



a  
**Development Environment**  
for  
**Distributed Behavior-based Control**

Barry Brian Werger  
October, 1998



**ULLANTA** Performance Robotics

<b>1.</b>	<b>INTRODUCTION TO AYLLU</b>	<b>4</b>
<b>1.1</b>	<b>WHAT IS AYLLU?</b>	<b>4</b>
1.1.1	THE AYLLU “LANGUAGE”?	4
1.1.2	WHENCE THE NAME “AYLLU”?	4
1.1.2.1	What is Quechua?	4
<b>1.2</b>	<b>HOW IS AYLLU USED?</b>	<b>5</b>
<b>1.3</b>	<b>WHAT IS AN AYLLU SYSTEM?</b>	<b>5</b>
<b>1.4</b>	<b>WHAT IS AN AYLLU BEHAVIOR?</b>	<b>5</b>
<b>1.5</b>	<b>HOW ARE AYLLU BEHAVIORS CONNECTED?</b>	<b>5</b>
<b>1.6</b>	<b>SOME TERMINOLOGY</b>	<b>6</b>
<b>1.7</b>	<b>HOW DOES AYLLU PROVIDE FOR PIONEER CONTROL?</b>	<b>6</b>
<b>2.</b>	<b>THE AYLLU LIBRARY</b>	<b>8</b>
<b>2.1</b>	<b>BEHAVIOR DEFINITION</b>	<b>8</b>
<b>2.2</b>	<b>BEHAVIOR INSTANTIATION</b>	<b>8</b>
<b>2.3</b>	<b>PROCESS DEFINITION</b>	<b>9</b>
<b>2.4</b>	<b>DECLARATIONS</b>	<b>9</b>
2.4.1	PORTS	9
2.4.1.1	Normal Ports	10
2.4.1.2	Summation Ports	10
2.4.1.3	Extrema Ports	10
2.4.1.4	Priority Ports	10
2.4.2	SLOTS	11
2.4.3	MONOSTABLES	11
2.4.4	LOCAL DECLARATIONS	11
<b>2.5</b>	<b>PORT FUNCTIONS</b>	<b>11</b>
2.5.1	IMPORTANT NOTE ABOUT MESSAGE UNRELIABILITY	12
2.5.2	READING LOCAL PORTS	12
2.5.3	DATA AVAILABILITY	12
2.5.4	PORT PRIORITY	12
2.5.5	WRITING TO LOCAL PORTS	12
2.5.6	WRITING TO LOCAL PORTS WITH PRIORITY	13
2.5.7	COMMUNICATING DIRECTLY WITH PEER AND REMOTE PORTS	13
2.5.7.1	Reading Peer and Remote Ports	13
2.5.7.2	Writing to Peer and Remote Ports	13
2.5.8	CONNECTING PORTS	14
2.5.8.1	General Connection	14
2.5.8.2	Normal Connections	14
2.5.8.3	Suppressing/Inhibiting Connections	14
2.5.8.4	Overriding Connections	15
<b>2.6</b>	<b>SLOT FUNCTIONS</b>	<b>15</b>
2.6.1	WRITING TO SLOTS	15
2.6.2	READING SLOTS	15

<b>2.7</b>	<b>MONOSTABLE FUNCTIONS</b>	<b>15</b>
<b>2.8</b>	<b>SCHEDULER CONTROL FUNCTIONS</b>	<b>16</b>
<b>2.9</b>	<b>PIONEER CONTROL SETUP FUNCTIONS</b>	<b>16</b>
<b>3.</b>	<b>AYLLU BEHAVIORS FOR PIONEER CONTROL</b>	<b>17</b>
<b>3.1</b>	<b>PIONEER BASE CONTROLLER</b>	<b>17</b>
<b>3.2</b>	<b>SERVER PACKET PARSER</b>	<b>18</b>
<b>3.3</b>	<b>VISION PACKET PARSER</b>	<b>19</b>
<b>3.4</b>	<b>PAN-TILT-ZOOM CAMERA CONTROLLER</b>	<b>20</b>
<b>3.5</b>	<b>LOW-LEVEL COMMUNICATION</b>	<b>21</b>
3.5.1	PIONEER ROBOT STARTER	21
3.5.2	PIONEER PACKET RECEIVER	21
3.5.3	PIONEER PACKET SENDER	21
<b>4.</b>	<b>EXAMPLE: AN AYLLU SOCCER SYSTEM</b>	<b>23</b>
<b>5.</b>	<b>INDEX</b>	<b>29</b>

# 1. Introduction to Ayllu

## 1.1 What is Ayllu?

Ayllu is a tool for development of behavior-based control systems for intelligent mobile robots. It extends subsumption-style message passing to the multi-robot domain, provides for a wide variety of behavior-arbitration techniques, and allows a great deal of run-time system flexibility including dynamic reconfiguration of behavior structure and redistribution of tasks across a group of robots as determined by either task constraints or changing availability of resources.

Ayllu has a standard set of basic motor-control and sensor-interpretation behaviors for the Pioneer Mobile Robot. These include a base controller that takes advantage of direct wheel-speed control to provide highly responsive, arc-based, velocity-based (rotational and translational), and distance-based motor commands; as well as support for all Pioneer accessories including control of vision system modes and training. Due to the nature of Ayllu as an extendable collection of behaviors, all new versions will be able to remain strictly upward compatible with earlier versions so that upgrades will be painless.

Ayllu facilitates implementation of both reactive and deliberative systems, and provides the means for coordinating processor-intensive tasks, such as high-level planning and vision processing, with responsive low-level control. It supports and encourages fast data-driven control strategies.

Ayllu provides for rapid code development and effective code re-use. Behaviors can be multiply-instantiated and interact through abstract "ports", which can be dynamically connected to other ports at run-time. Behaviors can be arbitrarily distributed across hosts without code changes.

Ayllu is available on a wide variety of platforms including Unix, Macintosh, Windows, and QNX, and allows heterogeneous hosts to interact transparently.

### 1.1.1 The Ayllu "language"?

The question arises as to whether Ayllu is a language or a library. Strictly speaking, Ayllu may be referred to as a language, since it involves syntactic structures foreign to C itself, and some differences in semantics. Ayllu makes extensive use of C's macro facilities, and as a result the standard C preprocessor is able to translate all Ayllu code into standard C code. Since Ayllu therefore consists of header files and object code that uses standard C compilation, we refer to it in general as a library, or development environment, rather than a language.

### 1.1.2 Whence the name "Ayllu"?

*Ayllu* is a Quechua-language term that refers to a close-knit community, usually but not exclusively of kin, which engages in many mutual and reciprocal activities.

#### 1.1.2.1 What is Quechua?

Quechua is a native language of the Andean region, spoken by approximately 13 million people in Bolivia, Perú, Ecuador, Chile, Colombia, and Argentina. For information on the Quechua language, check Ullanta Performance Robotics' Quechua Homepage at <http://www-robotics.usc.edu/~barry/Quechua>. Yes, *Ullanta* comes from Quechua, too.

## 1.2 How is Ayllu Used?

You will notice that Ayllu for Pioneer provides no functions for control of the Pioneer robot - that is none that provide sensor information or cause motor activity. Instead, a comprehensive set of standard Pioneer control behaviors (see Section 1.7) are provided. The process of building an Ayllu system is more one of directing information between simple components than one of algorithmic design. For example, effective collision avoidance can be achieved by a behavior that merely scales its input, if its input is connected to sonar-range ports and its output is connected to the velocity port of the base-controller behavior. Another instance of the scaling behavior, connected to a visual information port for input and the rotational velocity port of the motor controller, can cause the robot to rotate to face an object perceived by the Fast-Trak Vision System. If both of these behaviors are active at the same time, the robot will display a very robust behavior of following objects while avoiding obstacles. The extended example in Section 4 demonstrates how such simple building blocks, combined with more complex higher-level ones, can generate complex behavior with concise code.

## 1.3 What is an Ayllu system?

An Ayllu system consists of a group of **behaviors** which run in parallel, and a set of **connections** through which messages are passed between behaviors. The behaviors may all be instantiated on one host computer, or may be spread across a network. Thus a single computer might control some number of robots, a number of computers may control a single robot, or a group of robots could be controlled in a fully distributed manner.

## 1.4 What is an Ayllu behavior?

An Ayllu behavior is an encapsulated group of lightweight processes that share an interface. The interface consists of **ports**, **slots**, and **monostables**.

- **Ports** and **slots** are registers that hold a single value of a specific type - integer, floating point, or character array (which may be interpreted as an arbitrary data structure). **Slots** are accessible only to processes in the behavior, while **ports** can be connected to other behaviors. When a value is written to a **port**, it is propagated along all outgoing connections from that **port**, subject to connection rules below.
- **Monostables** are Boolean values that have an associated period. When a **monostable** is "triggered," it remains true for the specified period of time, then reverts to a false. **monostables** are accessible only to processes within their behavior.

Behaviors all have a special **port** which holds the current activation level, and can be used to either switch on and off or prioritize a behavior as a whole.

## 1.5 How are Ayllu behaviors connected?

Ports are connected dynamically at run-time, and such connections may be to peer behaviors (on the same host computer) or to remote behaviors (across a network). Connections are unidirectional, though ports may have both incoming and outgoing connections in any number. Any messages written to a port, whether from a local process inside the behavior or from an incoming connection, are immediately propagated along all outgoing connections, subject to the following conditions:

- An incoming connection may be **inhibitory**. This means that, for some period of time after receiving a message along this connection, the **port** will not accept any other incoming messages.

- An incoming message may **suppress** a port. This means that, for some period of time after receiving a message along this connection, the port will not propagate any messages along outgoing connections.
- Ports may be selective about the messages they receive and propagate - they can be set to accept prioritized messages, or extreme (maximum or minimum) messages.
- Ports may sum messages rather than replace their current values - in this case, as a new message is received, the new sum is propagated.
- 

Please also read carefully the note about message unreliability in Section 2.5.1.

## 1.6 Some terminology

Ayllu's multiple levels of "locality" have the potential to confound discussion. We therefore define and adhere to the following standard terminology (it may seem belabored, but we believe it can only help):

- A **host** is a physical computer. An **Ayllu host** is a host running an Ayllu system. In a multiple-host system, each host must have a unique IP address.
- Behaviors running on the same host are referred to as **peer** behaviors. Behaviors running on separate hosts are described as **remote** with respect to each other.
- Ports, slots, and monostables are **local** to the behavior in which they are defined. Specifically, they are **local** to all procedures included in this behavior.
- Ports on **peer** behaviors are referred to as **peer ports**, and connections between them are referred to as **peer connections**. Ports on **remote** behaviors are **remote** ports, connected through **remote connections**.
- Every connection is unidirectional, having a **source** and a **destination**. With respect to a given port *P*, **incoming connections** are connections that specify *P* as the destination, and **outgoing connections** specify *P* as the source.
- In the discussion of the Ayllu library, the following terms are used to describe parameters: *name* is any legal C identifier; *localport*, *localslot*, and *localmono* are identifiers that have been used in appropriate declarations of the current behavior; *beh*, *srcbeh*, and *destbeh* are identifiers that have been used to name behavior instances; *port*, *srcport*, and *destport* are identifiers that have been used in declarations of the appropriate behavior; *host*, *srchost*, and *desthost* are strings (**char \***) that specify IP addresses - numerical or symbolic - of Ayllu hosts.

## 1.7 How does Ayllu provide for Pioneer control?

The following Standard Behaviors for Pioneer Control, described in detail in Section 3, are active by default on all Pioneer Ayllu hosts. While they may be multiply instantiated on a single host for control of more than one robot, we list them here by the default instance name.

- **SPP** - The Server Packet Parser provides ports that propagate sonar ranges, dead-reckoning information, robot status information, compass readings, and digital I/O and gripper information.
- **PBC** - The Pioneer Base Controller accepts messages that direct motor control. The PBC functions in two modes, one of which is Fast (direct-wheelspeed control), and the other is Precise (Saphira-style PSOS-mediated control). In either mode, ports accept the following commands: translational velocity, angular velocity, arc radius (maintained across varying velocities), change heading by degrees, translate by millimeters, stop. The PBC also processes gripper and Expansion Module commands.
- **VPP** - The Vision Packet Parser provides blob and line information from the three vision channels of the FastTrack Vision System, and accepts commands to change modes. It also parses "visual sonar" and frame-grab packets.

- **PRS** - The Robot Starter opens and maintains a serial connection with the Pioneer. In the case of a reset, the PRS will re-connect to the robot.
- **PTZ** - The Pan-Tilt-Zoom Camera Controller maintains the state of the PTZ camera, if present. It provides current angles and magnifications, and assures that the camera is pointing in the desired direction (that is, recovers from rough-terrain or motion jostles).
- **PPR** - The PSOS Packet Receiver receives packets from the serial line and sends them to the proper parser (SPP, VPP, or user-supplied).
- **PPS** - The PSOS Packet Sender prioritizes, packages, and sends PSOS packets down the serial line to the robot. It has three ports for different types of "volatile" commands, and a queue for commands that must go through.

## 2. The Ayllu Library

As discussed in Section 1.1.1, Ayllu is something between a language and a library. The following sections describe not only new functions, but new syntax and semantics that augment those of standard C. Specifically, **ayDefBehaviorClass** (and its required **ayINTERFACE** and **ayPROCESSES** sections), **ayDefProcess**, and all of the declarations (Section 2.4) are syntactically novel; and the “scoping” that allows multiple-instantiation of behaviors and clean behavior interfaces is very different from anything found in C, especially the semantics of local ports, slots, and monostables.

### 2.1 Behavior Definition

---

**ayDefBehaviorClass**(*name*)  
**ayINTERFACE**  
**ayPROCESSES**

---

Ayllu behavior definitions do not have counterparts in C, and thus have a unique syntax. Behaviors are defined as *behavior classes*, and may have multiple instantiations. The exact form for a behavior definition is:

```
ayDefBehaviorClass(name)
{
    ayINTERFACE {
        port declarations
        slot declarations
        monostable declarations
    }
    ayPROCESSES {
        process initializations
    }
}
```

*Name* must be a unique identifier. Both **ayINTERFACE** and **ayPROCESSES** must appear, even if the brackets must be empty in behaviors without processes or declarations. Port, slot, and monostable declarations are described below in Section 2.4. Each process initialization takes the following form:

---

**ayInitProcess**(*procname*, *rate*)

---

where *procname* is a name given in a process definition (see 2.3), and *rate* is a call to either **ratepersecond**(*freq*), which causes the process to run at the specified frequency, in Hz, or **delayseconds**(*secs*), which causes the process to run every *secs* seconds.

Behavior classes must be defined before they are referred to in code, and are thus subject to C’s prototyping requirements. Behavior class definitions can be prototyped in the form:

```
ayDefBehaviorClass(name);
```

See Section 4 for full examples of behavior definitions.

### 2.2 Behavior Instantiation

---

**ayInitBehavior**(*behclass*, *name*)

---

This function creates an instance of the class specified by *behclass*, which must be the name given in an **ayDefBehaviorClass**. The instance is subsequently referred to by *name*, which must be unique among behavior instances. A behavior class may be instantiated any number of times if the included processes follow the guidelines presented in Section 2.3.

## 2.3 Process Definition

---

**ayDefProcess**(*name*)

---

An Ayllu process definition is similar to a C function definition. It takes the form:

```
ayDefProcess(name)
{
    local port, slot, and monostable declarations
    variable declarations

    statements
}
```

Local port, slot, and monostable declarations are discussed below in Section 2.4.4; each such element declared must be matched by a declaration in the **ayINTERFACE** section of any behavior that instantiates the process. Variable definitions cannot be **static**. Any statement allowed in a function may be used in a process, including **return** to end the current run of the process, but the constraints of a *lightweight process* described below limit what statements may be practical.

Ayllu processes are both *lightweight* and *reentrant*, which requires that they be defined with certain restrictions.

Being *lightweight* means that the process is never interrupted; it runs to completion every time. Thus the process must be guaranteed to run in a short period of time - the exact period of time depends on processor speed and other processes in the system; in general it is safest to make sure that all Ayllu processes can safely run within each 50 millisecond cycle given a typical system load. Functions such as `sleep()`, or any blocking I/O functions, for example, should be avoided, as should open-ended loops.

Being *reentrant* means that the process may be run in numerous contexts; in this case, it means that the process code may be run from within various behaviors. Local ports, slots, and monostables are defined in a behavioral context that allows them to retain the values appropriate to their calling behavior across runs of the process. It is important that variables not be declared **static** within processes, because the values would then be modifiable by all instances of the process. Local volatile variables are treated exactly as local variables in a function, not retaining any value between runs.

Processes must be defined before they are referred to in code, and are thus subject to C's prototyping requirements. Process definitions can be prototyped in the form:

```
ayDefProcess(name);
```

See Section 4 for full examples of process definitions.

## 2.4 Declarations

### 2.4.1 Ports

These port declarations are part of a behavior class definition (as discussed above), and appear in the **ayINTERFACE** section of the behavior.

### 2.4.1.1 Normal Ports

---

<b>ayIntPort</b> ( <i>name</i> , <i>value</i> )	<i>32-bit integer value</i>
<b>ayFloatPort</b> ( <i>name</i> , <i>value</i> )	<i>double floating point value</i>
<b>ayStringPort</b> ( <i>name</i> , <i>value</i> )	<i>string value - char * to region inside port</i>
<b>ayMemPort</b> ( <i>name</i> , <i>value</i> )	<i>unsigned char * to region inside the port</i>

---

where *name* is a string (unique within the behavior) and *value* is a value of the appropriate type, defines a port named *name* and initializes it to *value*. Values of StringPorts and MemPorts are always copied in their entirety (that is, the memory region is copied and passed, not just a pointer).

EXAMPLES:

```
ayIntPort(velocity, 0);
ayFloatPort(voltage, 0.0);
ayStringPort(host, "disco.usc.edu");
ayStringPort(position, "0 0 0");
```

### 2.4.1.2 Summation Ports

---

<b>ayIntSumPort</b> ( <i>name</i> )	<i>32-bit integer value</i>
<b>ayFloatSumPort</b> ( <i>name</i> )	<i>double floating point value</i>

---

where *name* is a string (unique within the behavior), defines a port named *name* and initializes it to 0. When values are written to the port from outside the behavior, they are added to the port's current value (instead of replacing it). Writes from within the behavior replace the value as usual... this allows "resetting" of the port.

EXAMPLES:

```
ayIntSumPort(PacketsReceived);
ayFloatSumPort(EnergyExpended);
```

### 2.4.1.3 Extrema Ports

---

<b>ayIntMaxPort</b> ( <i>name</i> , <i>val</i> )	<i>32-bit integer value</i>
<b>ayFloatMaxPort</b> ( <i>name</i> , <i>val</i> )	<i>double floating point value</i>
<b>ayIntMinPort</b> ( <i>name</i> , <i>val</i> )	<i>32-bit integer value</i>
<b>ayFloatMinPort</b> ( <i>name</i> , <i>val</i> )	<i>double floating point value</i>

---

where *name* is a string (unique within the behavior), defines a port named *name* and initializes it to *val*. When values are written to the port from outside the behavior, they replace the port's current value **only if** they are greater than (less than, in the case of **MinPorts**) the current value.. Writes from within the behavior always replace the value as usual, which allows "resetting" and "thresholding" of the port.

EXAMPLES:

```
ayIntMaxPort(Fastest);
ayFloatMaxPort(FarthestObject);
```

### 2.4.1.4 Priority Ports

---

**ayPrioritize**(*port declaration*)

---

Wrapping `ayPrioritize` around a port declaration causes the port to become a *priority port*. In addition to the port properties specified by the port declaration, the port maintains accepts and propagates only messages that have a priority greater than or equal to its current priority. Whenever such a message is received, its priority replaces the current priority. The port's priority may be reset from within the behavior through the use of `aySetPortPriority`, and priority of messages are set as described in Sections 2.5.4 and 2.5.6.

EXAMPLES:

```
ayPrioritize(ayIntPort(FastestID, 0));
ayPrioritize(ayStringPort(BestTarget, "red"));
```

## 2.4.2 Slots

These slot declarations are part of a behavior class definition (as discussed above, under `ayDefBehaviorClass`), and appear in the `ayINTERFACE` section of the behavior.

---

<code>ayIntSlot(name, value)</code>	<i>32-bit integer value</i>
<code>ayFloatSlot(name, value)</code>	<i>double floating point value</i>
<code>ayStringSlot(name, value)</code>	<i>string value - char * to region inside port</i>
<code>ayMemSlot(name, value)</code>	<i>unsigned char * to region inside the port</i>

---

where *name* is a string (unique within the behavior) and *value* is a value of the appropriate type, defines a slot named *name* and initializes it to *value*. Values of StringSlots and MemSlots are copied in their entirety (that is, the memory region is copied and passed, not just a pointer).

## 2.4.3 Monostables

The monostable declaration is part of a behavior class definition (as discussed above), and appears in the `ayINTERFACE` section of the behavior.

---

<code>ayMonostable(name, duration)</code>	<i>double floating point value</i>
---	------------------------------------

---

where *name* is a string (unique within the behavior), defines a monostable named *name*, which remains active (`ayMonoActive` returns a value of true (1)) for duration seconds after it has been triggered by `ayTrigger`, and inactive (`ayMonoActive` returns a value of false (0)) otherwise.

## 2.4.4 Local Declarations

These local declarations appear in process definitions (as discussed in Section 2.3). They are used in the same manner as variable types (that is, they are followed by a list of names). Each name following one of these declarations must match the name of a port, slot, or monostable declared in the `ayINTERFACE` section of the definition of any behavior that includes the process.

---

<code>ayLocalPort</code>	<i>port in the local behavior</i>
<code>ayLocalSlot</code>	<i>slot in the local behavior</i>
<code>ayLocalMono</code>	<i>monostable in the local behavior</i>

---

Examples:

```
ayLocalPort Speed, position, MinSonar;
ayLocalSlot MostRecentSpeed;
ayLocalMono LastReceived;
```

## 2.5 Port Functions

## 2.5.1 Important note about message unreliability

Ayllu message passing is by nature *unreliable*. This means that there is no guarantee that a message that is sent will be noticed by the port it is sent to. Any message may be overwritten by a later message before it is read, a port may be overridden, inhibited, or suppressed, and in the case of messages sent over a network, there may be timing issues or loss of communication. The Ayllu philosophy favors up-to-date information over complete information; in a robot control system it is undesirable to spend time insuring that outdated sensor readings or motor commands arrive. Behavior design should take this into consideration, using feedback loops rather than sending commands and assuming the result. As robots are physical systems subject to noise and uncertainty, this is an essential practice regardless of the programming system; Ayllu merely facilitates this, as opposed to most systems which tend to queue messages.

## 2.5.2 Reading Local Ports

---

<i>int</i>	<b>ayReadIntPort</b> ( <i>localport</i> )
<i>double</i>	<b>ayReadFloatPort</b> ( <i>localport</i> )
<i>char *</i>	<b>ayReadStrPort</b> ( <i>localport</i> )
<i>unsigned char *</i>	<b>ayReadMemPort</b> ( <i>localport</i> )

---

return the value of the specified local port; the type of the returned value is appropriate to the port. After **ayReadTypePort**, **ayPortWritten** returns false until the port is written again.

## 2.5.3 Data Availability

---

<i>int</i>	<b>ayPortWritten</b> ( <i>localport</i> )
------------	---

---

returns true (1) or false (0), depending on whether the specified port has been written to (by **ayWriteTypePort**, **aySendMessage**, or a connection) since the last time it was read by **ayReadTypePort**. Useful for event-driven programming (as a test to allow processes to run only when they have new data or when certain events occur).

## 2.5.4 Port Priority

---

<i>double</i>	<b>ayPortPriority</b> ( <i>localport</i> )
<i>void</i>	<b>aySetPortPriority</b> ( <i>localport</i> , <i>double priority</i> )

---

*Priority ports* (as discussed in Section 2.4.1.4) have an associated priority that is the maximum of the priorities of messages received. Only messages with priorities greater than or equal to the port's current priority are accepted and propagated.

The current priority level of a local port can be retrieved through a call to **ayPortPriority**. The priority can be set from within the behavior through **aySetPortPriority**, to allow thresholding or periodic maximums.

Outgoing messages are sent with the priority of the sending port. **aySetPortPriority** can be called before an **ayWriteTypePort** if the message needs a tailored priority, or one of the functions in Section 2.5.6 can be used.

## 2.5.5 Writing to Local Ports

---

<i>void</i>	<b>ayWriteIntPort</b> ( <i>localport</i> , <i>int value</i> )
-------------	---

---

---

<code>void</code>	<code>ayWriteFloatPort(localport, double value)</code>
<code>void</code>	<code>ayWriteStrPort(localport, char * value)</code>
<code>void</code>	<code>ayWriteMemPort(localport, unsigned char * value)</code>

---

Each of these functions writes the specified value to localport. The types of the port and *value* must agree. *Value* is then propagated to all ports to which there are outgoing connections (as created by `ayConnect`).

## 2.5.6 Writing to Local Ports with Priority

---

<code>void</code>	<code>ayWriteIntPriority(localport, int value, double priority)</code>
<code>void</code>	<code>ayWriteFloatPriority(localport, double value, double priority)</code>
<code>void</code>	<code>ayWriteStrPriority(localport, char * value, double priority)</code>
<code>void</code>	<code>ayWriteMemPriority(localport, unsigned char * value, double priority)</code>

---

Each of these functions is equivalent to a call to `aySetPortPriority` followed by a call to `ayWriteTypePort`. See 2.5.4 above for information on how priority is assigned to messages.

## 2.5.7 Communicating Directly with Peer and Remote Ports

The following functions should be avoided whenever possible, as they not only limit portability of code but are less efficient than **connections** between local and peer or remote ports. However, they can be extremely useful for diagnostic purposes, and for user interaction which runs as a separate thread within an Ayllu process.

### 2.5.7.1 Reading Peer and Remote Ports

---

<code>int</code>	<code>ayIntPortValue(beh, port)</code>
<code>double</code>	<code>ayFloatPortValue(beh, port)</code>
<code>char *</code>	<code>ayStrPortValue(beh, port)</code>
<code>int</code>	<code>ayRemoteIntValue(host, beh, port)</code>
<code>double</code>	<code>ayRemoteFloatValue(host, beh, port)</code>
<code>char *</code>	<code>ayRemoteStrValue(host, beh, port)</code>

---

like `ayReadTypePort`, but addresses a port in another behavior and does not affect its "Written" status.

### 2.5.7.2 Writing to Peer and Remote Ports

---

<code>void</code>	<code>aySendIntMessage(beh, port, int value)</code>
<code>void</code>	<code>aySendFloatMessage(beh, port, double value)</code>
<code>void</code>	<code>aySendStrMessage(beh, port, char * value)</code>
<code>void</code>	<code>aySendMemMessage(beh, port, unsigned char * value) void</code>
	<code>aySendRemoteInt(host, beh, port, int value)</code>
<code>void</code>	<code>aySendRemoteFloat(host, beh, port, double value)</code>
<code>void</code>	<code>aySendRemoteStr(host, beh, port, char * value)</code>
<code>void</code>	<code>aySendRemoteMem(host, beh, port, unsigned char * value)</code>

---

like `ayWriteTypePort`, but addresses a non-local port (that is, a port of the behavior named *beh*).

## 2.5.8 Connecting Ports

### 2.5.8.1 General Connection

---

<code>void</code>	<code>ayConnectPorts(source, dest)</code>
<code>void</code>	<code>ayConnectSpecial(source, dest, type, period)</code>

---

All of the connection functions which follow are implemented as macros which call one of these two functions; **ayConnectPorts** is used for normal connections, while **ayConnectSpecial** is used for all the other types described below, as indicated by the type argument which may be **ayOVERRIDEIN**, **ayOVERRIDEOUT**, **ayINHIBITOUT**, or **aySUPPRESSIN**. The arguments *source* and *dest* must be calls to either **ayFindRemotePort**(*host, behavior, port*) for remote behaviors or **ayFindPort**(*behavior, port*) for peer behaviors. *Period* should be a call to **seconds**(*s*), which accepts a floating-point argument.

### 2.5.8.2 Normal Connections

---

<code>void</code>	<code>ayConnect(srcbeh, srcport, destbeh, destport)</code>
<code>void</code>	<code>ayConnectToRemote(srcbeh, srcport, desthost,</code> <code>destbeh, destport)</code>
<code>void</code>	<code>ayConnectFromRemote(srchost, srcbeh, srcport,</code> <code>destbeh, destport)</code>

---

where all arguments are strings, sets up a connection that causes any message written to the source (either from a local **ayWriteTypePort** call or from an incoming connection) to be propagated to the destination, subject to overriding, suppression, and inhibition from other connections.

### 2.5.8.3 Suppressing/Inhibiting Connections

---

<code>void</code>	<code>aySuppressIn(srcbeh, srcport, destbeh, destport,</code> <code>period)</code>
<code>void</code>	<code>ayInhibitOut(srcbeh, srcport, destbeh, destport,</code> <code>period)</code>
<code>void</code>	<code>aySuppressRemoteIn(srcbeh, srcport, desthost,</code> <code>destbeh, destport, period)</code>
<code>void</code>	<code>ayRemoteSuppressIn(srcbeh, srcport, desthost,</code> <code>destbeh, destport, period)</code>
<code>void</code>	<code>ayInhibitRemoteOut(srcbeh, srcport, desthost,</code> <code>destbeh, destport, period)</code>
<code>void</code>	<code>ayRemoteInhibitOut(srchost, srcbeh, srcport,</code> <code>destbeh, destport, period)</code>

---

These connection types do not propagate any messages; instead, they affect propagation of other connections. When a message is written to the source of a **suppressing** connection, the destination port will be unable to receive any messages along incoming connections for the specified *period*. After the source of an **inhibiting** message is written, no messages will be propagated along outgoing connections of the destination port for the specified *period*. *Period* should be a call to **seconds**(*float s*).

## 2.5.8.4 Overriding Connections

---

<code>void</code>	<code>ayOverrideIn(srcbeh, srcport, destbeh, destport, period)</code>
<code>void</code>	<code>ayOverrideOut(srcbeh, srcport, destbeh, destport, period)</code>
<code>void</code>	<code>ayOverrideRemoteIn(srcbeh, srcport, desthost, destbeh, destport, period)</code>
<code>void</code>	<code>ayRemoteOverrideIn(srchost, srcbeh, srcport, destbeh, destport, period)</code>
<code>void</code>	<code>ayOverrideRemoteOut(srcbeh, srcport, desthost, destbeh, destport, period)</code>
<code>void</code>	<code>ayRemoteOverrideOut(srchost, srcbeh, srcport, destbeh, destport, period)</code>

---

These combine normal and suppressive/inhibitory messaging: `ayOverrideIn` causes a message to be sent to (and propagated from) the destination, followed by a *period* during which the destination will accept no other messages along incoming connections; `ayOverrideOut` causes a message to be sent to (and propagated from) the destination, followed by a *period* during which no messages will be propagated along outgoing connections from the destination.

## 2.6 Slot Functions

### 2.6.1 Writing to Slots

---

<code>int</code>	<code>ayReadIntSlot(localslot)</code>
<code>double</code>	<code>ayReadFloatSlot(localslot)</code>
<code>char *</code>	<code>ayReadStrSlot(localslot)</code>
<code>unsigned char *</code>	<code>ayReadMemSlot(localslot)</code>

---

return the value of the specified local slot; the type of the returned value is appropriate to the slot.

### 2.6.2 Reading Slots

---

<code>void</code>	<code>ayWriteIntSlot(localslot, int value)</code>
<code>void</code>	<code>ayWriteFloatSlot(localslot, double value)</code>
<code>void</code>	<code>ayWriteStrSlot(localslot, char * value)</code>
<code>void</code>	<code>ayWriteMemSlot(localslot, unsigned char * value)</code>

---

writes the specified value to *localslot*. The types of the slot and value must agree.

## 2.7 Monostable Functions

---

<code>int</code>	<code>ayMonoActive(monostable)</code>
------------------	---------------------------------------

---

returns the current activity state of the specified monostable: 1 if active, 0 if inactive.

---

<code>void</code>	<code>ayTrigger(monostable)</code>
-------------------	------------------------------------

---

causes monostable name to be active for the period of time specified by its declaration. Repeated triggerings result in the monostable being active until the specified duration after the most recent call to **ayTrigger**.

## 2.8 Scheduler Control Functions

---

<code>void</code>	<code>ayRunBehaviors()</code>	
<code>void</code>	<code>ayBackgroundRunBehaviors()</code>	
<code>void</code>	<code>ayStopBehaviors()</code>	
<code>void</code>	<code>ayStopBehAction()</code>	<i>USER DEFINED</i>

---

**ayRunBehaviors** causes the scheduler to begin running all processes included in all instantiated behaviors, at their specified rates (see Section 2.1). It returns only when a call to **ayStopBehaviors** is made from inside a behavior. **ayBackgroundRunBehaviors** also starts the scheduler, but returns immediately so that behaviors run in the background (this is not available on some systems). When the behaviors run in the background, the foreground process can communicate with them using the functions described in Section 2.5.7, and the scheduler can be shut down by a foreground call to **ayStopBehaviors**.

If the function **ayStopBehAction** is defined by the user, it will be called by **ayStopBehaviors** immediately after the behaviors have stopped running.

## 2.9 Pioneer Control Setup Functions

---

<code>void</code>	<code>ayInitPioControl(char * serialport)</code>
<code>void</code>	<code>ayInitNamedPioControl(name, char * serialport)</code>

---

**ayInitPioControl** instantiates the standard behaviors for Pioneer control (see Sections 1.7 and 3), connects them to each other as appropriate, and assigns a serial port for communication with the robot base. Thus, after this function is called, the behavior instances *PPR*, *SPP*, *VPP*, *PRS*, *PBC*, *PTZ*, and *PPS* are set to run and available for connections, and *PPS* and *PPR* will communicate with the robot base through *serialport*.

**ayInitNamedPioControl** performs the same function, but instantiates the standard behaviors with *name* prepended to the instance names - that is, the call

`ayInitNamedPioControl(Red, serialport)`

will result in behavior instances named *RedPPR*, *RedSPP*, *RedVPP*, *RedPRS*, *RedPBC*, *RedPTZ*, and *RedPPS*, with *RedPPS* and *RedPPR* set to communicate through *serialport*. This is intended for situations where a single Ayllu host will control multiple robots.

### 3. Ayllu Behaviors for Pioneer Control

This section details the standard behaviors for control of Pioneer mobile robots. We list here the name of the behavior class, the name of the default instance as created by `ayInitPioControl` (see Section 2.9), and all of their available ports with a short description of the messages they provide or accept.

#### 3.1 Pioneer Base Controller

Prototype: `ayDefBehaviorClass(ayiBaseControllerMaker);`

Standard instance name: `PBC`

<b>Ports Used for Input</b>	
<code>ayIntPort(ControlMode, ayFAST);</code>	Motor Controller mode - can be one of: <ul style="list-style-type: none"> <li>• <code>aySMOOTH</code> - for smoother motion</li> <li>• <code>ayFAST</code> - for more responsive control</li> </ul>
<code>ayIntPort(DefaultVel, 200);</code>	Set velocity to be used when executing distance-based commands (see Distance port)
<i>Rotational Control</i> - writing to any of these ports cancels the effect of previous writes to another.	
<code>ayIntPort(RelHeading, 0);</code>	Change robot's heading by specified number of degrees; positive is counterclockwise, negative clockwise
<code>ayIntPort(AbsHeading, 0);</code>	Rotate robot to specified heading, in degrees
<code>ayIntPort(ArcRadius, 0);</code>	Maintain an arc of the specified radius (in millimeters). As any velocity or distance commands are sent (to Velocity and Distance ports), this arc radius is maintained. If constant arc radius is maintained, the robot will circle. Positive radius indicates counterclockwise rotation, negative indicates clockwise.
<code>ayIntPort(AngularVel, 0);</code>	Rotate robot at specified number of degrees per second; positive is counterclockwise, negative clockwise
<i>Translational Control</i> - writing to either of these ports cancels the effect of previous writes to the other.	
<code>ayIntPort(Velocity, 0);</code>	Set robot's velocity to specified millimeters/second; positive is forward, negative is backward
<code>ayIntPort(Distance, 0);</code>	Move the robot the specified number of millimeters; positive is forward, negative is backward
<i>Wheel Velocity Control</i> - writing to these cancels both previous velocity and translation commands. If only one of these receives a message, the old value is used for the other wheel.	
<code>ayIntPort(LeftWheelVel, 0);</code>	Sets velocity to be traveled by left wheel, in mm/second; positive is forward, negative is backward.
<code>ayIntPort(RightWheelVel, 0);</code>	Sets velocity to be traveled by right wheel.
<code>ayIntPort(Stop, 0);</code>	Stops all translational, rotational, and/or wheel-velocity based motion.
<code>ayIntPort(Gripper, 0);</code>	Sets the state of the Pioneer's gripper. May be one of: <ul style="list-style-type: none"> <li>• <code>ayGRIPOPEN</code> - open at the bottom position</li> </ul>

- ayGRIPTOP - closed at the top position
- ayGRIPCARRY - closed at the carry position

## 3.2 Server Packet Parser

Prototype: `ayDefBehaviorClass(ayiSPParserMaker);`

Standard instance name: `SPP`

Ports Used for Input	
<i>Dead-Reckoning Position Information</i> - the origin of the coordinate system is the point at which the robot base was first connected to by the host computer, or the position of the most recent write to the ResetPosition port. A heading of 0 indicates that the robot is oriented with the positive x-axis; heading increases clockwise and decreases counterclockwise.	
<code>ayFloatPort(XPos,0);</code>	X-coordinate, in millimeters
<code>ayFloatPort(YPos,0);</code>	Y-coordinate, in millimeters
<code>ayFloatPort(Heading,0);</code>	Heading in degrees; increases counterclockwise.
<code>ayFloatPort(LWheelVel,0);</code>	Left wheel velocity, in millimeters/second
<code>ayFloatPort(RWheelVel,0);</code>	Right wheel velocity, in millimeters/second
<code>ayFloatPort(Voltage,0);</code>	Battery level, in volts
<code>ayIntPort(Stalls,0);</code>	Motor stalls
<code>ayFloatPort(PTU,0);</code>	PTU pulse width
<code>ayIntPort(Compass,0);</code>	Compass heading in degrees; increases counterclockwise
<code>ayIntPort(Sonar0,0);</code>	Sonar reading of pinger 0, in millimeters
<code>ayIntPort(Sonar1,0);</code>	Sonar reading of pinger 1, in millimeters
<code>ayIntPort(Sonar2,0);</code>	Sonar reading of pinger 2, in millimeters
<code>ayIntPort(Sonar3,0);</code>	Sonar reading of pinger 3, in millimeters
<code>ayIntPort(Sonar4,0);</code>	Sonar reading of pinger 4, in millimeters
<code>ayIntPort(Sonar5,0);</code>	Sonar reading of pinger 5, in millimeters
<code>ayIntPort(Sonar6,0);</code>	Sonar reading of pinger 6, in millimeters
<code>ayIntPort(Sonar7,0);</code>	Sonar reading of pinger 7, in millimeters; the standard Pioneer has only 6 pingers, but the user can add an eighth
<code>ayIntPort(InputTimer,0);</code>	Time measured by the Pioneer's input timer, in microseconds
<code>ayIntPort(AnalogIn,0);</code>	Level of the analog input
<code>ayIntPort(DigitalIn,0);</code>	State of the Pioneer's digital input pins
<code>ayIntPort(DigitalOut,0);</code>	State of the Pioneer's digital output pins
<code>ayIntPort(GripperBeams,0);</code>	State of the breakbeams in the optional gripper. One of: <ul style="list-style-type: none"> <li>• ayGRIPFRONTBEAM</li> <li>• ayGRIPREARBEAM or</li> <li>• ayGRIPBOTHBEAMS to indicate that an object is</li> </ul>

	obstructing one or both beams, or <ul style="list-style-type: none"> <li>• 0, if neither beam is obstructed</li> </ul>
<code>ayIntPort(GripperState, 0);</code>	Position of the optional gripper, one of: <ul style="list-style-type: none"> <li>• <code>ayGRIPATCARRY</code> - closed, at carry position</li> <li>• <code>ayGRIPATOPEN</code> - open, at bottom</li> <li>• <code>ayGRIPATTOP</code> - closed, at top</li> <li>• <code>ayGRIPMOVING</code> - between positions</li> </ul>
<code>ayIntPort(GripperContact, 0);</code>	Contact sensors of the optional gripper. One of: <ul style="list-style-type: none"> <li>• <code>ayGRIPCONTACT</code></li> <li>• 0</li> </ul>
<code>ayIntPort(XModButton, 0);</code>	State of the button on the optional expansion module. One of: <ul style="list-style-type: none"> <li>• <code>ayBUTTONPRESSED</code></li> <li>• 0</li> </ul>
<code>ayIntPort(XModSwitch, 0);</code>	Position of the switch on the optional expansion module. One of: <ul style="list-style-type: none"> <li>• <code>aySWITCHUP</code></li> <li>• <code>aySWITCHDOWN</code></li> </ul>
<code>ayStringPort(RobotName, "");</code>	Name of the Pioneer base
<code>ayStringPort(RobotClass, "");</code>	Class of the Pioneer base
<code>ayStringPort(RobotSubclass, "");</code>	Subclass of the Pioneer base
<b>Ports Used for Input</b>	
<code>ayIntPort(ResetPosition, 0);</code>	Resets the Pioneer's coordinate system so that the current position becomes (0, 0) with heading 0.

### 3.3 Vision Packet Parser

Prototype: `ayDefBehaviorClass(ayivPParserMaker);`

Standard instance name: VPP

<b>Ports Used for Output</b>	
General Line-Mode Information	
<code>ayIntPort(LineBottomRow, 0);</code>	
<code>ayIntPort(LineNumSlices, 0);</code>	
<code>ayIntPort(LineSliceSize, 0);</code>	
<code>ayIntPort(LineMinMass, 0);</code>	
<i>Channel Mode</i> - specifies the type of information provided by each channel. Can be set by writing to <code>SetChannelXMode</code> .	
<code>ayIntPort(ChannelAMode, 0);</code>	Mode of channel A; one of: <ul style="list-style-type: none"> <li>• <code>ayBLOBMODE</code> - only color-blob information is provided</li> <li>• <code>ayBBOXMODE</code> - color-blob and bounding-box information is provided</li> </ul>

	<ul style="list-style-type: none"> <li>ayLINEMODE - only line mode information is provided</li> </ul>
<code>ayIntPort(ChannelBMode, 0);</code>	Mode of channel B
<code>ayIntPort(ChannelCMode, 0);</code>	Mode of channel C
<p><i>Color-Blob Information</i> - there are three of each of these; one for each channel. Replace <i>X</i> with A, B, or C (for example, A_BlobX). This information is updated in ayBLOBMODE and ayBBOXMODE.</p>	
<code>ayIntPort(X_BlobArea, 0);</code>	Area of the largest blob detected by channel <i>X</i> , in pixels
<code>ayIntPort(X_BlobX, 0);</code>	Visual X-coordinate of center of largest blob (0-249)
<code>ayIntPort(X_BlobY, 0);</code>	Visual Y-coordinate of center of largest blob (0-249)
<p><i>Blob Bounding-Box Information</i> - returned only in ayBBOXMODE. Replace <i>X</i> with A, B, or C. These provide information on the bounding box of the largest blob detected by the channel.</p>	
<code>ayIntPort(X_BBoxX1, 0);</code>	X-coordinate of left side of bounding box
<code>ayIntPort(X_BBoxX2, 0);</code>	X-coordinate of right side of bounding box
<code>ayIntPort(X_BBoxY1, 0);</code>	Y-coordinate of top of bounding box
<code>ayIntPort(X_BBoxY2, 0);</code>	Y-coordinate of bottom of bounding box
<p><i>Line Mode Information</i> - returned only in ayLINEMODE. Replace <i>X</i> with A, B, or C. These provide line-mode information, intended for following along a line of the color the channel is trained for.</p>	
<code>ayIntPort(X_NearSlice, 0);</code>	The number of the nearest (lowest) slice in which the line can be seen.
<code>ayIntPort(X_NumSlices, 0);</code>	Reports the number of slices in which the line is seen.
<code>ayIntPort(X_XInNearSlice, 0);</code>	Provides the X-coordinate of the line as it goes through the nearest slice
<b>Ports Used for Input</b>	
<code>ayIntPort(SetChannelAMode, 0);</code>	Set the mode for channel A
<code>ayIntPort(SetChannelBMode, 0);</code>	Set the mode for channel B
<code>ayIntPort(SetChannelCMode, 0);</code>	Set the mode for channel C
<code>ayIntPort(SetLineBottomRow, 0);</code>	Set the lowest row in which the line should be detected
<code>ayIntPort(SetLineNumSlices, 0);</code>	Set the number of slices to divide the camera image
<code>ayIntPort(SetLineSliceSize, 0);</code>	Set the size of each slice (in pixel height)
<code>ayIntPort(SetLineMinMass, 0);</code>	Set the minimum number of pixels of the appropriate color that must appear in each slice for it to be considered a detection of the line.

### 3.4 Pan-Tilt-Zoom Camera Controller

Prototype: `ayDefBehaviorClass(ayIPTZCamController);`  
Standard instance name: **PTZ**

This behavior is used to control the Pioneer PTZ camera. This camera does not provide feedback on its position, so the information provided in the ports TiltAngle and PanAngle are not necessarily correct; they reflect the most recent position the camera was commanded to. The camera takes a few seconds to pan from 90 to -90 degrees, so if commands are sent more frequently than this there is a high likelihood that the reported angles will be garbage. The

PTZ behavior provides some protection from jostling of the camera; the camera is ordered to its position repeatedly so that if robot motion causes it to shift (as happens with the AT's on rough terrain), it will return.

The pan range is -90 to 90 degrees (-90 is right of the robot, 90 is left); the tilt range is -20 to 20 degrees (-20 is looking up, 20 down), and the zoom range is 0 - 1023 (0 is "wide", 1023 is "telephoto").

Ports Used for Input	
<code>ayIntPort(RelPan, 0);</code>	Change pan angle by specified degrees
<code>ayIntPort(RelTilt, 0);</code>	Change tilt angle by specified degrees
<code>ayIntPort(AbsPan, 0);</code>	Set camera pan angle to specified degrees
<code>ayIntPort(AbsTilt, 0);</code>	Set camera tilt angle to specified degrees
Ports Used for Input and Output	
<code>ayIntPort(Zoom, 1023);</code>	Set camera zoom magnification
Ports Used for Output	
<code>ayIntPort(TiltAngle, 0);</code>	Reports current tilt angle (see caveat above)
<code>ayIntPort(PanAngle, 0);</code>	Reprts current pan angle (see caveat above)

## 3.5 Low-Level Communication

### 3.5.1 Pioneer Robot Starter

Prototype: `ayDefBehaviorClass(ayiRobotStarter);`

Standard instance name: **PRS**

Ports Used for Output	
<code>ayIntPort(Running, 0);</code>	Indicates whether the robot base is connected and running (1) or not (0)

### 3.5.2 Pioneer Packet Receiver

Prototype: `ayDefBehaviorClass(ayiPacketReceiverMaker);`

Standard instance name: **PPR**

Ports Used for Output	
<code>ayMemPort(OtherPacket, "");</code>	If a packet of non-standard type is received, it is written to this port so that a user-defined parsing behavior can process it.

### 3.5.3 Pioneer Packet Sender

Prototype: `ayDefBehaviorClass(ayiPacketSenderMaker);`

Standard instance name: **PPS**

The Packet Sender performs prioritization of the packets coming in from the PBC, PTZ, and VPP. It is not recommended that the user send packets directly to the PPS unless these other behaviors are replaced.

<b>Ports Used for Input</b>	
<code>ayMemPort(RotPacket, "");</code>	Packets to be sent with rotation-packet priority
<code>ayMemPort(VelPacket, "");</code>	Packets to be sent with velocity-packet priority
<code>ayMemPort(VisPacket, "");</code>	packets to be sent with vision-packet priority

## 4. Example: An Ayllu Soccer System

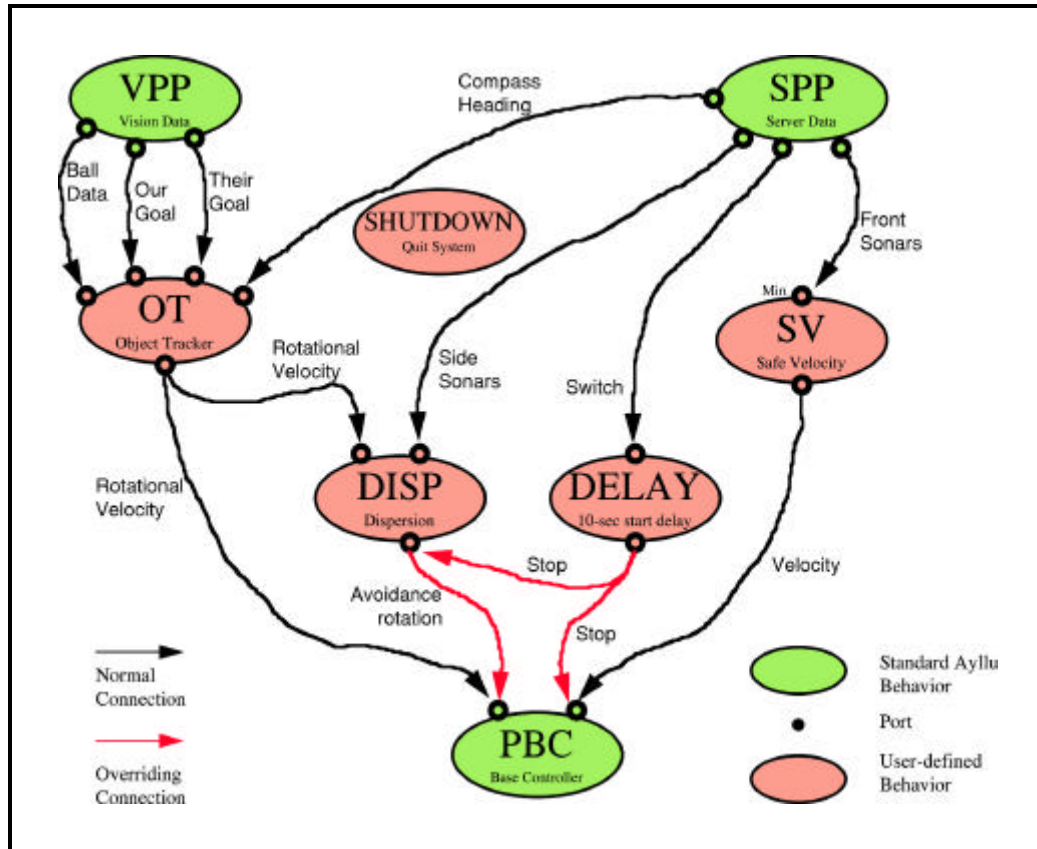


Diagram of the Soccer System

*/\*... Ayllu Example -*

*Code from "The Spirit of Bolivia"*

*A simple soccer system that displays sophisticated behavior, this code is similar to the system that was undefeated in both RoboCup '97 and '98. Rapid switching between basis behaviors leads to smooth, efficient trajectories for ball manipulation. Very safe and responsive, and thus good for play against children...*

*For details and papers published about this soccer approach, see: <http://www-robotics.usc.edu/~barry/ullanta>*

*/\* A number of constant definitions have been omitted... \*/*

`#include "ayllu.h"`

*/\*\* Basis Behavior functions - for ball manipulation \*\*\*/*

```

int Forward (int BallX, int Bally)          /* Go straight toward ball, */
{ return(-(BallX - 125) / 2); }            /* kicking it if close     */

int orbitCCW (int x, int y)                /* Orbit counter-clockwise */
{                                           /* around the ball         */
    if (Bally > 150)
        return (-BallX/2);
    else return (Forward(BallX, Bally));    /* if it's far, go right */
}                                           /* toward it instead      */

int orbitCW (int x, int y)                 /* Orbit clockwise        */
{                                           /* around the ball        */
    if (Bally > 150)
        return (BallX/2);
    else return (Forward(BallX, Bally));    /* if it's far, go right */
}                                           /* toward it instead      */

int Patrol(void)                            /* Cover an area of the field - works */
{ return PATROL_ROTVEL; }                  /* with sonar avoidance to "circle"  */

/*          **** BALL TRACKER behavior ****          */
/* Based on visual perception of ball and goals,    */
/* and compass if neither goal seen, decides which  */
/* of the Basis behaviors is appropriate to most   */
/* safely and efficiently line up to push the ball */
/* towards their (the opponent's) goal.          */

/**/ Object Tracking - behavior selecting process *//
ayDefProcess(objecttrack)
{
    ayLocalPort heading, Bally, BallX, BallSize, og, ogsize,
        tgs, tgleft, tgright, RVel;

    ayLocalSlot lastX;                      /* last location of ball if seen      */
                                           /* within last few seconds; used for */
    ayLocalMono recentBall;                 /* "persistent" perception of ball    */

    int head, tgl, tgr, ogc, ogs, tgs, ballx, bally, vel, close;

    head = ayReadIntPort(heading);          /* compass heading; 0=their goal */
                                           /* our goal center and size, in pixels */
    ogc = ayReadIntPort(og);                ogs = ayReadIntPort(ogsize);
                                           /* their goal left and right edges, in pixel x-coords */
    tgl = ayReadIntPort(tgleft); tgr = ayReadIntPort(tgright);

    if (ayReadIntPort(BallSize) > MIN_BALL_SIZE)
        /**/ We see the ball ***/
        {
            ayTrigger(recentBall); /* remember ball pos for a few seconds */
            ayWriteIntSlot(lastX, ayReadIntPort(BallX)); /* if we lose sight */
            ballx = ayReadIntSlot(lastX);
            bally = ayReadIntPort(Bally);
        }
}

```

```

    if (ayReadIntPort(tgsize) > 200)
    { /* we see their goal */
        if (ballx < tgl) vel = orbitCW(ball, bally);
        else if (ballx > tgr) vel = orbitCCW(ballx, bally);
        else vel = Forward(ballx, bally);
    }
    else if (ogs > 200)
    { /* we see our goal */
        if (ballx < ogc) vel = orbitCCW(ballx, bally);
        else vel = orbitCW(ballx, bally);
    }
    else if ((head > 345) || (head < 15)) /* we're facing their goal */
        vel = Forward(ballx, bally);
    else if (head > 179) /* stay between ball and our goal */
        vel = orbitCCW(ballx, bally);
    else vel = orbitCW(ballx, bally);
}
else /*** we don't see the ball ***/
    if ayMonoActive(recentBall) /* if we've seen ball recently */
    { if (ayReadIntSlot(lastX) < 125)
        vel = 75; /* turn towards where last seen */
        else vel = -75;
    }
    else vel = Patrol(); /* otherwise, patrol area */

ayWriteIntPort(RVel, vel);
}

/* Reporting Process */
ayDefProcess(TrackerReport) /* Print a brief report every second */
{ ayLocalPort BallX, Bally, RVel; /* about object-tracker activity */
  ayLocalSlot seconds;

  ayWriteIntSlot(seconds, ayReadIntSlot(seconds) + 1);

  printf("%i\n", ayReadIntSlot(seconds));
  printf("Ball coords: (%i,%i), rotation command: %i\n",
    ayReadIntPort(BallX), ayReadIntPort(Bally),
    ayReadIntPort(RVel));
}

ayDefBehaviorClass(ObjectTracker)
{
  ayINTERFACE {
    ayIntPort(heading, 0); /* Compass heading */
    /* Ball, their goal, and our goal - visual information */
    ayIntPort(BallX, 0); ayIntPort(Bally, 0); ayIntPort(BallSize, 0);
    ayIntPort(tgleft, 0); ayIntPort(tgright, 0); ayIntPort(tgsize, 0);
    ayIntPort(og, 0); ayIntPort(ogsize, 0);
    ayIntPort(RVel, 0); /* Output - rotational velocity */

    ayIntSlot(lastX, 0); /* For persistent perception, to */
    ayMonostable(recentBall, seconds(6)); /* overcome noisy vision */
  }
  ayPROCESSES {
    ayInitProcess(objecttrack, ratepersecond(20));
    ayInitProcess(TrackerReport, ratepersecond(1));
  }
}

```

```

/*          **** DISPERSION behavior ****          */
/* Causes robot to move away from objects on the sides when combined
   with safe velocity and ball-handling behaviors, leads to offensive
   and defensive formations. Overrides Ball Tracker when necessary -
   see connections in main()          */

ayDefProcess(sidewatcher)
{ ayLocalPort intendedRot, leftside, leftfront,
  rightside, rightfront, avoidRot;
  int intended, avoid = 0;

  intended = ayReadIntPort(intendedRot);
  if ((ayReadIntPort(leftside) < SIDEAVOIDTHRESH)
      || (ayReadIntPort(leftfront) < FRONTAVOIDTHRESH))
      avoid = -AVOIDROT;
  else
      if ((ayReadIntPort(rightside) < SIDEAVOIDTHRESH)
          || (ayReadIntPort(rightfront) < FRONTAVOIDTHRESH))
          avoid = AVOIDROT;

  if (avoid < 0)          /* send dispersion command onlu if necessary */
      { if (avoid < intended) ayWriteIntPort(avoidRot, avoid); }
      else if (avoid > 0)
          { if (avoid > intended) ayWriteIntPort(avoidRot, avoid); }
  }

ayDefBehaviorClass(Disperser)
{
  ayINTERFACE {
    ayIntPort(leftside, 0);    ayIntPort(rightside, 0);
    ayIntPort(leftfront, 0);  ayIntPort(rightfront, 0);
    ayIntPort(intendedRot, 0); ayIntPort(avoidRot, 0);
  }
  ayPROCESSES {
    ayInitProcess(sidewatcher, ratepersecond(20));
  }
}

/*          *** Delayed Start Behavior ***          */
/* RoboCup requires a 10-second start delay... this keeps robot
   from moving until ten seconds after switch is moved          */
ayDefProcess(delayafterswitch)
{ ayLocalPort Switch, StopCommand;
  ayLocalMono StartDelay;
  if ((ayReadIntPort(Switch) & ayXMODSWITCH) == aySWITCHDOWN)
      ayTrigger(StartDelay);
  if (ayMonoActive(StartDelay)) ayWriteIntPort(StopCommand, 0);
  return;
}

ayDefBehaviorClass(DelayedStarter)
{
  ayINTERFACE {
    ayIntPort(StopCommand, 0);
    ayIntPort(Switch, 0);
    ayMonostable(StartDelay, seconds(10));
  }
}

```

```

    ayPROCESSES {
        ayInitProcess(delayafterswitch, ratepersecond(20));
    }
}

/*          *** Safe Velocity Behavior ***          */
/* Sets the translational velocity of the robot to the maximum
   safe value - avoids collisions with walls and other robots */
ayDefProcess(sonarwatcher)
{ ayLocalPort MinSonar, VelCom;
  int min, speed;

  min = ayReadIntPort(MinSonar);
  speed = MAX_SPEED;

  if (min < DANGER_DISTANCE)
    speed = BACKUP_SPEED;
  else if (min < SAFETY_DISTANCE)
    speed = (min - DANGER_DISTANCE) * MAX_SPEED
            / (SAFETY_DISTANCE - DANGER_DISTANCE);

  ayWriteIntPort(VelCom, speed);
  ayWriteIntPort(MinSonar, MAX_SONAR_READING);
  return;
}

ayDefProcess(SafetyReport)
{ ayLocalPort MinSonar, VelCom;

  printf("Min Sonar is %i, Velocity Command is %i\n",
        ayReadIntPort(MinSonar), ayReadIntPort(VelCom));
  return;
}

ayDefBehaviorClass(SafeVelocity)
{
  ayINTERFACE {
    ayIntMinPort(MinSonar, MAX_SONAR_READING);
    ayIntPort(VelCom, 0);
  }
  ayPROCESSES {
    ayInitProcess(sonarwatcher, ratepersecond(4));
    ayInitProcess(SafetyReport, ratepersecond(1));
  }
}

/*          *** Shutdown Behavior ***          */
/* Turns off robot and quits program when a key is hit */
ayDefProcess(StopWhenKeyHit)
{ if (kbhit()) ayStopBehaviors();
  return;
}

ayDefBehaviorClass(Stopper)
{ ayINTERFACE { }
  ayPROCESSES { ayInitProcess(stopper, ratepersecond(20)); }
}

int main()

```

```

{
    ayInitPioControl("modem");

    ayInitBehavior(SafeVelocity, SV);          /* Instantiate behaviors */
    ayInitBehavior(Disperser, DISP);
    ayInitBehavior(DelayedStarter, DELAY);
    ayInitBehavior(ObjectTracker, OT);
    ayInitBehavior(Stopper, SHUTDOWN);

    ayConnect(SPP, Sonar1, SV, MinSonar); /* Connect forward-facing */
    ayConnect(SPP, Sonar2, SV, MinSonar); /* sonars to safe velocity */
    ayConnect(SPP, Sonar3, SV, MinSonar); /* behavior - MinSonar */
    ayConnect(SPP, Sonar4, SV, MinSonar); /* only accepts the minnum */
    ayConnect(SPP, Sonar5, SV, MinSonar); /* one */
    ayConnect(SV, VelCom, PBC, Velocity);

    ayConnect(SPP, Compass, OT, heading);
    ayConnect(VPP, A_BlobX, OT, BallX);
    ayConnect(VPP, A_BlobY, OT, BallY);
    ayConnect(VPP, A_BlobArea, OT, BallSize);
    ayConnect(VPP, B_BlobArea, OT, tgsiz); /* Goals are marked with */
    ayConnect(VPP, C_BlobArea, OT, ogsiz); /* colors - switch B and */
    ayConnect(VPP, B_BBoxX1, OT, tgleft); /* C here to switch side */
    ayConnect(VPP, B_BBoxX2, OT, tgright); /* of the field. */
    ayConnect(VPP, C_BlobX, OT, og);
    ayConnect(OT, RVel, PBC, AngularVel);
    ayConnect(OT, RVel, DISP, intendedRot);

    ayConnect(SPP, Sonar0, DISP, leftside); /* Side sonars are used */
    ayConnect(SPP, Sonar1, DISP, leftfront); /* for dispersion... */
    ayConnect(SPP, Sonar5, DISP, rightfront); /* DISP overrides the */
    ayConnect(SPP, Sonar6, DISP, rightside); /* Object Tracker when */
    ayOverrideInput(DISP, avoidRot, PBC, AngularVel, /* necessary. */
        seconds(0.1));

    ayConnect(SPP, DigitalIn, DELAY, Switch); /* 10-sec. */
    ayOverrideInput(DELAY, StopCommand, PBC, Velocity, /* delay on */
        seconds(0.1));
    ayOverrideOutput(DELAY, StopCommand, DISP, avoidRot, /* start. */
        seconds(0.1));

    aySendIntMessage(PBC, ControlMode, ayFAST); /* Set for most */
                                                /* responsive control */
    printf("Behaviors initialized and connected\n");

    ayRunBehaviors(); /* run behaviors until they decide to stop */

return 0;
}

```

## 5. Index

---

### A

AbsHeading  
  port (in PBC) á 17  
AbsPan  
  port (in PTZ) á 21  
AbsTilt  
  port (in PTZ) á 21  
activation á 5  
AnalogIn  
  port (in SPP) á 18  
AngularVel  
  port (in PBC) á 17  
ArcRadius  
  port (in PBC) á 17  
*ayDefBehaviorClass* á 8  
*ayFloatMaxPort* á 10  
*ayFloatMinPort* á 10  
*ayFloatPort* á 10  
*ayFloatSlot* á 11  
*ayFloatSumPort* á 10  
ayInitBehavior á 8  
*ayInitProcess* á 8  
ayINTERFACE á 8  
*ayIntMaxPort* á 10  
*ayIntMinPort* á 10  
*ayIntPort* á 10  
*ayIntSlot* á 11  
*ayIntSumPort* á 10  
Ayllu  
  as a language á 4, 8  
  What is Ayllu? á 4  
*ayLocalMono* á 11  
*ayLocalPort* á 11  
*ayLocalSlot* á 11  
*ayMemPort* á 10  
*ayMemSlot* á 11  
ayMonoActive á 11  
*ayMonostable* á 11  
*ayPortPriority* á 12  
*ayPortWritten* á 12  
ayPrioritize á 10  
ayPROCESSES á 8  
*ayReadFloatPort* á 12  
*ayReadIntPort* á 12  
*ayReadMemPort* á 12  
*ayReadStrPort* á 12  
*aySetPortPriority* á 12  
*ayStringPort* á 10  
*ayStringSlot* á 11

ayTrigger á 11  
*ayWriteFloatPort* á 12  
*ayWriteFloatPriority* á 13  
*ayWriteIntPort* á 12  
*ayWriteIntPriority* á 13  
*ayWriteMemPort* á 12  
*ayWriteMemPriority* á 13  
*ayWriteStrPort* á 12  
*ayWriteStrPriority* á 13

---

### B

*beh* á 6  
behaviors  
  definition á 8  
  instantiation á 8  
  peer á 6  
  remote á 6  
  what is an Ayllu behavior á 5

---

### C

ChannelAMode  
  port (in VPP) á 19  
Compass  
  port (in SPP) á 18  
connections  
  inhibitory á 5  
  peer á 6  
  remote á 6  
  suppressive á 5  
  what is a connection? á 5  
ControlMode  
  port (in PBC) á 17

---

### D

data availability á 12  
data-driven programming á 12  
DefaultVel  
  port (in PBC) á 17  
delayseconds á 8  
*destbeh* á 6  
*desthost* á 6  
DigitalIn  
  port (in SPP) á 18  
DigitalOut  
  port (in SPP) á 18  
Distance

port (in PBC) á 17

---

## **E**

event-driven programming á 12

extrema

ports á 10

---

## **G**

Gripper

port (in PBC) á 17

GripperBeams

port (in SPP) á 18

GripperContact

port (in SPP) á 19

GripperState

port (in SPP) á 19

---

## **H**

Heading

port (in SPP) á 18

host á 6

---

## **I**

**incoming connections** á 6

**inhibitory**

connections á 5

InputTimer

port (in SPP) á 18

---

## **L**

LeftWheelVel

port (in PBC) á 17

*lightweight process* á 9

LineBottomRow

port (in VPP) á 19

LineMinMass

port (in VPP) á 19

LineNumSlices

port (in VPP) á 19

LineSliceSize

port (in VPP) á 19

**local**

declarations á 11

monostables á 9, 11

ports á 9, 11

slots á 9, 11

use of term á 6

*localmono* á 6

*localport* á 6

*localslot* á 6

LWheelVel

port (in SPP) á 18

---

## **M**

message

prioritized á 6

messages

priority of á 12

propagation of á 5

unreliability of á 11

**monostables**

declaration á 11

local á 6, 9, 11

what is a monostable? á 5

---

## **N**

*name* á 6

---

## **O**

OtherPacket

port (in PPR) á 21

**outgoing connections** á 6

---

## **P**

PanAngle

port (in PTZ) á 21

Pan-Tilt-Zoom Camera á 20

PBC á 6, 29

Pioneer Base Controller á 6

**peer behavior** á 6

Pioneer robot

Standard Behaviors for á 6

port priority á 10

**ports**

data availability á 12

extrema á 6, 10

local á 6, 9, 11, 12

peer á 6

priority á 6, 12, 13

reading á 12

remote á 6

summation á 6, 10

what is a port? á 5

writing to á 12, 13

PPR

Pioneer Packet Receiver á 7, 14, 21

PPS

Pioneer Packet Sender á 7, 22

priority  
  of messages á 6, 12  
  of ports á 10, 12, 13  
priority ports á 10, 12  
*processes*  
  definition á 9  
  *instantiation* á 8  
  lightweight á 9  
  reentrant á 9  
propagation á 5  
**prototypes**  
  behavior á 8  
  process á 9  
PRS  
  Pioneer Robot Starter á 6, 21  
PTU  
  port (in SPP) á 18  
**PTZ**  
  **Pan-Tilt-Zoom Camera Controller** á 7, 22

---

## Q

Quechua language á 4

---

## R

*ratepersecond* á 8  
*reentrant process* á 9  
RelHeading  
  port (in PBC) á 17  
RelPan  
  port (in PTZ) á 21  
RelTilt  
  port (in PTZ) á 21  
**remote**  
  behavior á 6  
  port á 6  
**remote host** á 6  
ResetPosition  
  port (in SPP) á 19  
RobotClass  
  port (in SPP) á 19  
RobotName  
  port (in SPP)  
  port (in SPP) á 19  
RobotSubclass  
  port (in SPP) á 19  
RotPacket  
  port (in PPS) á 22  
Running  
  port (in PRS) á 21  
RWheelVel  
  port (in SPP) á 18

---

## S

scoping  
  differences from C á 8  
semantics  
  differences from C á 4, 8  
**slots**  
  declaration á 11  
  local á 6, 9, 11  
  what is a slot? á 5  
SonarN  
  port (in SPP) á 18  
SPP á 6, 7, 28, 29  
  Server Packet Parser á 6  
*srcbeh* á 6  
*srchost* á 6  
Stalls  
  port (in SPP) á 18  
Stop  
  port (in PBC) á 17  
summation  
  ports á 10  
  summation ports á 6  
**suppressive**  
  connections á 5  
syntax  
  differences from C á 4, 8  
system  
  definition of á 5

---

## T

terminology á 6  
TiltAngle  
  port (in PTZ) á 21  
triggering  
  of monostables á 5

---

## U

unreliability  
  of messages á 11

---

## V

Velocity  
  port (in PBC) á 17  
VelPacket  
  port (in PPS) á 22  
VisPacket  
  port (in PPS) á 22  
Voltage  
  port (in SPP) á 18  
VPP  
  Vision Packet Parser á 6, 7, 29

---

## **X**

XModButton  
port (in SPP) á 19  
XModSwitch  
port (in SPP) á 19  
XPos  
port (in SPP) á 18

---

## **Y**

YPos  
port (in SPP) á 18

---

## **Z**

Zoom  
port (in PTZ) á 21