` will not be returned if the user attempts to change the motor's direction while the motor is engaged. The robot's motor controller board, however, will detect this and ignore the command.

### 3.2.6 `setDriveMotor()`

```
guint8 setDriveMotor ( const guint8 mode,
                      const gint8 dir,
                      const float& duty_cycle,
                      const float& angular_disp );
```

`setDriveMotor()` tells the control process the speed and direction to set for the drive motor. The `dir` setting is either `MOTOR_FORWARD` or `MOTOR_BACKWARD`; `MOTOR_FORWARD` sets the direction forward and `MOTOR_BACKWARD` sets it backward (reverse). `mode` is a zero or non-zero value, where zero means use the `angular_disp` to set the speed to radians per seconds and non-zero means use the `duty_cycle` to set the speed to a particular percentage duty cycle.

The return code from `setDriveMotor()` will be

- `ERR_SERVER_TERMINATE`: server has terminated
- `ERR_BAD_ARGUMENT`: unexpected packet received from control process, or argument disagreement
- `RCI_CSTRP_ABORT`: `RCI_CStMt` command aborted

- RCI\_CSTRP\_BUSY: command sent
- RCI\_CSTRP\_IDLE: command completed
- RCI\_CSTRP\_NO\_SUPPORT: command not recognized
- RCI\_CSTRP\_INVALID\_ID: command refers to unknown motor

## Notes

In Revision 0.1.0 of the BSAL software (libraries and programs), only duty cycle is supported. Additionally, the ERR\_BAD\_ARGUMENT will not be returned if the user attempts to change the motor's direction while the motor is engaged. The robot's motor controller board, however, will detect this and ignore the command.

### 3.2.7 getSteerMotor()

```
guint8 getSteerMotor ( guint8& mode,
                      gint8& dir,
                      float& duty_cycle,
                      float& angular_disp,
                      guint8& pos_type,
                      guint8& pos_unit,
                      float& displacement );
```

getSteerMotor() reads the settings from the steering motor. In addition to returning the settings for dir, mode, and duty\_cycle or angular\_disp (depending on the mode settings), this procedure also returns the position of the steering wheels. pos\_type and pos\_unit are used to interpret displacement as though the values were of type mposit\_type.

The return code from getSteerMotor() will be

- ERR\_SERVER\_TERMINATE: server has terminated
- ERR\_BAD\_ARGUMENT: unexpected packet received from control process, or argument disagreement
- RCI\_CSTRP\_BUSY: command sent
- RCI\_CSTRP\_IDLE: command completed
- RCI\_CSTRP\_NO\_SUPPORT: command not recognized
- RCI\_CSTRP\_INVALID\_ID: command refers to unknown motor

### 3.2.8 getDriveMotor()

```
guint8 getDriveMotor ( guint8& mode,
                      gint8& dir,
                      float& duty_cycle,
                      float& angular_disp );
```

`getDriveMotor()` reads the settings from the drive motor. It returns the settings previously assigned by a `setDriveMotor()` procedure call. Unlike the `getSteerMotor()` procedure, no position information about the drive motor is returned. The `readEncoder()` will give this information.

The return code from `getDriveMotor()` will be

- `ERR_SERVER_TERMINATE`: server has terminated
- `ERR_BAD_ARGUMENT`: unexpected packet received from control process, or argument disagreement
- `RCI_CSTRP_IDLE`: get motor completed

### 3.2.9 `startUltrasonics()` and `stopUltrasonics()`

```
guint8 startUltrasonics ( void );
```

```
guint8 stopUltrasonics ( void );
```

These procedures are used to start or stop all the ultrasonics on the robot. This is accomplished using the `RCI_CAcSn` command.

The return code from `startUltrasonic()` and `stopUltrasonic()` will be

- `ERR_SERVER_TERMINATE`: server has terminated
- `ERR_BAD_ARGUMENT`: unexpected packet received from control process, or argument disagreement
- `RCI_CSTRP_ABORT`: `RCI_CAcSn` command aborted
- `RCI_CSTRP_BUSY`: command sent
- `RCI_CSTRP_IDLE`: command completed
- `RCI_CSTRP_NO_SUPPORT`: command not recognized
- `RCI_CSTRP_INVALID_ID`: command refers to unknown sensor

### Notes

In Revision 0.1.0 of the BSAL programs, status information as defined in the RCI Specifications is not fully implemented. `RCI_CSTRP_ABORT` will not be returned

### 3.2.10 `readUltrasonics()`

```
guint8 readUltrasonics ( vector<ultraData>& info );
```

`readUltrasonics()` are used to read data from the ultrasonics on the robot. The procedure takes as its argument a vector of type `ultraData`. This vector can either be uninitialized (containing no items) or can be an explicit list of sensors to read. An uninitialized vector will be

initialized by `readUltrasonics()` to contains all twelve ultrasonics; an explicit list must contain a valid identifier in the `id` field for each sensor. An invalid identifier will cause the procedure to terminate without reading any sensors and return the `RCI_CSTRP_INVALID_ID` code.

On a successful call, the `delay` and `time` fields will be set for each corresponding sensor.

The return code from `readUltrasonic()` will be

- `ERR_SERVER_TERMINATE`: server has terminated
- `ERR_BAD_ARGUMENT`: unexpected packet received from control process, or argument disagreement
- `RCI_CSTRP_BUSY`: command sent
- `RCI_CSTRP_IDLE`: command completed
- `RCI_CSTRP_NO_SUPPORT`: command not recognized
- `RCI_CSTRP_INVALID_ID`: command refers to unknown sensor

### Notes

In Revision 0.1.0 of the BSAL programs, status information as defined in the RCI Specifications is not implemented.

#### 3.2.11 `startEncoders()` and `stopEncoders()`

```
guint8 startEncoders ( void );
```

```
guint8 stopEncoders ( void );
```

These procedures are used to start or stop all relative encoders on the robot. This is accomplished using the `RCI_CAcSn` command.

The return code from `startEncoders()` and `stopEncoders()` will be

- `ERR_SERVER_TERMINATE`: server has terminated
- `ERR_BAD_ARGUMENT`: unexpected packet received from control process, or argument disagreement
- `RCI_CSTRP_ABORT`: `RCI_CAcSn` command aborted
- `RCI_CSTRP_BUSY`: command sent
- `RCI_CSTRP_IDLE`: command completed
- `RCI_CSTRP_NO_SUPPORT`: command not recognized
- `RCI_CSTRP_INVALID_ID`: command refers to unknown sensor

### Notes

In Revision 0.1.0 of the BSAL programs, status information as defined in the RCI Specifications is not fully implemented. `RCI_CSTRP_ABORT` will not be returned

### 3.2.12 clearEncoders()

```
guint8 clearEncoders ( void );
```

`clearEncoders()` resets the counters on the robot's relative encoders using the `RCI_CStSn` command.

The return code from `clearEncoder()` will be

- `ERR_SERVER_TERMINATE`: server has terminated
- `ERR_BAD_ARGUMENT`: unexpected packet received from control process
- `RCI_CSTRP_BUSY`: command sent
- `RCI_CSTRP_IDLE`: command completed
- `RCI_CSTRP_NO_SUPPORT`: command not recognized
- `RCI_CSTRP_INVALID_ID`: command refers to unknown sensor

### 3.2.13 readEncoders()

```
guint8 readEncoders ( vector<encoderData>& info );
```

`readEncoders()` reads data from the relative encoders on the robot. The procedure takes a its argument a vector of type `encoderData`. This vector can either be uninitialized (containing no items) or can be an explicit list of sensors to read. An uninitialized vector will be initialized by `readEncoders()` to contains all twelve ultrasonics; an explicit list must contain a valid identifier in the `id` field for each sensor. An invalid identifier will cause the procedure to terminate without reading any sensors and return the `RCI_CSTRP_INVALID_ID` code.

On a successful call, the `position` and `time` fields will be set for each corresponding sensor.

The return code from `readEncoder()` will be

- `ERR_SERVER_TERMINATE`: server has terminated
- `ERR_BAD_ARGUMENT`: unexpected packet received from control process, or argument disagreement
- `RCI_CSTRP_BUSY`: command sent
- `RCI_CSTRP_IDLE`: command completed
- `RCI_CSTRP_NO_SUPPORT`: command not recognized
- `RCI_CSTRP_INVALID_ID`: command refers to unknown sensor

## 3.3 Example program – client-app.cc

```
1 //
2 // client-app.cc -- a sample client application using the BRAL/SAL RCI
3 // protocol to control the robot.
```

```

4 //
5
6 //
7 // Copyright (C) 2000 Ken Hughes <khughes@uop.edu>
8 //
9 // This program is free software; you can redistribute it and/or
10 // modify it under the terms of the GNU General Public License
11 // as published by the Free Software Foundation; either
12 // version 2 of the License, or (at your option) any later
13 // version.
14 //
15 // This program is distributed in the hope that it will be useful,
16 // but WITHOUT ANY WARRANTY; without even the implied warranty of
17 // MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
18 // GNU General Public License for more details.
19 //
20 // You should have received a copy of the GNU General Public License
21 // along with this program; if not, write to the Free Software
22 // Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
23 //
24
25 #include <iostream>
26 #include <string>
27 #include <vector>
28
29 #include <unistd.h>
30
31 #include "rci-client.h" // interface routines for
32 // the CA
33
34 main ( int argc, unsigned char *argv[] )
35 {
36     string id = "ken";
37     guint8 auth_key[16];
38     vector<ultraData> distances;
39     vector<encoderData> counts;
40     guint8 mode;
41     gint8 dir;
42     float duty_cycle;
43     float angular_disp;
44     guint8 pos_type;
45     guint8 pos_unit;
46     float displacement;
47

```

```
48     //
49     // did the user give the name of the machine running the CP?
50     //
51
52     if ( argc != 2 )
53     {
54         exit (1);
55     }
56
57     //
58     // connection to the CP
59     //
60
61     connectToServer ( id , auth_key , argv [1] );
62
63     //
64     // initialize the robot
65     //
66
67     resetRobot ( false );
68     homeRobot ( false );
69     sleep (1);
70     startEncoders ();
71     clearEncoders ();
72
73     //
74     // turn on the ultrasonics and motors
75     //
76
77     startUltrasonics ();
78     setDriveMotor ( 1, MOTOR_REVERSE , 20.0 , 0.0 );
79
80     //
81     // take some readings
82     //
83
84     for ( int i = 0 ; i < 10 ; ++i )
85     {
86         readUltrasonics ( distances );
87         readEncoders ( counts );
88         usleep (250000);
89     }
90
91     //
```

```
92     // turn off everything
93     //
94
95     stopUltrasonics ();
96     setDriveMotor ( 1, MOTOR_REVERSE , 0.0 , 0.0 );
97 }
```

Line 61 initiates the connection to the control process using a command line argument for the host name. Although in this example no return code is checked for `connectToServer()`, most client applications should do this.

Lines 67-71 prepares the robot for operation by resetting the entire system, homing the steering wheels, and activating the drive encoders. Also notice that a system `sleep()` call is used following the `homeRobot()` call. This is necessary in Revision 0.1.0 due to a timing problem in the software. For the near future it is a Very Good Idea for all client applications to do this as well.

Lines 77-78 activate the ultrasonics and sets the drive motor for a 20% pulse-width modulation. This is then followed by the loop beginning at Line 84 which repeatedly reads the ultrasonics and drive encoder. In this example nothing is done with the sensor readings returned from the procedures, and in addition a delay is inserted between procedure calls to assure that different sensor readings are obtained.

Line 95-96 complete the control application by deactivating the ultrasonics and drive motor. Note that this is not strictly necessary as the client process will automatically deactivate all ultrasonics and motors when it detects that the control application has broken the socket connection; however, a well-behaved control application should always do this prior to terminating to avoid the (slight) chance of a delay in the control program detecting that the connection has been terminated.



## Chapter 4

# Robot Control Interface (RCI) Specification

### 4.1 Introduction

This chapter defines the protocol stream which is passed between the CP and the CA.

The stream is defined as a two-way communication. The CA is the “master” or initiator of most communication, but data may be written to or read from the CP. Since some applications may use different revisions of this protocol, it is required that the CA and CP communicate their capabilities to each other and that both entities use the latest common version of the protocol.

There should be no problems with the CA and CP losing synchronization in this protocol. This protocol stream is designed to be implemented using a socket interface, and data over a TCP socket connection are guaranteed to arrive correctly, in the proper order, so long as the connection is open. However, the possibility of long delays and loss of connection does exist; a part of this interface specification therefore needs to address what actions the program make take for the safety of the robot and how these actions are communicated to the CA if and when communication resumes.

### 4.2 Revision History

The RCI Specification has been revised as follows:

- Version 0.1, June 24, 2000
- Version 0.2, September 4, 2000
- Version 0.3, September 21, 2000
- Version 0.4, (in progress)

### 4.3 General interface overview

The CP is designed to act as a “server”, idle when there is no CA connected. The CA is designed to act as a “client”; when invoked it will first attempt to contact the CP and request access to the robot or simulator. Only one CA may connect to the CP at a time, and optionally the CP may choose to disconnect from the CA if a need arises (e.g., the CP determine the CA is not a trusted client, the CA refuses to acknowledge CP requests, etc). This is referred to as a “CP refusal of service”. The CP will transmit a status message to the CA before disconnecting to inform it of the reason for disconnecting.

The communication between the CP and CA consists of two phases: the initialization phase and the control phase. The purpose of the initialization phase is to let both entities determine the necessary configuration(s) needed for successful operation. Once this is accomplished, the control phase then lets the entities exchange information so that the robot may work safely and reliably.

### 4.4 Interface packet formats

Each packet contains an identifier code which identifies the type of packet. Depending on this code, the rest of the information in the packet can be determined using the descriptions of each packet in the following sections. Additionally, control command packets which contains information about specific sensors and motors may be encoded in *poly* or *group* packet formats. These formats are designed to allow more efficient encoding of data over the interface. Only packets with the RCI\_MULTI bit set in their identifier code can be encoded in these formats.

#### 4.4.1 General packets

data type	identifier	description
guint16	id_code	packet identifier
guint8	id_sub_code	item subclass
guint16	id	id of item
	{data packet}	data item

A *general packet* consists of a single data packet. The `id_code` field identifies the class of packet (sensor, motor, etc) while the packet-specific `id_sub_code` field identifies the specific type of packet within that class. The `id` field identifies which particular item is associated with the data packet.

#### 4.4.2 Poly packets (RCI\_POLY bit)

A *poly packet* contains one or more data packets with unrelated device identifiers. The `id_code` field identifies the class of devices (e.g., sensor, motor, etc.) and the class subcode identifies the

specific type of device within that class. The `count` field gives the total number of data packets within the overall packet; it is always greater than zero. Each data packet is preceded by an `id` field for that particular item.

data type	identifier	description
guint16	<code>id_code</code>	packet identifier
guint8	<code>id_sub_code</code>	item subclass
guint8	<code>count</code>	number of devices
guint16	<code>id</code>	id of first device
	{data packet}	data of first device
	...	
guint16	<code>id</code>	id of $n^{\text{th}}$ device
	{data packet}	data of $n^{\text{th}}$ device

#### 4.4.3 Group packets (RCI\_GROUP bit)

A *group packet* contains one or more data packets with unrelated device identifiers, as defined below. The `id_code` field identifies the class of devices (e.g., sensor, motor, etc) and the `id_sub_code` field identifies the specific type of device within that class. The `count` field gives the total number of data packets within the overall packet; it is always greater than zero. A single `id` field is transmitted, representing the identifier of the first data packet. Each successive data packet is assumed to have a consecutive id (i.e., the last data packet has identifier  $id+n-1$ ).

data type	identifier	description
guint16	<code>id_code</code>	packet identifier
guint8	<code>id_sub_code</code>	item subclass
guint8	<code>count</code>	number of devices
guint16	<code>id</code>	id of first device
	{data packet}	data of first device
	...	
	{data packet}	data of $n^{\text{th}}$ device

## 4.5 Initialization phase

The initialization phase is entered immediately after the CP and CA establish a connection. Its purpose is to guarantee that both entities are legitimate and that the capabilities of each entity is known before entering the control phase.

### 4.5.1 Overview

When the CP and CA first establish a connection, both enter an initialization phase. In this phase, they first exchange an RCI Identification (RCI-IIId) packet. The purpose of this packet

is to (a) authenticate the identity of the CA user, (b) agree upon the protocol version for further communication, and (c) other future uses. The CA and CP will then agree to exchange only those packets which are supported by both in the latest version of the interface (usually the “oldest” or least recent of the two interface versions).

Next, the CA and CP exchange a RCI Capability Inquiry (RCI-ICaIq) packet. This packet requests information from the other entity about support for zero or more types and/or versions of packets. If information is requested, the other entity responds with a RCI Capability Reply (RCI-ICaRp) packet.

The initialization phase is terminated when the CA and CP exchange RCI Initialization Completed (RCI-IInCp) packets. An entity (CA or CP) cannot proceed beyond the initialization phase (i.e., cannot transmit any other type of packet) until the RCI-IInCp packets are exchanged. The only exception is a “CP refusal of service”, when the CP may send a RCI-IInCp packet followed immediately by a RCI Server Termination (RCI-CSvTm) packet, without first waiting for a RCI-IInCp packet from the CA.

#### 4.5.2 Identification (RCI-IIId) packet

data type	identifier	description
guint16	id_code	RCI-IIId
guint8	vers_major	major version number
guint8	vers_minor	minor version number
guint8[16]	user_id	16 character user identifier
guint8[16]	auth_key	128-bit authentication key
guint8[24]	reserved	reserved for future use

The purpose of this packet is to (a) authenticate the identity of the CA user, (b) agree upon the protocol version for further communication, and (c) other possible future uses.

The major version number and minor version number fields specify the version of the interface supported by the sender. The CA and CP agree to exchange only those packets which are supported by both in the latest version of the interface (usually the “oldest” of the two versions, with the smallest version numbers). This “version synchronization” takes place immediately, so that the remainder of the packet can be correctly processed (assuming that version differences might even exist within this packet).

If there is a major difference in versions, either entity can choose to disconnect.

The user identifier and authentication fields are used to identify the user of the CA. This identifier must (a) be registered with the CP and (b) be matched with the correct authentication key. If either of these conditions is not met, the CP will immediately send a Server Termination (RCI-CSvTm) packet with the appropriate status message. User authentication also may enable only certain control commands, depending on how the user is registered with the CP. The lowest level of service is reserved for the user identifier “guest”.

### 4.5.3 Initialization Completed (RCI-IInCp) packet

data type	identifier	description
guint16	id_code	RCI-IInCp

The purpose of this packet is to inform the other entity that initialization is completed. This packet can be sent at any time following the exchange of RCI-IId packets.

### 4.5.4 Capability Inquiry (RCI-ICaIq) packet

data type	identifier	description
guint16	id_code	RCI-ICaIq
guint8	count	number of data packets
guint16	control_id	first control id code
guint8	inq_type	first inquiry type
	...	
guint16	control_id	$n^{\text{th}}$ control id code
guint8	inq_type	$n^{\text{th}}$ inquiry type

The purpose of this packet is to request information about the capabilities of specific control commands from the other entity. The packet uses a poly packet format, with the `count` field containing the number of inquiries in the packet. The `control_id` field contains the code of the command for the inquiry, and the `inq_type` field contains a code for the type of inquiry requested, as defined below:

RCI_ICAIQ_ACCESS	is command allowed
RCI_ICAIQ_VERSION	what command version

Either entity may transmit RCI-ICaIq packets.

The other entity is **not** obligated to respond to a RCI-ICaIq packet when the inquired control command is allowed. See Section 4.5.5.

### 4.5.5 Capability Reply (RCI-ICaRp) packet

data type	identifier	description
guint16	id_code	RCI-ICaRp
guint8	count	number of data packets
guint16	control_id	first control id code
	{data packet}	first inquiry response
	...	
guint16	control_id	$n^{\text{th}}$ control id code
	{data packet}	$n^{\text{th}}$ inquiry response

The purpose of this packet is to supply information to the other entity about the capability of specific control commands. A RCI-ICaRp packet is sent only in response to a RCI-ICaIq packet (see Section 4.5.4). The packet uses a poly packet format, with the `count` field containing the number of inquiries in the packet. The `control id` field identifies which control command is being described. The data fields contains contains one of the following, depending on the inquiry type requested:

<b>inquiry type</b>	<b>data type</b>	<b>description</b>
RCI_ICAIQ_ACCESS	guint8 <code>reply</code>	capability inquiry response
RCI_ICAIQ_VERSION	guint8 <code>vers_major</code>	major version number
	guint8 <code>vers_minor</code>	minor version number

The `reply` field may contain one of the following values:

RCI_ICARP_ERROR	command unrecognized
RCI_ICARP_NO_SUPPORT	command not implemented
RCI_ICARP_NO_PERMISSION	command unavailable
RCI_ICARP_PERMISSION	command available

RCI\_ICARP\_NO\_SUPPORT, RCI\_ICARP\_NO\_PERMISSION and RCI\_ICARP\_PERMISSION are returned in response to a RCI-ICaIq RCI\_ICAIQ\_ACCESS packet. The `vers_major` and `vers_minor` fields are only returned in response to a RCI-ICaIq RCI\_ICAIQ\_VERSION packet. RCI\_ICARP\_ERROR may be returned in response to any RCI-ICaIq packet, indicating that the command was not recognized. This is different than a RCI\_ICARP\_NO\_SUPPORT, which indicates the command is recognized but not implemented.

An entity may choose to *not* return a RCI\_ICARP\_PERMISSION response if it receives a RCI-ICaIq packet querying RCI\_ICAIQ\_ACCESS capability for a command which is implemented and allowed. This is intended to “speed up” the initialization process by assuming access to all commands will normally be allowed to both entities, thereby avoiding unnecessary data traffic.

Transmitting a RCI-ICaRp packet without receiving a corresponding RCI-ICaIq control command packet is an error and must not be done.

## 4.6 Control phase packet formats

In the control phase the CA initiates most communication, even though data moves somewhat independently in both directions. This phase “slaves” the CP to the commands of the “master” CA. The CP can initiate a few types of communication, mainly in the form of status and exception information. Future extensions of the interface may allow the CA to grant more control to the CP so that, when appropriate, the CP can initiate communication to the CA (for example, sending sensor updates as new data becomes available). Initialization phase packets are not allowed in the control phase; results are not defined (the entity may choose to ignore the packet, terminate, etc.) The control phase remains in effect until the interface connection is

discontinued.

#### 4.6.1 Overview

Control phase packets comprise the majority of information sent over the interface. Most, but not all, packets are initiated by the CA. *Read packets*, or packets in which the initiator of the transfer requests data from the other entity, are implicitly transferred with acknowledgment. In other words, a read packet assumes that the other entity will eventually respond to the packet by sending back data. *Write packets*, or packets in which the initiator of the transfer sends data to the other entity, may be transferred with or without acknowledgment, since again it is assumed the TCP socket implementation of the interface guarantees either transmission or failure notification. However, the initiating entity may request acknowledgment on any packet in order to avoid overwhelming the other entity (e.g., when homing/initializing the robot's steering system, the robot is unable to accept other motor commands) or to maintain "in-step" synchronization between the CA and CP. These packets should set the "Request Acknowledge" (RCI\_RQ\_ACK) bit within the id code field. In addition, the initiating entity may also choose to send a "write" packet without acknowledgment, followed by one or more Status Inquiry (RCI-CStIq) packets to determine if the last control phase packet has been processed.

#### 4.6.2 Status Inquiry (RCI-CStIq) packet

data type	identifier	description
guint16	id_code	RCI-CStIq
guint16	control_id	control id code

The purpose of this packet is for one entity to inquire the status of the other entity. Transmitting this packet will initiate the eventual transmission of a Status Reply (RCI-CStRp) packet by the other entity.

The `control_id` field can contain the code for most supported commands, but generally should only contain the code for an RCI-CStIq packet. Support for this packet is always guaranteed. Any other type of control command packet may return either a valid status or `RCI_CSTRP_NO_SUPPORT`. The valid status for other control command packets depends on the command.

A RCI-CStIq status inquiry will return one of the following RCI-CStRp replies: `RCI_CSTRP_BUSY` or `RCI_CSTRP_IDLE`.

#### 4.6.3 Status Reply (RCI-CStRp) packet

data type	identifier	description
guint16	id_code	RCI-CStRp
guint16	control_id	control id code
guint8	status_resp	

The purpose of this packet is for one entity to respond to the status inquiry of the other entity. Transmitting this packet is initiated by the previous transmission of a Status Inquiry (RCI-CStIq) packet by the other entity.

The `control_id` field contains the code requested in the previous RCI-CStIq packet. It can contain the code for most supported commands, but generally should only contain the code for an RCI-CStIq packet. Support for this packet is always guaranteed.

The `status_resp` field is specific to the control command; see the individual control command for valid replies.

A RCI-CStRp status inquiry will always return a RCI-CStRp packet with RCI\_CSTRP\_NO\_SUPPORT status.

#### 4.6.4 Server Termination (RCI-CSvTm) packet

data type	identifier	description
guint16	<code>id_code</code>	RCI-CSvTm
guint16	<code>term_code</code>	termination code

The purpose of this packet is to notify the CA that the CP is terminating the connection. This is a “friendly” way for the CP to deny access to the robot, which allows the CA to at least know why the access is denied (as opposed to simply terminating the connection).

The `term_code` field identifies the reason for terminating the connections, defined as follows:

RCI_CSVTM_ERROR	system error
RCI_CSVTM_EMERGENCY	system emergency
RCI_CSVTM_USER	unknown user id
RCI_CSVTM_AUTHENTICATE	bad authentication
RCI_CSVTM_ACCESS	attempt to access denied capability
RCI_CSVTM_NOACK	CA has not acknowledged CP queries

A RCI-CSvTm status inquiry will always return a RCI-CStRp packet with RCI\_CSTRP\_NO\_SUPPORT status.

#### 4.6.5 Control Acknowledge (RCI-CCnAc) packet

data type	identifier	description
guint16	<code>id_code</code>	RCI-CCnAc
guint16	<code>control_id</code>	control id code

The purpose of this packet is to respond to control command packets which specifically request acknowledgment (i.e., control packets which have the “Request Acknowledge” (RCI\_RQ\_ACK) bit set). This packet is sent only once, upon completion of the control command.

It is important to note that an entity which sends a control command packets containing `RCI_RQ_ACK` implicitly agrees to suspend any further transmissions until the `RCI-CCnAc` packet is received. Any other packets sent before receiving the `RCI-CCnAc` packet may be discarded by the other entity, and the results are undefined.

The `control_id` field contains the packet code of the completed control command.

Transmitting a `RCI-CCnAc` packet, except in response to a control command packet requesting acknowledgment, is an error.

A `RCI-CCnAc` status inquiry will always return a `RCI-CStRp` packet with `RCI_CSTRP_NO_SUPPORT` status.

#### 4.6.6 Reset (RCI-CRs) packet

data type	identifier	description
guint16	<code>id_code</code>	RCI-CRs

The purpose of this packet is to set the robot to a known condition. Specifically, the following actions occur when this command executes:

- stop all sensors and motors
- initialize any internal data structures

A `RCI-CRs` status inquiry will return a `RCI-CStRp` packet with one of the following codes: `RCI_CSTRP_NO_SUPPORT`, `RCI_CSTRP_BUSY` or `RCI_CSTRP_IDLE`.

#### 4.6.7 Activate Motor (RCI-CAcMt) packet

data type	identifier	description
guint16	<code>id_code</code>	RCI-CAcMt
guint8	<code>id_sub_code</code>	motor subclass
guint16	<code>id</code>	id of motor
guint8	<code>data</code>	configuration data

The purpose of this packet is to activate or deactivate one or more of the robot's motors. The CP provides for certain protection of the robot by aborting commands if necessary (e.g., turning off a motor needed for some crucial operation). If any protection is violated, the entire command is aborted and no changes are made.

The Activate Motor control command supports group and poly format packets.

The `id_sub_code` field specifies the type of motor for the control command. Currently defined motor classes include `RCI_MOTOR_DC` and `RCI_MOTOR_PSERVO`. The `RCI_MOTOR_PSERVO` class represents “pseudo-servo” motors which possess some positioning capabilities. This class can be

used, for example, for steering motors which have a homing capability but limited positioning capability.

The `id` field specifies the motor (or the first motor) associated with values in the `data` field. The `data` field contains either the value 0 (for deactivation) or 1 (for activation). Other values are undefined.

A RCI-CAcMt status inquiry will return a RCI-CStRp packet with one of the following codes: RCI\_CSTRP\_NO\_SUPPORT, RCI\_CSTRP\_ABORT, RCI\_CSTRP\_BUSY, RCI\_CSTRP\_IDLE or RCI\_CSTRP\_INVALID\_ID.

#### 4.6.8 Set Motor (RCI-CStMt) packet

data type	identifier	description
guint16	<code>id_code</code>	RCI-CStMt
guint8	<code>id_sub_code</code>	motor subclass
guint16	<code>id</code>	id of motor
	{data packet}	motor data

The purpose of this packet is to set one or more of the robot's motors to a given speed and direction. The CP provides for certain protection of the robot by aborting commands if necessary (e.g., changing the direction of a motor while moving is not allowed). If any protection is violated, the entire command is aborted and no changes are made.

The Set Motor control command supports group and poly format packets.

The `id_sub_code` field specifies the type of motor for the control command. Currently defined motor classes include RCI\_MOTOR\_DC and RCI\_MOTOR\_PSERVO.

A RCI-CStMt status inquiry will return a RCI-CStRp packet with one of the following codes: RCI\_CSTRP\_NO\_SUPPORT, RCI\_CSTRP\_ABORT, RCI\_CSTRP\_BUSY, RCI\_CSTRP\_IDLE or RCI\_CSTRP\_INVALID\_ID.

Note that RCI\_CSTRP\_IDLE code means only that the command has been successfully executed, not that the motor(s) have stopped moving. Also, no guarantee is made for the resolution of any motor.

The `id` field specifies the motor (or the first motor) associated with values in the `data` field. The interpretation of the data field is motor-specific.

##### Set DC Motor packets

Set Motor packets with the RCI\_MOTOR\_DC subcode allow the speed and direction of the motor to be specified. The `data` field for each sensor id can contain the `guint8` value RCI\_MOTOR\_DC\_SPVAL, followed by a `speedval` value (see Section 4.7.4).

##### Set P-Servo Motor packets

Set Motor packets with the RCI\_MOTOR\_PSERVO subcode allow the speed and direction of the motor to be specified or allow the motor to be repositioned to a "default" orientation. The `data`

field for each sensor id can contain:

- A `guint8` value `RCI_MOTOR_PSERVO_SPVAL`, followed by a `speedval` value (see Section 4.7.4)
- A `guint8` value `RCI_MOTOR_PSERVO_RESET`.

#### 4.6.9 Get Motor Inquiry (RCI-CGtMtIq) packet

data type	identifier	description
<code>guint16</code>	<code>id_code</code>	<code>RCI-CGtMtIq</code>
<code>guint8</code>	<code>id_sub_code</code>	motor subclass
<code>guint16</code>	<code>id</code>	id of motor
<code>guint8</code>	<code>status_code</code>	status type

The purpose of this packet is to retrieve information about the status of motors on the robot. Status information can include the current speed/direction settings and operation status of the motor and motor control board.

The Get Motor Inquiry control command supports group and poly format packets.

The `id_sub_code` field specifies the type of motor for the control command. Currently defined motor classes include `RCI_MOTOR_DC` and `RCI_MOTOR_PSERVO`.

The `id` field specifies the motor (or the first motor) associated with status type code field. The `status_code` type field contains the following codes:

<code>RCI_CGTMSTST_DATA</code>	speed and direction
<code>RCI_CGTMSTST_STATUS</code>	operation status

A `RCI-CGtMtIq` status inquiry will return a `RCI-CStRp` packet with one of the following codes: `RCI_CSTRP_NO_SUPPORT`, `RCI_CSTRP_BUSY`, `RCI_CSTRP_IDLE` or `RCI_CSTRP_INVALID_ID`,

#### 4.6.10 Get Motor Reply (RCI-CGtMtRp) packet

data type	identifier	description
<code>guint16</code>	<code>id_code</code>	<code>RCI-CGtMtRp</code>
<code>guint8</code>	<code>id_sub_code</code>	motor subclass
<code>guint16</code>	<code>id</code>	id of motor
	{data packet}	status reply data

The purpose of this packet is to return information about the status of motors on the robot requested by a `RCI-CGtMtIq` control command.

The Get Motor Reply control command supports group and poly format packets.

The `id_sub_code` field specifies the type of motor for the control command. Currently only the `RCI_MOTOR_DC` subcode is defined.

The `id` field specifies the motor (or the first motor) associated with the status reply packet. The contents of the status reply packet vary depending on the types and amount of status information requested in the previous `RCI-CGtMtIq` control command for the corresponding motor. A `RCI_CGTMST_STATUS` status request returns the same type of information for all motor types:

<b>inquiry type</b>	<b>data type</b>	<b>description</b>
<code>RCI_CGTMST_STATUS</code>	<code>statval status</code>	motor and motor controller status

The status information returned for `RCI_CGTMST_DATA` status requests is dependent on the motor class:

<b>sensor type</b>	<b>data type</b>	<b>description</b>
<code>RCI_MOTOR_DC</code>	<code>speedval motion</code>	speed and direction or position
<code>RCI_MOTOR_PSERVO</code>	<code>speedval motion</code>	speed and direction or position
	<code>mposval position</code>	position information

The `position` field contains information describing the motor's position be as described in Section 4.7.1.

A `RCI-CGtMtRp` status inquiry will always return a `RCI-CStRp` packet with `RCI_CSTRP_NO_SUPPORT` status.

#### 4.6.11 Activate Sensor (RCI-CAcSn) packet

<b>data type</b>	<b>identifier</b>	<b>description</b>
<code>guint16</code>	<code>id_code</code>	<code>RCI-CAcSn</code>
<code>guint8</code>	<code>id_sub_code</code>	sensor subclass
<code>guint16</code>	<code>id</code>	id of sensor
<code>guint8</code>	<code>data</code>	configuration data

The purpose of this packet is to activate or deactivate one or more of the robot's sensors. The CP provides protection of the robot and environment; if a protection is violated, the entire command is aborted and no changes are made.

The Activate Sensor control command supports group and poly format packets.

The `id_sub_code` field specifies the type of sensor for the control command. Currently defined sensor classes include `RCI_SENSOR_ULTRASONIC`, `RCI_SENSOR_RELENCODER`, `RCI_SENSOR_ABSENCODER` and `RCI_SENSOR_ACCEL`. The `data` field contains either the value 0 (for deactivation) or 1 (for activation). Other values are undefined.

A `RCI-CAcSn` status inquiry will return a `RCI-CStRp` packet with one of the following codes: `RCI_CSTRP_NO_SUPPORT`, `RCI_CSTRP_ABORT`, `RCI_CSTRP_BUSY`, `RCI_CSTRP_IDLE` or `RCI_CSTRP_INVALID_ID`.

**4.6.12 Set Sensor (RCI-CStSn) packet**

<b>data type</b>	<b>identifier</b>	<b>description</b>
guint16	<code>id_code</code>	RCI-CStSn
guint8	<code>id_sub_code</code>	sensor subclass
guint16	<code>id</code>	id of sensor
	<code>{data packet}</code>	sensor data

The purpose of this packet is to set the configuration of one or more of the robot's sensors. The CP provides protection of the robot and environment; if a protection is violated, the entire command is aborted and no changes are made.

The Set Sensor control command supports group and poly format packets.

The `id_sub_code` field specifies the type of sensor for the control command. Currently defined sensor classes include `RCI_SENSOR_ULTRASONIC` and `RCI_SENSOR_RELENCODER`.

A RCI-CAtSn status inquiry will return a RCI-CStRp packet with one of the following codes: `RCI_CSTRP_NO_SUPPORT`, `RCI_CSTRP_ABORT`, `RCI_CSTRP_BUSY`, `RCI_CSTRP_IDLE` or `RCI_CSTRP_INVALID_ID`.

The `id` field specifies the sensor (or the first sensor) associated with values in the `data` field. The interpretation of the data field is sensor-specific.

**Set Ultrasonic Sensor packets**

Set Sensor packets with the `RCI_SENSOR_ULTRASONIC` subcode allow the firing order and timing of the ultrasonics to be specified. The `data` field for each sensor id can contain the `guint8` value `RCI_SENSOR_ULTRASONIC_ORDER` or `RCI_SENSOR_ULTRASONIC_TIMING`, with additional data specifying the order or timing. These functions are currently not implemented.

**Set Relative Encoder Sensor packets**

Set Sensor packets with the `RCI_SENSOR_RELENCODER` subcode allow the relative encoders have their counts cleared. The `data` field for each sensor id can contain the `guint8` value `RCI_SENSOR_RELENCODER_CLEAR`.

**4.6.13 Get Sensor Inquiry (RCI-CGtSnIq) packet**

<b>data type</b>	<b>identifier</b>	<b>description</b>
guint16	<code>id_code</code>	RCI-CGtSnIq
guint8	<code>id_sub_code</code>	sensor subclass
guint16	<code>id</code>	id of sensor
guint8	<code>status_code</code>	status type

The purpose of this packet is to retrieve information about the status of sensors on the robot. Status information can include the most recent sensor data and and operation status of the sensor and sensor control board.

The Get Sensor Inquiry control command supports group and poly format packets.

The `id_sub_code` field specifies the type of sensor for the control command. Currently defined sensor classes include `RCI_SENSOR_ULTRASONIC`, `RCI_SENSOR_RELENCODER`, `RCI_SENSOR_ABSENCODER` and `RCI_SENSOR_ACCEL`.

The `id` field specifies the sensor (or the first sensor) associated with the `status_code` field. The contents of the status type code field specifies the type of status information being requested:

<code>RCI_CGTSNST_DATA</code>	sensor data
<code>RCI_CGTSNST_STATUS</code>	operation status

A `RCI-CGtSnIq` status inquiry will return a `RCI-CStRp` packet with one of the following codes: `RCI_CSTRP_NO_SUPPORT`, `RCI_CSTRP_BUSY`, `RCI_CSTRP_IDLE` or `RCI_CSTRP_INVALID_ID`.

#### 4.6.14 Get Sensor Reply (RCI-CGtSnRp) packet

<b>data type</b>	<b>identifier</b>	<b>description</b>
guint16	<code>id_code</code>	<code>RCI-CGtSnRp</code>
guint8	<code>id_sub_code</code>	sensor subclass
guint16	<code>id</code>	id of sensor
	{data packet}	status reply data

The purpose of this packet is to return information about the status of sensors on the robot requested by a `RCI-CGtSnIq` control command.

The Get Sensor Reply control command supports group and poly format packets.

The `id_sub_code` field specifies the type of sensor for the control command. Currently defined sensor classes include `RCI_SENSOR_ULTRASONIC`, `RCI_SENSOR_RELENCODER`, `RCI_SENSOR_ABSENCODER` and `RCI_SENSOR_ACCEL`.

The `id` field specifies the sensor (or the first sensor) associated with status reply packet. The contents of the status reply packet vary depending on the types and amount of status information requested in the previous `RCI-CGtSnIq` control command for the corresponding sensor and by the class of sensor. A `RCI_CGTSNST_STATUS` status request returns the same type of information for all sensor types:

<b>inquiry type</b>	<b>data type</b>	<b>description</b>
<code>RCI_CGTSNST_STATUS</code>	statval status	sensor and sensor controller status

The status information returned for `RCI_CGTSNST_DATA` status requests is dependent on the sensor class:

<b>sensor type</b>	<b>data type</b>	<b>description</b>
RCI_SENSOR_ULTRASONIC	guint16 <code>delay</code>	distance data (in microseconds)
RCI_SENSOR_RELENCODER	gint16 <code>d_count</code>	change in position (in counts)
	gint16 <code>count</code>	accumulated position (in counts)
RCI_SENSOR_ABSENCODER	gint16 <code>count</code>	accumulated position (in counts)
RCI_SENSOR_ACCEL	float <code>accel</code>	acceleration (in gravities)

A RCI-CGtSnRp status inquiry will always return a RCI-CStRp packet with RCI\_CSTRP\_NO\_SUPPORT status.

#### 4.6.15 Base Time Inquiry (RCI-CBsTmIq) packet

<b>data type</b>	<b>identifier</b>	<b>description</b>
guint16	<code>id_code</code>	RCI-CBsTmIq

The purpose of this packet is to request the current base time from the CP via a RCI Base Time (RCI-CBsTm) packet. The robot measures time in milliseconds, but the `statval` data type sent to the CA in status replies (such as the Get Motor Reply control command) is relative to a fixed “base time”; this avoids sending 64 bits of time information on every update. Since this base time is kept by the CP, the CA needs the CP to transfer this information periodically (approximately once every 4.6 hours).

The CP should be designed to send RCI-CBsTm packets immediately upon entering the control phase and periodically thereafter. In theory, there is no need for this command but is included as a convenience.

A RCI-CBsTmIq status inquiry will always return a RCI-CStRp packet with RCI\_CSTRP\_NO\_SUPPORT status.

#### 4.6.16 Base Time (RCI-CBsTm) packet

<b>data type</b>	<b>identifier</b>	<b>description</b>
guint16	<code>id_code</code>	RCI-CBsTm
timeval	<code>time</code>	absolute time

The purpose of this packet is to synchronize the base time between the CP and CA. The `time` field contains the robot’s “base time” as described in Section 4.7.2.

The CP should be designed to send RCI-CBsTm packets immediately upon entering the control phase and periodically (at least once every 4.6 hours) thereafter. Whenever the base time is updated on the CP, a RCI-CBsTm packet must be sent to avoid time skew.

A RCI-CBsTm status inquiry will always return a RCI-CStRp packet with RCI\_CSTRP\_NO\_SUPPORT status.

## 4.7 Data type descriptions

The data types used in the specification are described below.

### 4.7.1 mposval

<b>data type</b>	<b>description</b>
typedef bit 31:31 --	reserved
typedef bit 30:30 <b>type</b>	angular (=0) or linear (=1)
typedef bit 29:29 --	reserved
typedef bit 28:28 <b>units</b>	displacement units (see below)
typedef bit 27:24 --	reserved
typedef bit 23:23 <b>sign</b>	displacement sign bit
typedef bit 22:0 <b>disp</b>	displacement

`mposval` represents positional information for a pseudo-servo motor. Currently two bits are used to indicate the type and units for the position displacement. The type indicates whether the motor's displacement is angular or linear. Motors such as steering motors would use angular displacement while drive motors would use linear displacement. The unit indicates whether an angular motor's displacement is in radians(=0) or degrees(=1) or a linear motor's displacement is metric(=0) or standard(=1). The displacement is a 24-bit fixed point signed magnitude number. The size of the displacements' whole and fractional portions depends by the type and units; the encoding uses the minimum number of bits necessary to represent the largest whole value.

<b>type</b>	<b>unit</b>	<b>range(bits)</b>	<b>resolution</b>
angular(=0)	radians(=0)	3.141593(2)	4.7683716E-7
angular(=0)	degrees(=1)	180.000000(8)	3.0517578E-5
linear(=1)	metric(=0)	500000.000(19)	6.2500000E-2
linear(=1)	standard(=1)	50000.0000(16)	7.8125000E-3

Note that the type and unit values are defined by the CP and cannot be set or modified by the CA.

`mposval` values are transferred as a 32-bit value, typically as a `guint32` type. The unused bits are reserved for future use; their value is undefined and should not be used.

### 4.7.2 timeval

<b>data type</b>	<b>description</b>
<code>__time_t</code>	seconds time
<code>__time_t</code>	microseconds

`timeval` represents time in seconds and microseconds. The seconds, when used with Unix system functions such as `gettimeofday(2)`, represents elapsed seconds since 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC). It is defined in the header file `<sys/time.h>`.

### 4.7.3 statval

data type	description
typedef bit 0:0 <code>on</code>	on (=1) or off (=0)
typedef bit 1:1 <code>up</code>	up (=1) or down (=0)
typedef bit 2:2 <code>ok</code>	ok (=1) or error (=0)
typedef bit 3:7 <code>reserved</code>	reserved for future use
typedef bit 8:31 <code>rtimeval</code>	elapsed time (in milliseconds)

`statval` represents general status information about an object. It consists of a 8-bit status bitfield (defining individual operating characteristics) and a 24-bit elapsed time in milliseconds. The elapsed time used by all items is relative to the robot's base time. As such, it can represent intervals up to approximately 4.6 hours in length.

### 4.7.4 speedval

`speedval` represents the speed and direction for a motor. Speed is specified either in terms of a motor duty cycle or an rotational speed given in thousandths (0.001) radians per second. The sixteen bits in the motor speed/direction field represent the following for DC and pseudo-servo motors:

data type	description
typedef bit 15:15 <code>type</code>	duty cycle (=1) or distance (=0)
typedef bit 14:12 <code>--</code>	reserved
typedef bit 11:11 <code>sign</code>	sign of value
typedef bit 10:0 <code>magnitude</code>	speed setting

Note the sign and magnitude are independent (i.e., the representation is not two's complement but signed magnitude).

The range of the magnitude for each representation is as follows:

- duty cycle: 0% to 100% (0 to  $2^{11}-1$ )
- radians: 0 to 2.047 rad/s (0 to  $2^{11}-1$ )

### 4.7.5 General data types

<b>data type</b>	<b>description</b>
float	single-precision IEEE floating point (4 bytes)
gint32	signed short integer (4 bytes)
guint32	unsigned short integer (4 bytes)
gint16	signed short integer (2 bytes)
guint16	unsigned short integer (2 bytes)
gint8	signed short byte (1 byte)
guint8	unsigned short byte (1 byte)

All of these (except float) are defined in the Glib/GTK+ include file `<glib.h>`.

## 4.8 Constant definitions

### 4.8.1 Packet codes

<b>Packet name</b>	<b>Acronym</b>	<b>value</b>
Identification	RCI-IIId	0x0010
Initialization Completed	RCI-IIInCp	0x0011
Capability Inquiry	RCI-ICaIq	0x0012
Capability Reply	RCI-ICaRp	0x0013
Status Inquiry	RCI-CStIq	0x0100
Status Reply	RCI-CStRp	0x0101
Server Termination	RCI-CSvTm	0x0102
Control Acknowledge	RCI-CCnAc	0x0103
Reset	RCI-CRs	0x0104
Base Time Inquiry	RCI-CBsTmIq	0x0106
Base Time	RCI-CBsTm	0x0107
Activate Motor	RCI-CAcMt	0x0200
Set Motor	RCI-CStMt	0x0201
Get Motor Inquiry	RCI-CGtMtIq	0x0202
Get Motor Reply	RCI-CGtMtRp	0x0203
Activate Sensor	RCI-CAtSn	0x0210
Set Sensor	RCI-CStSn	0x0211
Get Sensor Inquiry	RCI-CGtSnIq	0x0212
Get Sensor Reply	RCI-CGtSnRp	0x0213
Multi-format bit	RCI_MULTI	0x0200
Request Acknowledge bit	RCI_RQ_ACK	0x0800
Group Packet bit	RCI_GROUP	0x8000
Poly Packet bit	RCI_POLY	0x4000

## 4.8.2 Status codes, Device subcodes, Device type command codes

Identifier	value
<i>Control Get Motor Status codes</i>	
RCI_CGTMTST_DATA	0x01
RCI_CGTMTST_STATUS	0x02
RCI_CGTMTST_TIME	0x03
<i>Control Get Sensor Status codes</i>	
RCI_CGTSNST_DATA	0x04
RCI_CGTSNST_STATUS	0x05
RCI_CGTSNST_TIME	0x06
<i>Control Status Reply codes</i>	
RCI_CSTRP_UNDEFINED	0x08
RCI_CSTRP_ABORT	0x09
RCI_CSTRP_BUSY	0x0a
RCI_CSTRP_IDLE	0x0b
RCI_CSTRP_INVALID_ID	0x0c
RCI_CSTRP_NO_SUPPORT	0x0d
<i>Control Server Termination codes</i>	
RCI_CSVTM_ACCESS	0x10
RCI_CSVTM_AUTHENTICATE	0x11
RCI_CSVTM_EMERGENCY	0x12
RCI_CSVTM_ERROR	0x13
RCI_CSVTM_NOACK	0x14
RCI_CSVTM_USER	0x15
<i>Initialization Capability Inquiry codes</i>	
RCI_ICAIQ_ACCESS	0x18
RCI_ICAIQ_SUPPORT	0x19
RCI_ICAIQ_VERSION	0x1a
<i>Initialization Capability Reply codes</i>	
RCI_ICARP_ERROR	0x20
RCI_ICARP_NO_PERMISSION	0x21
RCI_ICARP_NO_SUPPORT	0x22
RCI_ICARP_PERMISSION	0x23
<i>Control Sensor Type subcodes</i>	
RCI_SENSOR_ABSENCODER	0x80
RCI_SENSOR_ACCEL	0x81
RCI_SENSOR_RELENCODER	0x82
RCI_SENSOR_ULTRASONIC	0x83

<b>Identifier</b>	<b>value</b>
<i>Control Sensor Relative Encoder command codes</i>	
RCI_SENSOR_RELENCODER_CLEAR	0x90
<i>Control Sensor Ultrasonic command codes</i>	
RCI_SENSOR_ULTRASONIC_ORDER	0x91
RCI_SENSOR_ULTRASONIC_TIMING	0x92
<i>Control Motor Type subcodes</i>	
RCI_MOTOR_DC	0xc0
RCI_MOTOR_PSERVO	0xc1
<i>Control Motor DC Motor command codes</i>	
RCI_MOTOR_DC_SPVAL	0xd0
<i>Control Motor DC P-Servo command codes</i>	
RCI_MOTOR_PSERVO_SPVAL	0xd1
RCI_MOTOR_PSERVO_RESET	0xd2

## Chapter 5

# Robot Display Interface (RDI) Specification

### 5.1 Introduction

This chapter defines the protocol stream which is passed from the GP to monitoring software such as a GUI client application.

The stream is currently defined as a one-way communication. The client application is expected to recognize all packets described in this document; any packets which are unimplemented or unused in the client application can then be discarded.

Also, if the client loses its place in the stream there is no way to cause the server to re-synchronize. However, since this code is designed to be used with other BRAK socket layer library routines, this should not be an issue for most client applications.

Future extensions to this protocol stream will define methods for client applications to communicate with the server so that capabilities of the client can be identified and the types of data sent over the stream to the client can be tailored.

### 5.2 Revision History

The RDI Specification has been revised as follows:

- Version 0.1, March 31, 2000
- Version 0.2, April 1, 2000
- Version 0.3, June 28, 2000
- Version 0.4, in progress

### 5.3 Overall packet format

Each packet contains an identifier code (`id_code`), a data item type subcode (`id_sub_code`), an item identifier (`id`), and data packet. In addition to the general `id_code` codes, two code modifiers are defined: *groups* and *polys* as described below.

#### 5.3.1 General packets

data type	identifier	description
guint16	<code>id_code</code>	major class
guint8	<code>id_sub_code</code>	item subclass
guint16	<code>id</code>	id of item
	{data packet}	data item

A *general packet* consists of a single data packet. The `id_code` field identifies the class of packet (sensor, motor, actuator, orientation or misc) while the packet-specific `id_sub_code` field identifies the specific type of packet within that class. The `id` field identifies which particular item is associated with the data packet.

#### 5.3.2 Poly packet (RDI\_POLY bit)

data type	identifier	description
guint16	<code>id_code</code>	major class
guint8	<code>id_sub_code</code>	item subclass
guint8	<code>count</code>	number of items
guint16	<code>id</code>	id of first item
	{data packet}	data of first item
	...	
guint16	<code>id</code>	id of $n^{\text{th}}$ item
	{data packet}	data of $n^{\text{th}}$ item

A *poly packet* consists of one or more unrelated data packets. The `id_code` field identifies the class of packet (sensor, motor, actuator, orientation or misc) while the packet-specific `id_sub_code` field identifies the specific type of packet within that class. The `count` field gives the total number of data packets within the overall packet; it is always greater than 0. Each data packet is preceded by the `id` field of the particular item.

### 5.3.3 Group packet (RDI\_GROUP bit)

data type	identifier	description
guint16	id_code	major class
guint8	id_sub_code	item subclass
guint8	count	number of items
guint16	id	id of first item
	{data packet}	data of first item
	...	
	{data packet}	data of $n^{\text{th}}$ item

A *group packet* consists of one or more related data packets. The `id_code` field identifies the class of packet (sensor, motor, actuator, orientation or misc) while the packet-specific `id_sub_code` field identifies the specific type of packet within that class. The `count` field gives the total number of data packets within the overall packet; it is always greater than 0. A single `id` field is transmitted, representing the identifier of the first data packet. Each successive data packet is assumed to have a consecutive `id` field (i.e., the last data packet has identifier  $id + n - 1$ ).

## 5.4 Sensor (RDI-Sn) packets

Sensor data packets contains information from various sensors on the robot. The `id_sub_code` field identifies the specific type of sensor data contained in the data packet. There are currently four types of sensor data packets: *ultrasonic*, *relative encoder*, *absolute encoder*, and *accelerometer*.

### 5.4.1 Ultrasonic sensor (RDI\_SENSOR\_ULTRASONIC) packets

data type	identifier	description
guint16	id_code	RDI-Sn
guint8	id_sub_code	RDI_SENSOR_ULTRASONIC
guint16	id	id of item
guint16	delay	distance data (in microseconds)
statval	status	last time update

Ultrasonic sensor packets consist of a `delay` field containing the last time-of-flight reading from the sensor and a `status` field containing the relative time of the reading and the status of the sensor.

### 5.4.2 Relative encoder sensor (RDI\_SENSOR\_RELENCODER) packets

data type	identifier	description
guint16	id_code	RDI-Sn
guint8	id_sub_code	RDI_SENSOR_RELENCODER
guint16	id	id of item
gint16	d_count	position change (in counts)
statval	d_status	last update status
gint16	count	position (in counts)
statval	status	last time status

Relative encoder sensor packets consist of two `count` fields, containing the relative and accumulated encoder counts, and two `status` fields containing the relative times of each reading and the status of the sensor.

### 5.4.3 Absolute encoder sensor (RDI\_SENSOR\_ABSENCODER) packets

data type	identifier	description
guint16	id_code	RDI-Sn
guint8	id_sub_code	RDI_SENSOR_ABSENCODER
guint16	id	id of item
gint16	count	position (in counts)
statval	status	last time status

Absolute encoder sensor packets consist of a `count` field containing the accumulated encoder counts and a `status` field containing the relative time of the reading and the status of the sensor.

### 5.4.4 Accelerometer sensor (RDI\_SENSOR\_ACCEL) packets

data type	identifier	description
guint16	id_code	RDI-Sn
guint8	id_sub_code	RDI_SENSOR_ACCEL
guint16	id	id of item
float	accel	acceleration (in gravities)
statval	status	last time status

Accelerometer sensor packets consist of a `accel` field containing the last measured acceleration and a `status` field containing the relative time of the reading and the status of the sensor.

## 5.5 Motor (RDI-Mt) packets

Motor packets contains information from various motors on the robot. The `id_sub_code` field identifies the specific type of motor data contained in the packet. There is currently one type of motor packet: *DC motor*.

### 5.5.1 DC motor (RDI\_MOTOR\_DC) packets

data type	identifier	description
guint16	<code>id_code</code>	RDI-Mt
guint8	<code>id_sub_code</code>	RDI_MOTOR_DC
guint16	<code>id</code>	id of item
guint16	<code>speed</code>	speed/direction setting
statval	<code>status</code>	last time status

DC motor packets consist of a `speed` field (described below) a `status` field containing the relative time of the reading and the status of the motor.

Speed is specified either in terms of a motor duty cycle or an rotational speed given in fractional radians/s. The sixteen bits in the motor speed/direction field represent the following for a DC motor:

bit(s)	description
15	duty cycle (=1) or distance (=0)
14:12	reserved for future use
11	sign (direction)
10:0	magnitude (speed)

Note the sign and magnitude are independent (i.e., the representation is not two's complement but signed magnitude).

The range of the magnitude for each representation is as follows:

- duty cycle: 0% to 100% (0 to  $2^{11}-1$ )
- radians: 0 to 2.047 rad/s (0 to  $2^{11}-1$ )

## 5.6 Actuator (RDI-Ac) packets

Actuator packets contains information from various actuator on the robot. The `id_sub_code` field identifies the specific type of actuator data contained in the packet. There is currently two types of actuator data packets: *steering wheel* and *driving wheel*.

### 5.6.1 Steering wheel (RDI\_ACTUATOR\_STEER) packets

data type	identifier	description
guint16	<code>id_code</code>	RDI-Ac
guint8	<code>id_sub_code</code>	RDI_ACTUATOR_STEER
guint16	<code>id</code>	id of item
float	<code>angle</code>	wheel angle (in radians)
statval	<code>status</code>	last time status

Steering wheel actuator packets consist of an `angle` field containing the last measured wheel angle and a `status` field containing the relative time of the estimate and the status of the actuator.

### 5.6.2 Driving wheel (RDI\_ACTUATOR\_DRIVE) packets

data type	identifier	description
guint16	<code>id_code</code>	RDI-Ac
guint8	<code>id_sub_code</code>	RDI_ACTUATOR_DRIVE
guint16	<code>id</code>	id of item
float	<code>rotations</code>	wheel rotations
statval	<code>status</code>	last time status

Driving wheel actuator packets consist of a `rotation` field containing the last measured wheel rotation and a `status` field containing the relative time of the estimate and the status of the actuator.

## 5.7 Robot Orientation (RDI-RbOr) packets

Robot orientation data packets contains information about the location and orientation of the robot in its environment. `id_sub_code` field identifies the specific type of data contained in the data packet. There is two types of orientation data packets: *ego-centric* and *omni-centric*.

These values are estimated based upon sensor and motor values; they are not guaranteed to be accurate representations of reality.

### 5.7.1 Ego-centric orientation (RDI\_ORIENT\_EGO) packets

data type	identifier	description
guint16	id_code	RDI-RbOr
guint8	id_sub_code	RDI_ORIENT_EGO
float	xpos	estimated X position (in meters)
float	ypos	estimated Y position (in meters)
float	angle	estimated orientation (in radians)
statval	status	last time status

Ego-centric orientation packets provide estimates of the robot's orientation relative to an internal frame-of-reference. The `xpos` and `ypos` fields contain estimates of the robot's current position relative to some arbitrary starting position, and the `angle` field contains an estimate of the robot's current orientation relative to some arbitrary starting position. The `status` field containing the relative time of the last estimate.

### 5.7.2 Omni-centric orientation (RDI\_ORIENT\_OMNI) packets

data type	identifier	description
guint16	id_code	RDI-RbOr
guint8	id_sub_code	RDI_ORIENT_OMNI
float	direction	estimated direction (in radians)
float	speed	estimated velocity (in meters/s)
statval	status	last time status

Omni-centric orientation packets provide estimates of the robot's motion relative to an external frame-of-reference. The `direction` field contain an estimate of the robot's current direction of motion relative to its internal frame-of-reference and the `speed` field contain an estimate of the robot's current velocity relative to its internal frame-of-reference. The `status` field containing the relative time of the last estimate.

## 5.8 Miscellaneous (RDI-MS) packets

Miscellaneous packets contain any information about the system which does not neatly fit into another category. The `id_sub_code` field identifies the specific type of data contained in the packet. There is currently one type of miscellaneous data packets: *base time*.

### 5.8.1 Base Time data (RDI\_MISC\_BASETIME) packets

data type	identifier	description
guint16	id_code	RDI-MS
guint8	id_sub_code	RDI_MISC_BASETIME
timeval	status	absolute time

Base time data miscellaneous packets a `status` field which contains the robot's base time as described in Section 5.9.1.

## 5.9 Data type descriptions

The data types used in the specification are described below.

### 5.9.1 `timeval`

data type	description
<code>__time_t</code>	seconds time
<code>__time_t</code>	microseconds

`timeval` represents time in seconds and microseconds. The seconds, when used with Unix system functions such as `gettimeofday(2)`, represents elapsed seconds since 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC). It is defined in the header file `<sys/time.h>`.

### 5.9.2 `statval`

data type	description
typedef bit 0:0 <code>on</code>	on (=1) or off (=0)
typedef bit 1:1 <code>up</code>	up (=1) or down (=0)
typedef bit 2:2 <code>ok</code>	ok (=1) or error (=0)
typedef bit 3:7 <code>reserved</code>	reserved for future use
typedef bit 8:31 <code>rtimeval</code>	elapsed time (in milliseconds)

`statval` represents general status information about an object. It consists of a 8-bit status bitfield (defining individual operating characteristics) and a 24-bit elapsed time in milliseconds. The elapsed time used by all items is relative to the base time (see “Miscellaneous packets” in Section 5.8). As such, it can represent intervals up to approximately 4.6 hours in length.

### 5.9.3 General data types

data type	description
<code>float</code>	single-precision IEEE floating point (4 bytes)
<code>gint16</code>	signed short integer (2 bytes)
<code>guint16</code>	unsigned short integer (2 bytes)
<code>gint8</code>	signed short byte (1 byte)
<code>guint8</code>	unsigned short byte (1 byte)

All of these (except `float`) are defined in the Glib/GTK+ include file `<glib.h>`.

## 5.10 Constant definitions

### 5.10.1 Packet codes

Packet name	Acronym	value
Sensor	RDI-Sn	0x0010
Motor	RDI-Mt	0x0020
Actuator	RDI-Ac	0x0030
Robot orientation	RDI-RbOr	0x0040
Misc	RDI-Ms	0x0800
Poly Packet bit	RDI_POLY	0x4000
Group Packet bit	RDI_GROUP	0x8000

### 5.10.2 Status codes, Device subcodes, Device type command codes

Identifier	value
<i>Sensor subclass codes</i>	
RDI_SENSOR_ULTRASONIC	0x01
RDI_SENSOR_RELENCODER	0x02
RDI_SENSOR_ABSENCODER	0x03
RDI_SENSOR_ACCEL	0x04
<i>Motor subclass codes</i>	
RDI_MOTOR_DC	0x01
<i>Actuator subclass codes</i>	
RDI_ACTUATOR_STEER	0x01
RDI_ACTUATOR_DRIVE	0x02
<i>Robot Orientation subclass codes</i>	
RDI_ORIENT_EGO	0x01
RDI_ORIENT_OMNI	0x02
<i>Miscellaneous subclass codes</i>	
RDI_MISC_BASETIME	0x01



# Index

- RCI-CACMt, *see* RCI specifications, packets
- RCI-CACSn, *see* RCI specifications, packets
- RCI-CBsTmIq, *see* RCI specifications, packets
- RCI-CBsTm, *see* RCI specifications, packets
- RCI-CCnAc, *see* RCI specifications, packets
- RCI-CGtMtIq, *see* RCI specifications, packets
- RCI-CGtMtRp, *see* RCI specifications, packets
- RCI-CGtSnIq, *see* RCI specifications, packets
- RCI-CGtSnRp, *see* RCI specifications, packets
- RCI-CRs, *see* RCI specifications, packets
- RCI-CStIq, *see* RCI specifications, packets
- RCI-CStMt, *see* RCI specifications, packets
- RCI-CStRp, *see* RCI specifications, packets
- RCI-CStSn, *see* RCI specifications, packets
- RCI-CSvTm, *see* RCI specifications, packets
- RCI-ICaIq, *see* RCI specification, packets
- RCI-ICaRp, *see* RCI specification, packets
- RCI-IIId, *see* RCI specification, packets
- RCI-IIInCp, *see* RCI specification, packets
- RDI-Ac, *see* RDI specification, packets
- RDI-MS, *see* RDI specification, packets
- RDI-Mt, *see* RDI specification, packets
- RDI-RbOr, *see* RDI specification, packets
- RDI-Sn, *see* RDI specification, packets
- RDI\_ACTUATOR\_DRIVE, *see* RDI specification, packets
- RDI\_ACTUATOR\_STEER, *see* RDI specification, packets
- RDI\_GROUP, *see* RDI specification, packets
- RDI\_MISC\_BASETIME, *see* RDI specification, packets
- RDI\_MOTOR\_DC, *see* RDI specification, packets
- RDI\_ORIENT\_EGO, *see* RDI specification, packets
- RDI\_ORIENT\_OMNI, *see* RDI specification, packets
- RDI\_POLY, *see* RDI specification, packets
- RDI\_SENSOR\_ABSENCODER, *see* RDI specification, packets
- RDI\_SENSOR\_ACCEL, *see* RDI specification, packets
- RDI\_SENSOR\_RELENCODER, *see* RDI specification, packets
- RDI\_SENSOR\_ULTRASONIC, *see* RDI specification, packets
- timeval, 54
- RobotTime, 14
- bsal-cp, 7
- child-routines.cc, 9
- clearEncoders(), 22
- client-app.cc, 22
- config.robot, 9
- connectToServer(), 15
- encoderData, 14
- getServerTermination(), 16
- getSteerMotor(), 19
- homeRobot(), 17
- mposit\_type, 14
- mposval, 42
- rbp.cc, 8
- rci-client.cc, 13
- rci-client example, 22
- rci-server.cc, 9
- readEncoders(), 22
- readUltrasonics(), 20
- resetRobot(), 16

- robot-shm.cc, 11
- robot-time.hh, 15
- robotconfig.cc, 9
- server-interface.cc, 8
- setDriveMotor(), 18
- setSteerMotor(), 17
- shmqueue.cc, 11
- speedval , 43
- startEncoders(), 21
- startUltrasonics(), 20
- statval , 43
- stopEncoders(), 21
- timestuff.cc, 10
- timeval, 15, 42
- ultraData, 14
  
- statval , 54
  
- BRAK Software Abstraction Layer
  - explanation, 1
- BSAL, *see* BRAK Software Abstraction Layer
- BSAL, data types, 14
  
- CA, *see* Control Application
- Control application
  - description, 2
- Control process
  - board objects, 9
  - configuration database, 9
  - control infrastructure, 8
  - description, 2
  - device objects, 9
  - overview, 8
  - process and signal control, 9
  - RCI handler, 9
  - real-time support, 10
  - RSI handler, 11
- CP, *see* Control process
  
- Device and board objects, overview, 10
  
- GLib, data types, 13
- GP, *see* GUI process
- GUI process
  - description, 3
  
- Interface protocols, 4
  
- RCI, *see* Robot Control Interface
- RCI specification
  - constant definitions, 44
  - device subcodes, 45
  - device type command codes, 45
  - general packet format, 28
  - packets
    - RCI-CAcMt (Activate Motor), 35
    - RCI-CAcSn (Activate Sensor), 38
    - RCI-CBsTmIq (Base Time Inquiry), 41
    - RCI-CBsTm (Base Time), 41
    - RCI-CCnAc (Control Acknowledge), 34
    - RCI-CGtMtIq (Get Motor Inquiry), 37
    - RCI-CGtMtRp (Get Motor Reply), 37
    - RCI-CGtSnIq (Get Sensor Inquiry), 39
    - RCI-CGtSnRp (Get Sensor Reply), 40
    - RCI-CRs (Reset), 35
    - RCI-CStIq (Status Inquiry), 33
    - RCI-CStMt (Set Motor), 36
    - RCI-CStRp (Status Reply), 33
    - RCI-CStSn (Set Sensor), 39
    - RCI-CSvTm (Server Termination), 34
    - RCI-ICaIq (Capability Inquiry), 31
    - RCI-ICaRp (Capability Reply), 31
    - RCI-IIId (Identification), 30
    - RCI-IIInCp (Initialization Completed), 31
    - group format, 29
    - poly format, 28
    - status codes, 45
- RDI, *see* Robot Display Interface
- RDI specification
  - constant definitions, 55
  - device subcodes, 55
  - device type command codes, 55
  - packets
    - RDI-Ac (actuator), 51
    - RDI-Ms (miscellaneous), 53
    - RDI-Mt (motors), 51
    - RDI-RbOr (robot orientation), 52
    - RDI-Sn (sensors), 49

- RDI\_ACTUATOR\_DRIVE (driving wheel actuator), 52
- RDI\_ACTUATOR\_STEER (steering wheel actuator), 52
- RDI\_GROUP (group format), 49
- RDI\_MISC\_BASETIME (base time data), 53
- RDI\_MOTOR\_DC (DC motor), 51
- RDI\_ORIENT\_EGO (ego-centric orientation), 53
- RDI\_ORIENT\_OMNI (omni-centric orientation), 53
- RDI\_POLY (poly format), 48
- RDI\_SENSOR\_ABSENCODER (absolute encoder sensor), 50
- RDI\_SENSOR\_ACCEL (accelerometer sensor), 50
- RDI\_SENSOR\_RELENCODER (relative encoder sensor ), 50
- RDI\_SENSOR\_ULTRASONIC (ultrasonic sensor ), 49
  - general format, 48
  - status codes, 55
- Robot Control Interface
  - description, 4
- Robot Display Interface
  - description, 5
- Robot Status Interface
  - description, 4
- RSI, *see* Robot Status Interface
  
- SCC, *see* Simulator control client
- SCI, *see* Simulator Control Interface
- Simulator control client
  - description, 3
- Simulator Control Interface
  - description, 5
- Simulator Status Interface
  - description, 5