



Distributed Behavior-Based Control
for
Pioneer Mobile Robots

Barry Brian Werger
January, 1999



| | | |
|------------|---|-----------|
| 1. | INTRODUCTION TO AYLLU | 7 |
| <hr/> | | |
| 1.1 | WHAT IS AYLLU? | 7 |
| 1.1.1 | THE AYLLU “LANGUAGE”? | 7 |
| 1.1.2 | WHENCE THE NAME “AYLLU”? | 8 |
| 1.1.2.1 | What is Quechua? | 8 |
| 1.2 | HOW IS AYLLU USED? | 8 |
| 1.3 | WHAT IS AN AYLLU SYSTEM? | 8 |
| 1.4 | WHAT IS AN AYLLU BEHAVIOR? | 8 |
| 1.5 | HOW ARE AYLLU BEHAVIORS CONNECTED? | 8 |
| 1.6 | SOME TERMINOLOGY | 9 |
| 1.7 | HOW DOES AYLLU PROVIDE FOR PIONEER CONTROL? | 9 |
| | | |
| 2. | A BRIEF TUTORIAL | 11 |
| <hr/> | | |
| 2.1 | VELOCITY CONTROL - PROGRAM STRAIGHTLINE.C | 11 |
| 2.1.1 | BEHAVIOR CLASS DEFINITION | 12 |
| 2.1.2 | PROCESS DEFINITION | 12 |
| 2.1.3 | BEHAVIOR INSTANTIATION | 12 |
| 2.1.4 | CONNECTIONS | 13 |
| 2.1.5 | INITIALIZING AYLLU AND RUNNING BEHAVIORS | 13 |
| 2.1.6 | PUTTING IT ALL TOGETHER | 13 |
| 2.1.7 | RESULTING BEHAVIOR | 15 |
| 2.2 | LIVELY MOTION AROUND AN AREA - PROGRAM WANDERER.C | 15 |
| 2.2.1 | BEHAVIOR DEFINITION | 15 |
| 2.2.2 | PROCESS DEFINITION | 16 |
| 2.2.3 | CONNECTIONS | 17 |
| 2.2.4 | RESULTING BEHAVIOR | 17 |
| 2.3 | MINIMAL COORDINATED ACTION - PROGRAM MODERNDANCE.C | 17 |
| 2.3.1 | CONNECTIONS | 18 |
| 2.3.2 | RESULTING BEHAVIOR | 19 |
| 2.4 | GOING TO A SPECIFIC POINT - PROGRAM GOTO.C | 19 |
| 2.4.1 | BEHAVIOR DEFINITION | 20 |
| 2.4.2 | PROCESS DEFINITION | 21 |
| 2.4.3 | CONNECTIONS | 21 |
| 2.4.4 | RESULTING BEHAVIOR | 21 |
| 2.5 | GOING TO A POINT WITH OBSTACLE AVOIDANCE - PROGRAM GETTO.C | 22 |
| 2.5.1 | BEHAVIOR DEFINITION | 22 |
| 2.5.2 | PROCESS DEFINITION | 23 |
| 2.5.3 | CONNECTIONS | 23 |
| 2.5.4 | RESULTING BEHAVIOR | 24 |
| 2.6 | TRAVELING IN FORMATION - PROGRAM MARCH.C | 24 |
| 2.6.1 | PUTTING IT ALL TOGETHER | 25 |
| 2.6.2 | RESULTING BEHAVIOR | 26 |
| 2.7 | FURTHER DEVELOPMENTS | 26 |

| | | |
|------------|---|-----------|
| 3. | AYLLU LANGUAGE EXTENSIONS | 27 |
| 3.1 | BEHAVIOR DEFINITION | 27 |
| 3.2 | BEHAVIOR INSTANTIATION | 27 |
| 3.3 | PROCESS DEFINITION | 28 |
| 3.4 | DECLARATIONS | 28 |
| 3.4.1 | PORTS | 28 |
| 3.4.1.1 | Normal Ports | 29 |
| 3.4.1.2 | Summation Ports | 29 |
| 3.4.1.3 | Extrema Ports | 29 |
| 3.4.1.4 | Priority Ports | 29 |
| 3.4.2 | SLOTS | 30 |
| 3.4.3 | MONOSTABLES | 30 |
| 3.4.4 | LOCAL DECLARATIONS | 30 |
| 3.4.5 | HOST DECLARATIONS | 30 |
| 4. | THE AYLLU FUNCTION LIBRARY | 33 |
| 4.1 | PORT FUNCTIONS | 33 |
| 4.1.1 | IMPORTANT NOTE ABOUT MESSAGE UNRELIABILITY | 33 |
| 4.1.2 | READING LOCAL PORTS | 33 |
| 4.1.3 | DATA AVAILABILITY | 33 |
| 4.1.4 | PORT PRIORITY | 33 |
| 4.1.5 | WRITING TO LOCAL PORTS | 34 |
| 4.1.6 | WRITING TO LOCAL PORTS WITH PRIORITY | 34 |
| 4.1.7 | COMMUNICATING DIRECTLY WITH PEER AND REMOTE PORTS | 34 |
| 4.1.7.1 | Reading Peer Ports | 34 |
| 4.1.7.2 | Writing to Peer and Remote Ports | 34 |
| 4.1.8 | CONNECTING PORTS | 35 |
| 4.1.8.1 | Normal Connections | 35 |
| 4.1.8.2 | Suppressing/Inhibiting Connections | 35 |
| 4.1.8.3 | Overriding Connections | 35 |
| 4.2 | SLOT FUNCTIONS | 36 |
| 4.2.1 | WRITING TO SLOTS | 36 |
| 4.2.2 | READING SLOTS | 36 |
| 4.3 | MONOSTABLE FUNCTIONS | 36 |
| 4.4 | GENERAL PACKET QUEUE FUNCTIONS | 37 |
| 4.5 | SCHEDULER CONTROL FUNCTIONS | 37 |
| 4.6 | PIONEER CONTROL SETUP FUNCTION | 37 |
| 4.7 | REMOTE HOST FUNCTIONS | 37 |
| 4.8 | FUNCTIONS FOR PIONEER ROBOT CONTROL | 38 |
| 4.8.1 | PACKET CONSTRUCTION | 38 |
| 5. | AYLLU BEHAVIORS FOR PIONEER CONTROL | 39 |
| 5.1 | PIONEER BASE CONTROLLER | 39 |
| 5.2 | SERVER PACKET PARSER | 40 |

| | | |
|------------|---|-----------|
| 5.3 | UNIVERSAL SONAR INTERPRETER | 42 |
| 5.4 | VISION PACKET PARSER | 42 |
| 5.5 | PAN-TILT-ZOOM CAMERA CONTROLLER | 43 |
| 5.6 | LOW-LEVEL COMMUNICATION | 44 |
| 5.6.1 | PIONEER ROBOT STARTER | 44 |
| 5.6.2 | PIONEER PACKET RECEIVER | 44 |
| 5.6.3 | PIONEER PACKET SENDER | 45 |
| 6. | EXTENDED EXAMPLES | 47 |
| 6.1 | AN AYLLU ROBOCUP SOCCER SYSTEM | 47 |
| 6.2 | HELICOPTER TRACKING: MULTI-ROBOT COORDINATION STRATEGIES | 53 |
| 6.2.1 | BEHAVIOR DEFINITIONS | 54 |
| 6.2.2 | CONNECTIONS | 55 |
| 7. | INDEX | 57 |

| | |
|--|----|
| Figure 1: Behavior diagram of <i>straightline.c</i> | 11 |
| Figure 2: Behavior diagram of <i>wander.c</i> | 16 |
| Figure 3: Behavior diagram of multi-robot <i>moderndance.c</i> system | 18 |
| Figure 4: Behavior diagram of <i>goto.c</i> | 20 |
| Figure 5: Behavior diagram of <i>getto.c</i> | 22 |
| Figure 6: Behavior diagram of <i>march.c</i> , demonstrating a formation of three robots | 24 |
| Figure 7: Behavior diagram of RoboCup soccer system | 47 |
| Figure 8: Communication Diagram of the Helicopter-Tracking System | 53 |

1. Introduction to Ayllu

1.1 What is Ayllu?

Ayllu is a tool for development of distributed control systems for groups of mobile robots. It facilitates communication between distributed system components and scheduling of tasks. While Ayllu has many features specialized for behavior-based systems, it is useful for development of a wide range of architectures from reactive to deliberative, and provides the means for coordinating processor-intensive tasks, such as high-level planning and vision processing, with responsive low-level control. A small interface, referred to as AylluLite, can be easily added to non-Ayllu programs, allowing Ayllu to serve as a simple and effective "glue" between heterogeneous system components. It is designed to be highly portable among operating systems and languages, and allow maximal interoperability.

Ayllu's principal goal is facilitate implementation of robust multi-robot systems that must cope with noisy and range-limited communication, rapidly-changing real-world situations, variations in resource availability, tasks that require redistribution of system resources, and various hardware failures. Towards this goal, Ayllu 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 Pioneer Mobile Robots. 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) is 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 (see Section 2.1). 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 Pioneer's 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 tutorial in Section 2, along with the extended examples in Section 6, demonstrates how such simple building blocks 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 written by 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 state. **monostables** are accessible only to processes within their behavior.

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 **suppress** its destination. 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 **inhibit** 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 4.1.1.

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 in the interface specification 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 identifiers that have been declared as Ayllu hosts.

1.7 How does Ayllu provide for Pioneer control?

The following Standard Behaviors for Pioneer Control, described in detail in Section 5, 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), rotate to specified heading, change heading by degrees, translate by millimeters, stop. The PBC also processes gripper and Expansion Module commands.

- **USI** - The Universal Sonar Interpreter provides generalized sonar readings that are valid for both Pioneer 1 and 2 sonar configurations - Front, LeftFront, Left, LeftSide, LeftRear, Rear, RightRear, Right, and RightFront. Programs written using these ports will be portable across Pioneer models.
- **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.
- **RW** - The Robot Starter opens and maintains a serial connection with the Pioneer. In the case of a reset, the RW 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. A Brief Tutorial

The best way of presenting the use of Ayllu is through demonstration. The process of programming an Ayllu system tends to be incremental, starting with simple robot behaviors that can be thoroughly tested before more complexity is added to the system. In this section we give an extended example of such incremental development as a tutorial introduction to the structures and techniques of Ayllu.

2.1 Velocity Control - program straightline.c

The most basic capability necessary for this (and almost any) system is the ability to move without colliding with anything. We will implement a *SafeVelocity* behavior which sets the robot's forward velocity to a maximal safe value, based on the readings of the forward-facing sonars. Since Ayllu de-couples control of forward and rotational motion, we don't need to complicate this behavior with any details of rotation; other behaviors can handle rotation without interfering with *SafeVelocity*'s operation.

Intuition: The maximal safe velocity is proportional to the closest sonar reading; that is, the closer an obstacle is detected, the slower the robot must move. Thus our behavior will basically do nothing but scale the minimum of its inputs. At some point, however, we'd actually like the robot to back up slowly (if it is too close to an obstacle), and we'd like to be able to limit the maximum velocity with a parameter. Thus, a function for this behavior could be paraphrased as:

```
if min_sonar_dist > backup_dist (in millimeters)
    velocity = min(scale * min_sonar_dist, max_speed) mm/second
else    velocity = -100 mm/second
```

Implementation: An Ayllu behavior is implemented in two parts, a *behavior class definition*, which specifies the behavior's interface and included processes, and one or more *process definitions*. Here we will need only a single process. The interface must allow the behavior to receive sonar readings and output the resulting speed. We'd also like the three parameters (*scale*, *max_speed*, and *backup_dist*) to be specified in the interface so that they can be changed from outside the behavior. We specify the behavior class as follows:

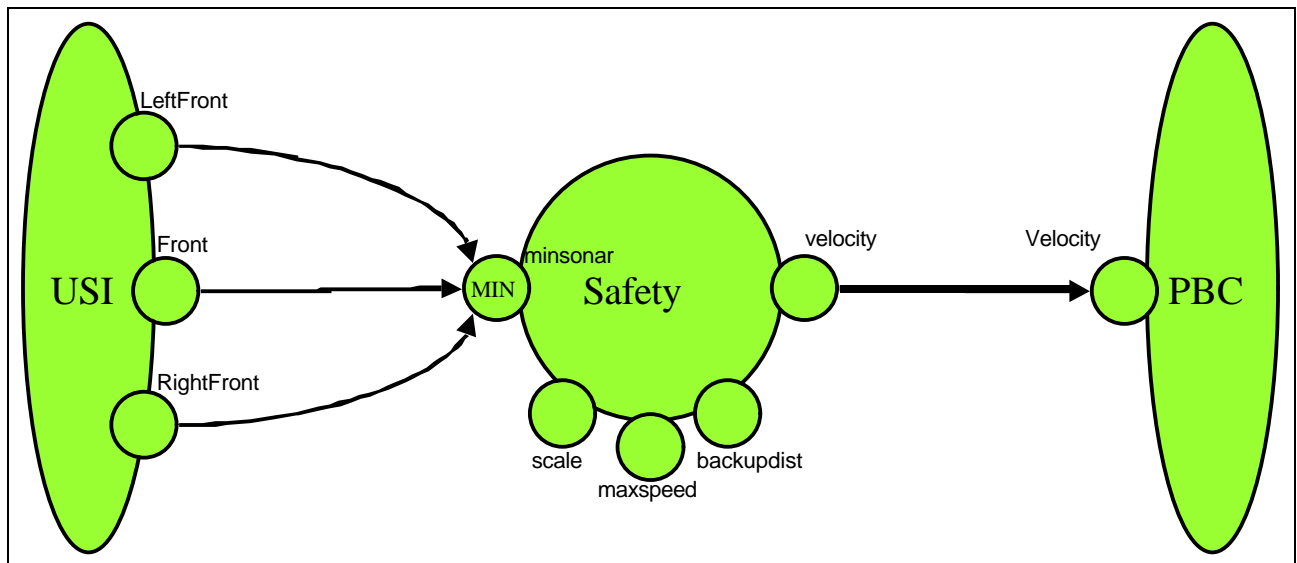


Figure 1: Behavior diagram of *straightline.c*

2.1.1 Behavior Class Definition

```
ayDefBehaviorClass(SafeVelocity)
{ ayINTERFACE {
    ayIntMinPort(minsonar, 5000); /* all of these ports */
    ayIntPort(velocity, 0);      /* can be accessed from */
    ayFloatPort(scale, 0.3);    /* outside the behavior */
    ayIntPort(maxspeed, 400);
    ayIntPort(backupdist, 400);
    }
  ayPROCESSES {
    ayInitProcess(SetSafeVelProc, ratepersecond(20));
    }
}
/* the process defined */
/* as SetSafeVel will */
/* run 20 times/sec */
```

In general, a *port* retains any value that is written to it. Thus *scale*, *maxspeed*, and *backupdist* defined above can be modified at run-time from outside the behavior. *minsonar* is defined as an *ayIntMinPort*; this means that when another behavior attempts to write a value to *minsonar*, that value is compared to *minsonar*'s current value, and the new value is only written if it is smaller than the current value. In the section **Connections** below, you'll see how this is useful.

2.1.2 Process Definition

The process, which specifies the computation that takes place in the behavior, can be specified as follows:

```
ayDefProcess(SetSafeVelProc)
{
  ayLocalPort minsonar, velocity, scale, maxspeed, backupdist;
  /* These must all be ports specified in ayINTERFACE section */
  /* of any behavior that this process is included in. Their */
  /* types are not specified here. */
  int closest;

  if ((closest=ayReadIntPort(minsonar)) > ayReadIntPort(backupdist))
    ayWriteIntPort(velocity, min(ayReadFloatPort(scale) * closest,
                                ayReadIntPort(maxspeed)));
  else
    ayWriteIntPort(velocity, -100);

  ayWriteIntPort(minsonar, 5000); /* This resets the MinPort to */
                                /* the largest possible sonar */
                                /* reading. */
}
}
```

2.1.3 Behavior Instantiation

We must instantiate the behavior in the main program. We name the instance *Safety*.

```
ayInitBehavior(SafetyVelocity, Safety);
```

2.1.4 Connections

Finally, we must *connect* the ports of our behavior instance *Safety* to ports of the standard Ayllu sensor and motor-control behaviors (discussed fully in Section 5). First, the appropriate sonar readings are connected to *minsonar*. For very robust and cautious obstacle detection, we connect the *Front*, *LeftFront*, and *RightFront* sonar readings from the sonar interpreting behavior *USI* (see Section 5.3) to *minsonar*. Since *minsonar* is an **ayIntMinPort** rather than a plain **ayIntPort**, the minimum of these (which is the closest object detected by any of them) is what remains in the port. Values of sonar readings are specified in millimeters.

```
ayConnect(USI, Front, Safety, minsonar);
ayConnect(USI, LeftFront, Safety, minsonar);
ayConnect(USI, RightFront, Safety, minsonar);
```

Next, we must connect the output of the behavior, *velocity*, to the appropriate port in the base-controller behavior *PBC* (see Section 5.1). This port accepts a velocity in mm/second.

```
ayConnect(Safety, velocity, PBC, Velocity);
```

We won't connect the ports which represent parameters (*maxspeed*, *backupdist*, and *scale*) right now, but if ever it is important for these to be manipulated by another behavior (or a user interface), the capability is in place. This is done by the *go* behavior, as discussed in Section 2.4.

2.1.5 Initializing Ayllu and Running Behaviors

The *main* function of an Ayllu program takes the following form:

```
void main()
{
    Declarations

    ayInitPioControl("/dev/cual"); /* Specifies the serial port that connects to robot
*/
    ayInitIPComms();              /* Necessary only when IP communication will be used
*/

    Behavior initializations

    Connections

    ayRunBehaviors();            /* does not return until system shuts down */
}
```

Behavior initializations (**ayInitBehavior**) and connections (**ayConnect**, **ayInhibitOut**, **ayOverrideIn**, etc.) are described above. **ayInitPioControl** starts up the standard Ayllu behaviors and determines the serial port to use; it *must* be called before any behavior initializations or connections. **ayRunBehaviors** actually runs all of the specified behaviors, and so does not return until some event in one of the behaviors causes the system to shut down (through **ayStopBehaviors**).

2.1.6 Putting It All Together

Since this is the first completed behavior of our tutorial, we will write out an entire program that runs this behavior to demonstrate how all the pieces fit together. We will not do this for subsequent examples, since the process is straightforward; however, the code for all examples is available on-line at www.activmedia.com/robots.

Program "straightline.c" - runs an instance of behavior *SafeVelocity*.

```
#include "ayllu.h"

ayDefProcess(SetSafeVelProc)
{
    ayLocalPort minsonar, velocity, scale, maxspeed, backupdist;
    int closest, safevel;

    if ((closest=ayReadIntPort(minsonar)) > ayReadIntPort(backupdist))
        ayWriteIntPort(velocity, min(ayReadFloatPort(scale) * closest,
                                ayReadIntPort(maxspeed)));
    else
        ayWriteIntPort(velocity, -100);
    ayWriteIntPort(minsonar, 5000);
}

ayDefBehaviorClass(SafeVelocity)
{ ayINTERFACE {
        ayIntMinPort(minsonar, 5000);           /* all of these ports */
        ayIntPort(velocity, 0);                /* can be accessed from */
        ayFloatPort(scale, 0.3);               /* outside the behavior */
        ayIntPort(maxspeed, 500);
        ayIntPort(backupdist, 400);
    }
    ayPROCESSES {
        ayInitProcess(SetSafeVelProc, ratepersecond(20));
    }
}

void main(void)
{
    ayInitPioControl("/dev/cual");
    ayInitBehavior(SafetyVelocity, Safety);

    ayConnect(USI, Front, Safety, minsonar);
    ayConnect(USI, LeftFront, Safety, minsonar);
    ayConnect(USI, RightFront, Safety, minsonar);
    ayConnect(Safety, velocity, PBC, Velocity);
    ayRunBehaviors();
}
```

Note that the order of definition and use of behaviors and processes is important, as per C rules for function and variable definition; each item must be defined before it can be referenced. Ayllu allows these to be prototyped, so that definitions can be deferred and header files can be constructed for multiple source files. The prototypes for our behavior take the form:

```
ayDefBehavior(SafetyVelocity);
ayDefProcess(SetSafeVel);
```

2.1.7 Resulting Behavior

The robot will move forward until it gets close to an obstacle. If the obstacle moves towards the robot, the robot will back slowly away. The farther the closest obstacle, the faster the robot will go. By itself, this tends to cause the robot to find an obstacle and maintain a distance in front of it, since the robot only moves forward and backward in a straight line.

2.2 Lively Motion Around an Area - program wanderer.c

Though the *SafeVelocity* behavior is so useful as to become ubiquitous, the *straightline* program by itself is not particularly interesting. In this section we add a behavior for rotational control that allows the robot to continuously roam around in a lively manner.

Intuition: We take the simplest approach: the robot turns away from the closest obstacle it detects, at a fixed (parameterized) rotational velocity:

```
if min(leftside, rightside) < avoid_thresh
    if leftside < rightside
        rotate right avoid_rot_vel deg/second
    else rotate left avoid_rot_vel deg/second
else rotate 0 deg/second
```

2.2.1 Behavior Definition

We need to combine *Left* and *LeftFront*, and *Right* and *RightFront*, readings from the sonar interpreter *USI* (Section 5.3), and accept parameters *avoidthresh* and *avoidrotvel*.

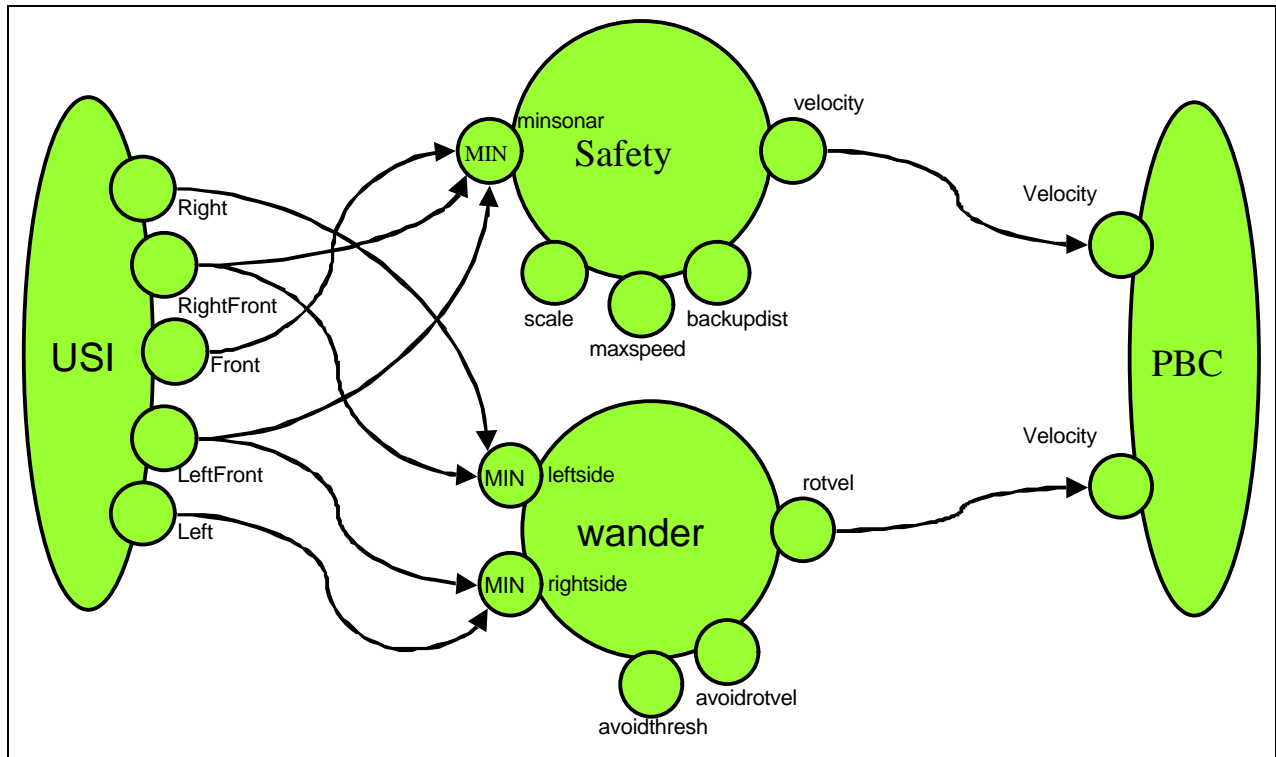


Figure 2: Behavior diagram of `wander.c`

```

ayDefBehaviorClass(WanderHeadingControl)
{ ayINTERFACE {
    ayIntMinPort(leftside, 5000);           /* mm */
    ayIntMinPort(rightside, 5000);        /* mm */
    ayIntPort(avoidthresh, 1500);         /* mm */
    ayIntPort(avoidrotvel, 40);           /* deg/second */
    ayIntPort(rotvel, 0);                 /* deg/second */
}
ayPROCESSES {
    ayInitProcess(WanderHeadProc, ratepersecond(5));
}
}

```

2.2.2 Process Definition

```

ayDefProcess (WanderHeadProc)
{
    ayLocalPort leftside, rightside, avoidthresh, avoidrotvel, rotvel;
    int left, right;

    left = ayReadIntPort(leftside); right = ayReadIntPort(rightside);

    if (min(left, right) < ayReadIntPort(avoidthresh))
        { if (left < right)

```

```

        ayWriteIntPort(rotvel, -ayReadIntPort(avoidrotvel));
    else ayWriteIntPort(rotvel, ayReadIntPort(avoidrotvel));
    }
else ayWriteIntPort(rotvel, 0);

ayWriteIntPort(leftside, ayMAXINT);      /* reset MinPorts */
ayWriteIntPort(rightside, ayMAXINT);
}

```

2.2.3 Connections

After instantiating *WanderHeadingControl* as *wander* (with **ayInitBehavior**), we make the following connections:

```

ayConnect(USI, LeftFront, wander, leftside);
ayConnect(USI, Left, wander, leftside);
ayConnect(USI, Right, wander, rightside);
ayConnect(USI, RightFront, wander, rightside);
ayConnect(wander, rotvel, PBC, AngularVel); /*see section 5.1*/

```

2.2.4 Resulting Behavior

We add all the code from this section to program *straightline* (Section 2.1) to produce program *wanderer*. The robot will now scamper around the room fairly quickly, veering away from obstacles when at a distance, or slowing to turn away if they are close. Though it may be necessary to tune the parameters for your environment, this behavior should allow the robot to run around continuously for an arbitrary time, and appear interesting while doing so.

2.3 Minimal Coordinated Action - program moderndance.c

Now we take a slight detour to introduce some of Ayllu's features for multi-robot control. We will demonstrate the simplicity of Ayllu's multi-robot communication and the flexibility of Ayllu's port-based communication through implementation of a system in which two or more robots do a minimalist modern dance.

Intuition: Modern dance is easy! We needn't change any of the behaviors at all, just the connections. The idea is that we leave the heading control (behavior *wander*) as it is, but change the connections so that sonar information from all robots is fed into each robot's velocity-control behavior. This way, all robots will move with a velocity equal to the global maximum safe velocity - that is, all robots will always move at the same velocity, determined by the sonar readings of the robot closest to an obstacle. Thus, they will rotate as conditions dictate individually, but will speed up and slow down as a group. With the right environment and the right music...

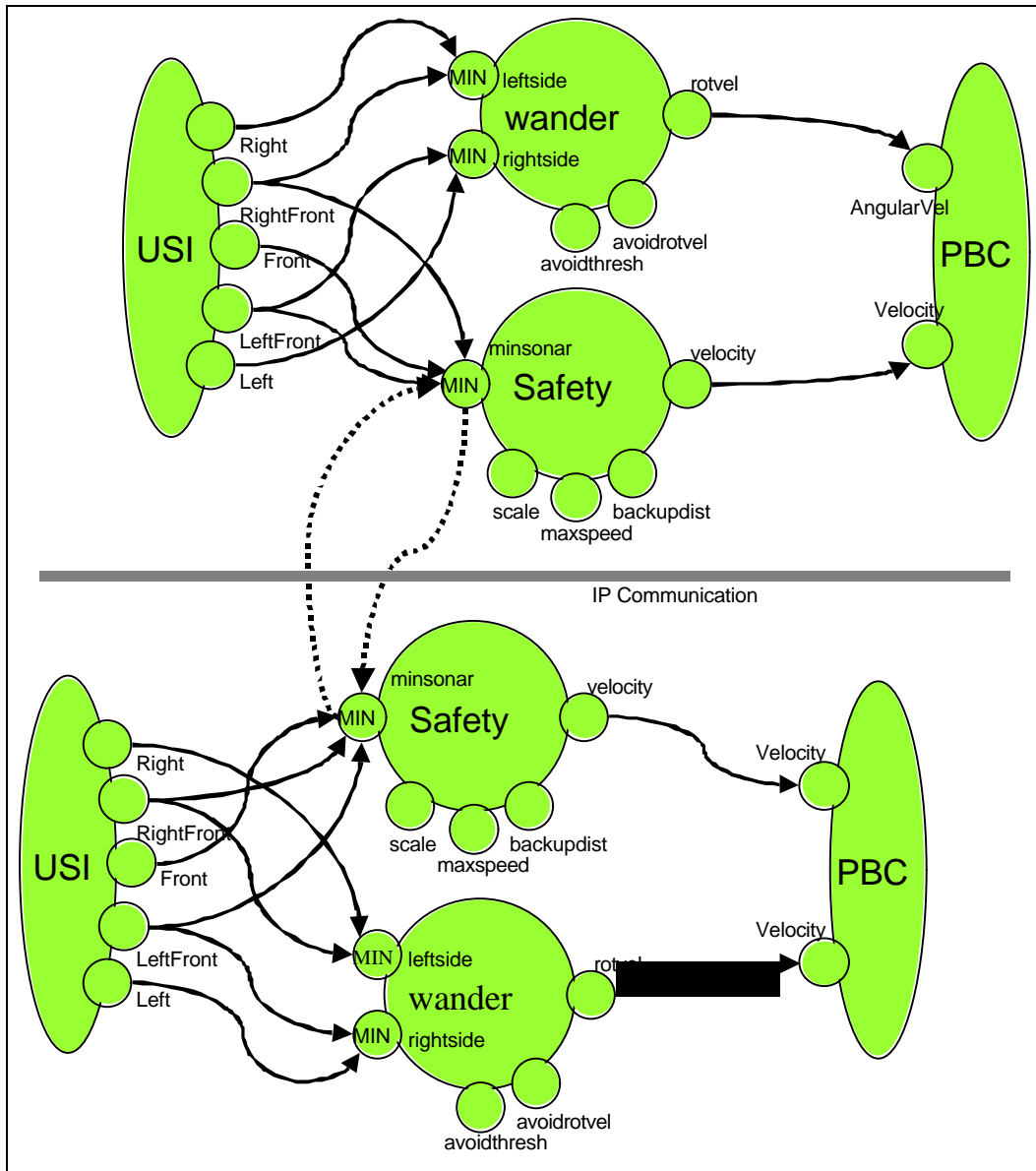


Figure 3: Behavior diagram of multi-robot *moderndance.c* system

2.3.1 Connections

Assume we have two robots controlled by computers at IP addresses **robot1.bluerobot.com** and **robot2.bluerobot.com**. We modify program *wander* as follows:

First, we introduce each robot to the other in *main()*. We declare in the variable declarations of *main()*:

```
ayHOST partner;
```

Before calling any other Ayllu functions, we must initiate network communications:

```
ayInitIPComms();
```

Then we assign the variable *partner* to the Ayllu host running on the other robot with

```
partner = ayMakeRemoteHost("robot2.bluerobot.com");
```

on robot1, and

```
partner = ayMakeRemoteHost("robot1.bluerobot.com");
```

on robot2.

The address can of course be passed as a command line argument, or input by the user, for more flexibility.

Finally, we add the following connections:

```
ayConnectFromRemote(partner, USI, Front, Safety, minsonar);
ayConnectFromRemote(partner, USI, LeftFront, Safety, minsonar);
ayConnectFromRemote(partner, USI, RightFront, Safety, minsonar);
```

Since the *partner* connections specify an **ayMinPort** as their destination, we can connect an arbitrary number of **ayHOSTS** as partners. For example, given an array *hosts* of *n* IP addresses we can write:

```
(in declarations)
ayHOST partners[n];

(in body)
ayInitIPComms(); /* necessary before any IP-related function calls */

for (i=0; i<n; i++)
{ if (ayMyIPNum() != ayIPNum(partners[i])) /* so that the same list can work
*/
{ /* on any host
*/
partners[i] = ayMakeRemoteHost(hosts[n]);
ayConnectFromRemote(partners[n], USI, Front, Safety, minsonar);
ayConnectFromRemote(partners[n], USI, LeftFront, Safety, minsonar);
ayConnectFromRemote(partners[n], USI, RightFront, Safety, minsonar);
}
}
```

2.3.2 Resulting Behavior

When this code is added to *wander* we get the modern dance we were looking for... the robots remain synchronized in speed, while de-coupled in rotation. If the environment is very cluttered, it is possible that the robots will always go slow or sit still, but if there is a fair amount of room, they should cover a range of speeds nicely.

2.4 Going to a Specific Point - program goto.c

A very basic and useful behavior is that of going to a specific point. The program *goto* incorporates a behavior class *GotoPointDR* that does this using the Pioneer's dead-reckoning ability - though if other means of determining position, such as GPS, are available, their data can be fed into the same behavior.

Intuition: Basically, the robot continuously rotates to point towards the target point, and changes its speed proportionally to distance to the target point, until within some radius of the target:

```
if target_dist < target_radius (in millimeters)
    velocity = 0
else
    velocity = min(scale * target_dist, max_vel);
    heading = angle_to_target
```

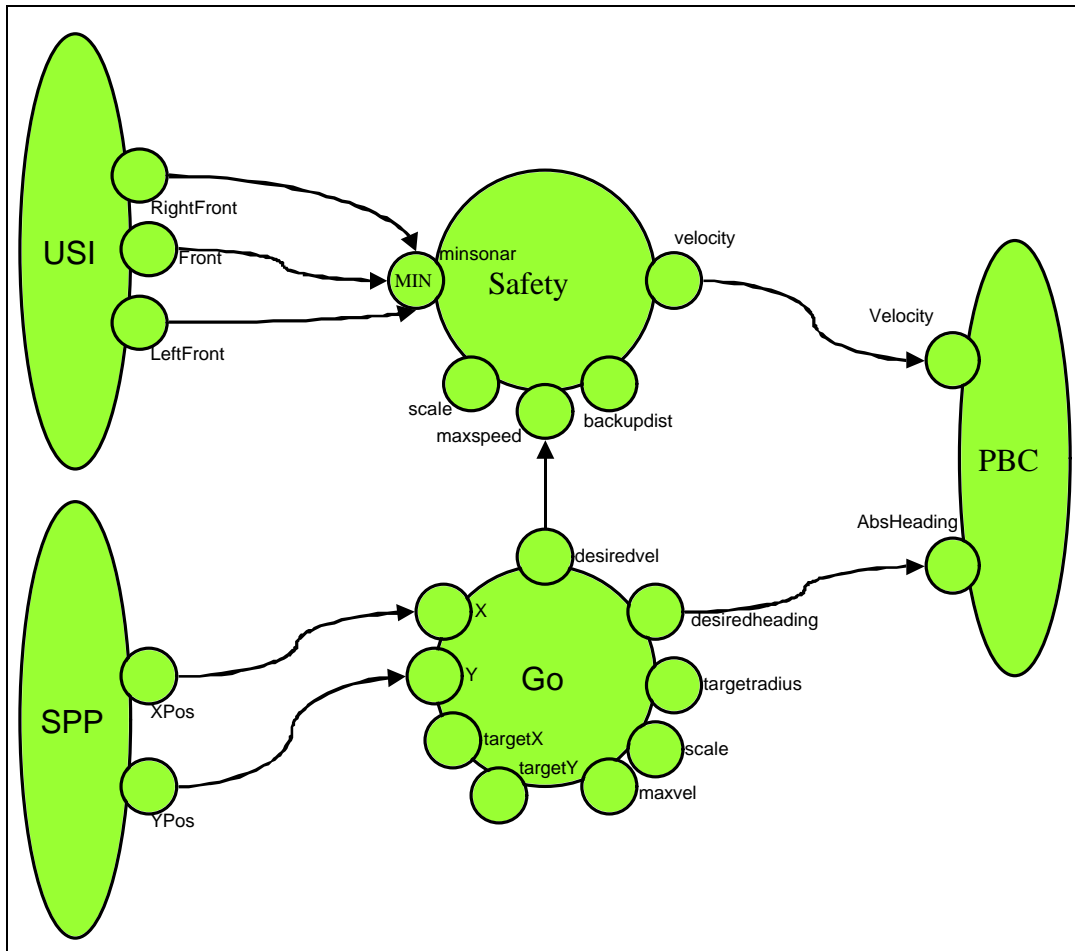


Figure 4: Behavior diagram of `goto.c`

2.4.1 Behavior Definition

We need to combine *Left* and *LeftFront*, and *Right* and *RightFront*, readings from the sonar interpreter *USI* (Section 5.3), and accept parameters *avoidthresh* and *avoidrotvel*.

```

ayDefBehaviorClass(GotoPointDR)
{ ayINTERFACE {
    ayIntPort(targetx, 0);          /* control inputs - set target */
    ayIntPort(targety, 0);
    ayFloatPort(x, 0);             /* current location */
    ayFloatPort(y, 0);
    ayIntPort(targetradius, 500); /* parameters*/
    ayIntPort(maxvel, 500);
    ayFloatPort(scale, 0.4);
    ayIntPort(desiredheading, 0); /* outputs to motor control */
    ayIntPort(desiredvel, 0);
    ayIntPort(arrived, 0);        /* notification of status */
}
ayPROCESSES {
    ayInitProcess(GotoDRProc, ratepersecond(10));
}

```

```

    }
}

```

2.4.2 Process Definition

```

ayDefProcess (GotoDRProc)
{
  ayLocalPort targetx, targety, x, y, targetradius, maxvel, scale,
              desiredheading, desiredvel, arrived;
  int targetdist;

  targetdist = ayDistance(ayReadIntPort(targetx), ayReadIntPort(targety),
                          ayReadFloatPort(x), ayReadFloatPort(y));

  if (targetdist < ayReadIntPort(targetradius)) /* if we are close enough, */
    { if (!ayReadIntPort(arrived))             /* stop moving and notify any */
      { ayWriteIntPort(desiredvel, 0);         /* interested parties */
        ayWriteIntPort(desiredheading, 0);
        ayWriteIntPort(arrived, 1);
      }
    }
  else /* otherwise, turn towards target
  */
    { /* and move at reasonable speed */
      ayWriteIntPort(desiredheading,
                    ayAngleTo(ayReadIntPort(targetx), ayReadIntPort(targety),
                              ayReadFloatPort(x), ayReadFloatPort(y)));
      ayWriteIntPort(desiredvel, min(ayReadIntPort(maxvel),
                                     (ayReadFloatPort(scale) * targetdist)));
      ayWriteIntPort(arrived, 0); /* let 'em know we're not there
yet */
    }
}

```

2.4.3 Connections

We add the above code to *straightline.c* (section 2.1.6), and instantiate *GotoPointDR* as *go*, we make the following connections:

```

ayConnect(SPP, Xpos, go, x);
ayConnect(SPP, Ypos, go, y);
ayConnect(go, desiredheading, PBC, AbsHeading);
ayConnect(go, desiredvel, SafetyVelocity, maxspeed); /* this allows us to maintain
*/
/* the properties of
SafetyVelocity */
/* namely, not hitting
anything */

```

2.4.4 Resulting Behavior

This *goto* program does little more than demonstrate the function of the *GotoPointDR* behavior. Once you start the robot, if you shut off the motor power, push the robot to another location, and then turn motor power back on, the robot should return to the original location. (The initial values of *targetx* and *targety* (0, 0) reflect the point at which the robot was turned on). You can do this repeatedly if you have a lot of time on your hands, or if you want to

experiment with the increasing position error inherent in dead-reckoning; or you can continue on to more interesting behaviors....

2.5 Going to a Point with Obstacle Avoidance - program `getto.c`

You may have noticed that the `goto` program may result in the robot's straightline-path to the goal being blocked by some obstacle, in which case the robot will sit in front of the obstacle, facing the target point, ad-infinitum. Here we incorporate obstacle avoidance with point tracking for a more robust system that can deal with this situation.

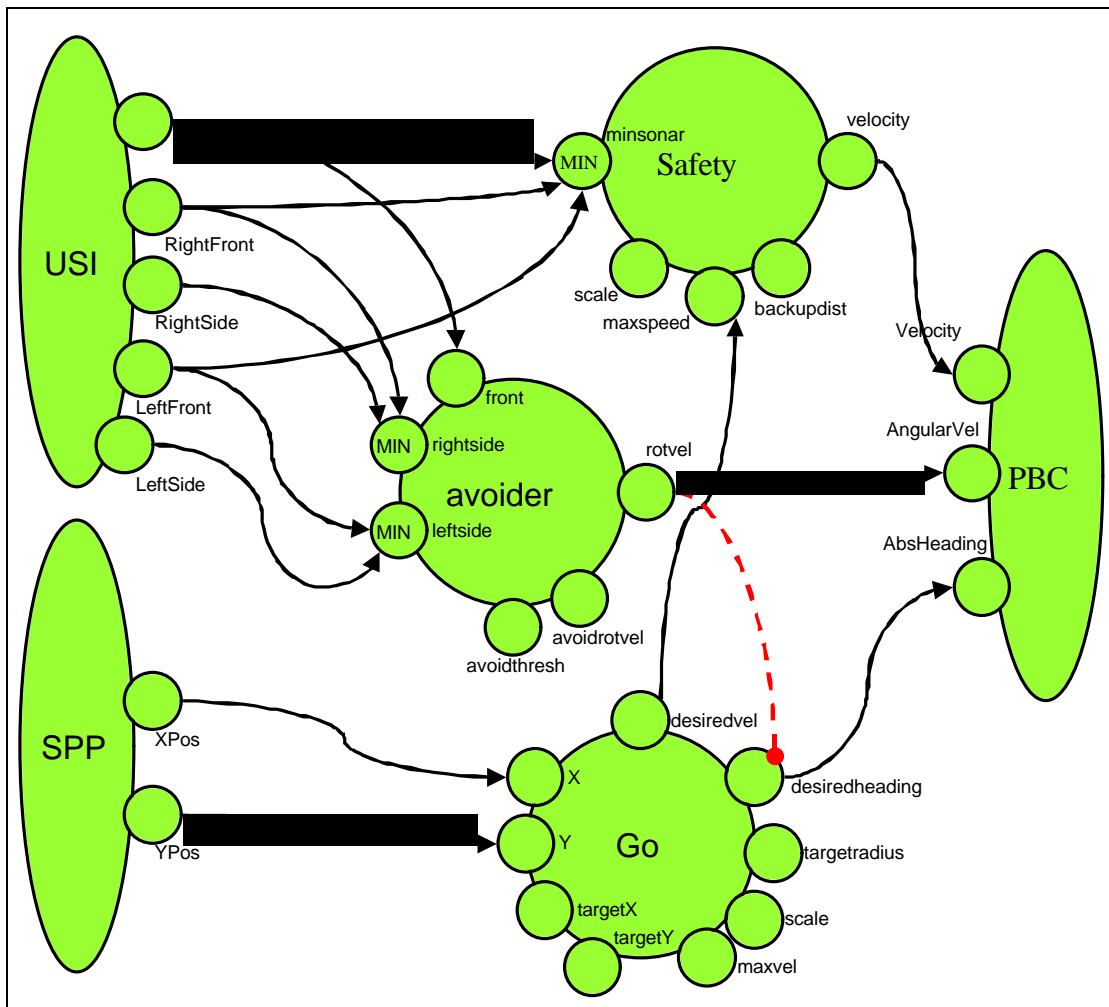


Figure 5: Behavior diagram of `getto.c`

Intuition: Whenever an obstacle is detected, the collision avoidance behavior overrides the point-tracking behavior. Thus, when near an obstacle, the avoidance behavior takes over for just long enough for the robot to clear the obstacle, at which point the point tracker will regain control and steer the robot again towards the target.

2.5.1 Behavior Definition

We are basically going to generalize the *wander* (section 2.2) behavior so that it is more appropriate for subsumptive-style control. That is, we will have it work exactly the same except that when no obstacles are detected, it will output nothing, rather than outputting a 0 degree (straight ahead) relative heading. This allows our new behavior, *Avoidance*, to override other behaviors only when appropriate - its output will be used not only to control the velocity

of the robot, but also to prevent another behavior from exercising similar control. How we do this is discussed in the **Connections** section below.

```

ayDefBehaviorClass(Avoidance)
{ ayINTERFACE {
    ayIntMinPort(leftside, 5000);          /* mm */
    ayIntMinPort(rightside, 5000);        /* mm */
    ayIntMinPort(front, 5000);           /* mm */
    ayIntPort(avoidthresh, 1000);        /* mm */
    ayIntPort(avoidrotvel, 50);          /* deg/second */
    ayIntPort(rotvel, 0);                /* deg/second */
}
ayPROCESSES {
    ayInitProcess(AvoidHeadProc, ratepersecond(10));
}
}

```

2.5.2 Process Definition

```

ayDefProcess (AvoidHeadProc)
{
    ayLocalPort leftside, rightside, front,
                avoidthresh, avoidrotvel, rotvel;
    int left, right;

    left = ayReadIntPort(leftside); right = ayReadIntPort(rightside);

    if (min(ayReadIntPort(front), min(left, right)) < ayReadIntPort(avoidthresh))
    { if ((left - right) > 200)
        ayWriteIntPort(rotvel, ayReadIntPort(avoidrotvel));
      else
        ayWriteIntPort(rotvel, -ayReadIntPort(avoidrotvel));
    }

    ayWriteIntPort(leftside, ayMAXINT);          /* reset MinPorts */
    ayWriteIntPort(rightside, ayMAXINT);
}

```

2.5.3 Connections

After adding the above code to *goto* and instantiating *Avoidance* as *avoider*, we make the following connections:

```

ayConnect(USI, LeftFront, avoider, leftside);
ayConnect(USI, Left, avoider, leftside);
ayConnect(USI, Right, avoider, rightside);
ayConnect(USI, RightFront, avoider, rightside);
ayConnect(USI, Front, avoider, front);
ayConnect(avoider, rotvel, PBC, AngularVel);
ayInhibitOut(avoider, rotvel, go, desiredheading, seconds(0.2));

```

The last line here specifies an *inhibitory connection*. This means that no data will be sent from the destination port for the specified period of time after it has been written through this connection. In this case, it means that as long as data is written on *avoider's rotvel* port, *go's desiredheading* port will not send out any messages. This allows *avoider* to take over control of the robot's rotation in the presence of an obstacle. As soon as the obstacle is cleared, *avoider*

stops sending rotation commands on its port *rotvel*, and 0.2 seconds later *go's desiredvel* port once again gains control of the robot's rotation.

2.5.4 Resulting Behavior

The robot will track a point as with *goto*, but if an obstacle is encountered it will move away from the obstacle. In many cases this leads to a situation in which the robot will regain a clear path to make progress directly towards the obstacle, although sometimes the robot may get into a situation (such as a box canyon) in which this combination of behaviors will lead to looping. In general, however, due to the real-world uncertainty, the robot will get to the target, albeit with perhaps a bit of fummerring.

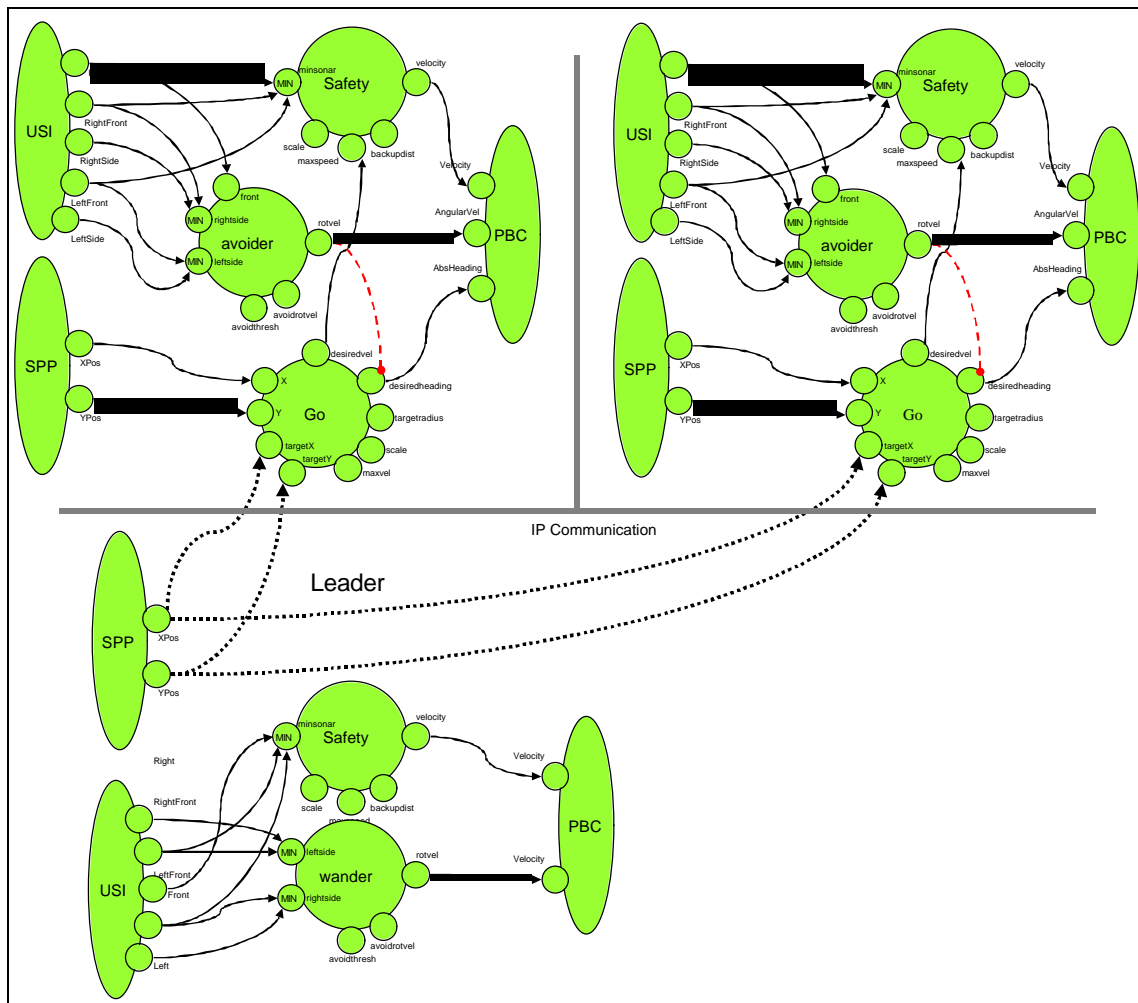


Figure 6: Behavior diagram of *march.c*, demonstrating a formation of three robots

2.6 Traveling in Formation - program *march.c*

We now have everything we need for a group of robots to move around "in formation."

Intuition: The robots consider the origin of their coordinate system (0,0) to be the point at which they are first turned on. This means that, if the robots are placed in a formation and then turned on, they remain in the same formation while moving about if they all have the same position on the coordinate system (that is, if at any given moment their (x,y) positions match). We designate one robot as a leader, which moves about and broadcasts its position to the

others. The rest of the robots merely run a variant of the *getto* program in which the leader's position is continuously fed into their *go* behaviors as the target point (which retained a (0,0) target value in *getto*).

2.6.1 Putting It All Together

We don't need to define any new behaviors; instead, we will provide the entire *main()* for the *march* program. We'll include headers for the behaviors for clarity; you should make sure to include a source file with the behaviors or replace the headers with definitions for this to be linkable.

```
#include "ayllu.h"

ayDefBehaviorClass(SafeVelocity);
ayDefBehaviorClass(GotoPointDR);
ayDefBehaviorClass(Avoidance);
ayDefBehaviorClass(WanderHeadingControl);

int n;
char *partners[];

void main(argc, char *argv[])
{
    ayHOST partners[n];

    /* insert something to define partner list */

    ayInitPioControl("/dev/cual");
    ayInitIPComms();

    ayInitBehavior(SafetyVelocity, Safety);
    ayConnect(USI, Front, Safety, minsonar);
    ayConnect(USI, LeftFront, Safety, minsonar);
    ayConnect(USI, RightFront, Safety, minsonar);
    ayConnect(Safety, velocity, PBC, Velocity);

    if (argc > 1) /* we're the leader */
    {
        ayInitBehavior(WanderHeadingControl, wander);
        ayConnect(USI, LeftFront, wander, leftside);
        ayConnect(USI, Left, wander, leftside);
        ayConnect(USI, Right, wander, rightside);
        ayConnect(USI, RightFront, wander, rightside);
        ayConnect(wander, rotvel, PBC, AngularVel);

        aySendIntMessage(Safety, maxspeed, 300); /*should move slower than others
*/
        for (i=0; i<n; i++)
        { if (ayMyIPNum() != ayIPNum(partners[i]))
            {
                followers[i] = ayMakeRemoteHost(hosts[n]);
                ayConnectToRemote(SPP, Xpos, partners[n], go, targetx);
                ayConnectToRemote(SPP, Ypos, partners[n], go, targety);
            }
        }
    }
    else /* we're a follower */
    {
```

```

ayInitBehavior(GotoPointDR, go);
ayInitBehavior(Avoidance, avoider);

ayConnect(go, desiredheading, PBC, AbsHeading);
ayConnect(go, desiredvel, Safety, maxspeed);
ayConnect(SPP, Xpos, go, x);
ayConnect(SPP, Ypos, go, y);
ayConnect(USI, LeftFront, avoider, leftside);
ayConnect(USI, Left, avoider, leftside);
ayConnect(USI, Front, avoider, front);
ayConnect(USI, Right, avoider, rightside);
ayConnect(USI, RightFront, avoider, rightside);
ayConnect(avoider, rotvel, PBC, AngularVel);
ayInhibitOut(avoider, rotvel, go, desiredheading, seconds(0.2));
}
ayRunBehaviors();
}

```

2.6.2 Resulting Behavior

The robots will move about in more or less their initial configuration, provided the area is large enough. Robots blocked by obstacles will attempt to get around them and regain their position, just as they got around obstacles in the previous *getto* program (Section 2.5). The "tightness" of the formation can be adjusted through the *targetradius* ports of the followers. Note that the formation does not rotate; if the leader is in the front of the group and turns around, the group will proceed with the leader in the rear. In the next section we'll explore how to increase the sophistication of this and other systems we have demonstrated.

2.7 Further Developments

We hope that this tutorial has clearly demonstrated both the technique and philosophy of Ayllu programming, especially the process of increasing complexity by combining simple components. It is possible to improve behaviors without interfering with the behavior of others, and even change the sources of their information. As we mentioned before, the coordinate system used in *go* can be produced by any device and used without code changes in the *goto* and *getto* programs. Better obstacle avoidance can be incorporated, using representational techniques such as maps or improved sensor-based techniques, without modifications to other behaviors. If the *avoidance* behavior is fed with information from *go's* *desiredheading* behavior, it can make better decisions about which direction to turn to get around obstacles.

The formation behavior would benefit from this better obstacle avoidance, but can of course be improved in other ways. If the leader broadcasts its heading, each robot can transform its coordinates in such a way that the whole formation rotates so that the leader remains in front. Using a global coordinate system, such as GPS, would not require changes to the existing behaviors but would require initial determination of coordinate offsets from the leader's. This could be accomplished by a behavior "filters" input from the leader.

Section 6 provides an additional, longer example - an Ayllu RoboCup robot soccer system; and there will be a constant supply of new Ayllu code at <http://robots.activmedia.com> - hopefully soon to include your contributions.

3. Ayllu Language Extensions

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 3.4) 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.

3.1 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 3.4. Each process initialization takes the following form:

ayInitProcess(*procname*, *rate*)

where *procname* is a name given in a process definition (see 3.3), 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 Sections 2 and 6 for full examples of behavior definitions.

3.2 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 3.3.

3.3 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 3.4.4; 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 Sections 2 and 6 for full examples of process definitions.

3.4 Declarations

3.4.1 Ports

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

3.4.1.1 Normal Ports

| | |
|--|--|
| ayIntPort (<i>name</i> , <i>value</i>) | <i>32-bit integer value</i> |
| ayFloatPort (<i>name</i> , <i>value</i>) | <i>double floating point value</i> |
| ayStringPort (<i>name</i> , <i>value</i>) | <i>string value - char * to region inside port</i> |
| ayMemPort (<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");
```

3.4.1.2 Summation Ports

| | |
|---------------------------------------|------------------------------------|
| ayIntSumPort (<i>name</i>) | <i>32-bit integer value</i> |
| ayFloatSumPort (<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);
```

3.4.1.3 Extrema Ports

| | |
|--|------------------------------------|
| ayIntMaxPort (<i>name</i> , <i>val</i>) | <i>32-bit integer value</i> |
| ayFloatMaxPort (<i>name</i> , <i>val</i>) | <i>double floating point value</i> |
| ayIntMinPort (<i>name</i> , <i>val</i>) | <i>32-bit integer value</i> |
| ayFloatMinPort (<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 of the use of **ayMinPorts** can be found in Sections 2.1, 2.3, and 6.2.

EXAMPLES:

```
ayIntMaxPort(Fastest, 5000);
ayFloatMaxPort(FarthestObject, MaxDistance);
```

3.4.1.4 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 4.1.4 and 4.1.6.

EXAMPLES:

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

3.4.2 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).

3.4.3 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.

3.4.4 Local Declarations

These local declarations appear in process definitions (as discussed in Section 3.3). 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;
```

3.4.5 Host Declarations

For an Ayllu system to initiate communication with another host, it must declare a variable to represent the remote host as follows:

ayHOST *varname*

where *varname* is any legal C identifier. Before this variable is referred to, it must be initialized, most likely through the **ayMakeRemoteHost** function (see Section 4.7, and Sections 2.3 and 2.6 for examples).

4. The Ayllu Function Library

The functions described in this section manipulate the status of the Ayllu scheduler, provide access to Ayllu's communication facilities, and perform system initializations. They do *not* provide for specific elements of robot control; that falls into the realm of the Ayllu Standard Behavior Library (see Section 5).

Note that certain functions only function within specific scopes - e.g., the top level of a process definition.

4.1 Port Functions

4.1.1 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.

4.1.2 Reading Local Ports

** THESE FUNCTIONS CAN ONLY APPEAR IN THE TOP LEVEL OF A PROCESS DEFINITION **

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

4.1.3 Data Availability

** THESE FUNCTIONS CAN ONLY APPEAR IN THE TOP LEVEL OF A PROCESS DEFINITION **

| | |
|------------|---|
| <i>int</i> | ayPortWritten (<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).

4.1.4 Port Priority

** THESE FUNCTIONS CAN ONLY APPEAR IN THE TOP LEVEL OF A PROCESS DEFINITION **

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

Priority ports (as discussed in Section 3.4.1.4) 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 4.1.6 can be used.

4.1.5 Writing to Local Ports

** THESE FUNCTIONS CAN ONLY APPEAR IN THE TOP LEVEL OF A PROCESS DEFINITION **

```
void      ayWriteIntPort(localport, int value)
void      ayWriteFloatPort(localport, double value)
void      ayWriteStrPort(localport, char * value)
void      ayWriteMemPort(localport, unsigned char * value)
```

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**).

4.1.6 Writing to Local Ports with Priority

** THESE FUNCTIONS CAN ONLY APPEAR IN THE TOP LEVEL OF A PROCESS DEFINITION **

```
void      ayWriteIntPriority(localport, int value,
                             double priority)
void      ayWriteFloatPriority(localport, double value,
                               double priority)
void      ayWriteStrPriority(localport, char * value,
                             double priority)
void      ayWriteMemPriority(localport, unsigned char * value,
                             double priority)
```

Each of these functions is equivalent to a call to **aySetPortPriority** followed by a call to **ayWriteTypePort**. See 4.1.4 above for information on how priority is assigned to messages.

4.1.7 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 ports. However, they can be extremely useful for diagnostic purposes, and for user interaction which runs as a separate thread within an Ayllu process.

4.1.7.1 Reading Peer Ports

```
int      ayIntPortValue(beh, port)
double   ayFloatPortValue(beh, port)
char *   ayStrPortValue(beh, port)
```

like **ayReadTypePort**, but addresses a port in another behavior and does not affect its **ayPortWritten** status.

4.1.7.2 Writing to Peer and Remote Ports

```
void      aySendIntMessage(beh, port, int value)
```

```

void      aySendFloatMessage(beh, port, double value)
void      aySendStrMessage(beh, port, char * value)
void      aySendMemMessage(beh, port, unsigned char * value) void
aySendRemoteInt(host, beh, port, int value)

void      aySendRemoteFloat(host, beh, port, double value)
void      aySendRemoteStr(host, beh, port, char * value)
void      aySendRemoteMem(host, beh, port,
                          unsigned char * value)

```

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

4.1.8 Connecting Ports

4.1.8.1 Normal Connections

```

void      ayConnect(srcbeh, srcport, destbeh, destport)
void      ayConnectToRemote(srcbeh, srcport, desthost,
                          destbeh, destport)
void      ayConnectFromRemote(srchost, srcbeh, srcport,
                          destbeh, destport)

```

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.

4.1.8.2 Suppressing/Inhibiting Connections

```

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

```

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 (s)`. Examples of inhibition can be seen in Sections 2.5 and 6.2.

4.1.8.3 Overriding Connections

```

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

```

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. Examples of overriding can be seen in Section 6.1.

4.2 Slot Functions

4.2.1 Writing to Slots

** THESE FUNCTIONS CAN ONLY APPEAR IN THE TOP LEVEL OF A PROCESS DEFINITION **

```

int      ayReadIntSlot(localslot)
double   ayReadFloatSlot(localslot)
char *   ayReadStrSlot(localslot)
unsigned char * ayReadMemSlot(localslot)

```

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

4.2.2 Reading Slots

** THESE FUNCTIONS CAN ONLY APPEAR IN THE TOP LEVEL OF A PROCESS DEFINITION **

```

void      ayWriteIntSlot(localslot, int value)
void      ayWriteFloatSlot(localslot, double value)
void      ayWriteStrSlot(localslot, char * value)
void      ayWriteMemSlot(localslot, unsigned char * value)

```

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

4.3 Monostable Functions

** THESE FUNCTIONS CAN ONLY APPEAR IN THE TOP LEVEL OF A PROCESS DEFINITION **

```

int      ayMonoActive(monostable)

```

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

** THESE FUNCTIONS CAN ONLY APPEAR IN THE TOP LEVEL OF A PROCESS DEFINITION **

```

void      ayTrigger(monostable)

```

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**.

Examples of the use of monostables can be seen in the soccer example of Section 6.1.

4.4 General Packet Queue Functions

| | |
|-------------------|---|
| <code>void</code> | <code>ayEnqueueGP(unsigned char *packet)</code> |
|-------------------|---|

As discussed in Section 5.6.3, there is a queue for packets to be sent to the robot base, for control of non-time-critical actions such as sound and digital I/O control. This function adds a packet to this queue.

4.5 Scheduler Control Functions

| | | |
|-------------------|---|---------------------|
| <code>void</code> | <code>ayRunBehaviors()</code> | |
| <code>void</code> | <code>ayBackgroundRunBehaviors()</code> | <i>Coming Soon!</i> |
| <code>void</code> | <code>ayStopBehaviors()</code> | |

ayRunBehaviors causes the scheduler to begin running all processes included in all instantiated behaviors, at their specified rates (see Section 3.1). 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 function is not yet available in Ayllu 1.0; if you have a newer version, please check the updated documentation). When the behaviors run in the background, the foreground process can communicate with them using the functions described in Section 4.1.7, and the scheduler can be shut down by a foreground call to **ayStopBehaviors**.

4.6 Pioneer Control Setup Function

| | |
|-------------------|--|
| <code>void</code> | <code>ayInitPioControl(char * serialport)</code> |
|-------------------|--|

ayInitPioControl instantiates the standard behaviors for Pioneer control (see Sections 1.7 and 5), 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*, *RW*, *PBC*, *PTZ*, and *PPS* are set to run and available for connections, and *PPS* and *PPR* will communicate with the robot base through *serialport*.

4.7 Remote Host Functions

| | |
|----------------------------|--|
| <code>ayHOST</code> | <code>ayMakeRemoteHost(char * IPaddr)</code> |
| <code>unsigned long</code> | <code>ayMyIPNum(void)</code> |
| <code>unsigned long</code> | <code>ayIPNum(char * IPaddr)</code> |

The **ayMakeRemoteHost** function returns an **ayHOST** structure that can be used in specifying remote connections. *IPaddr* may be either a hostname or a numerical address - i.e., either "robot1.bluerobot.com" or "128.253.2.101". Examples of use may be found in Sections 2.3 and 2.6.

ayIPNum returns the numerical representation of an IP address string of the same form as those acceptable by **ayMakeRemoteHost**; **ayMyIPNum** returns the numerical representation of the IP address of the machine executing the code. These two are intended to be compared to each other, as seen in Sections 2.3 and 2.6.

4.8 Functions for Pioneer Robot Control

4.8.1 Packet Construction

```
unsigned char * ayMakePSOSIntCommand(int command, int arg)
unsigned char * ayMakePSOSNoargCommand(int command)
unsigned char * ayMakePSOSStrnCommand(int command, char * arg, int len)
unsigned char * ayMakePSOSStrCommand(int command, char * arg)
unsigned char * ayMakePSOSSingCommand(char * arg, int len)
unsigned char * ayMakeDigOutCommand(int mask, int vals)
unsigned char * ayMakeADSelCommand(int pin)
unsigned char * ayMakeVisionCommand(char * command)
```

These functions return PSOS packets that can be sent down the serial line to the robot. In general, this should be done using **ayQueueGP** (see Section 4.4). They can also be directly written to appropriate ports of the PPS (see Section 5.6.3), but keep in mind that this may interfere with proper scheduling of other packets. Vision packets may be written to the PPS's **VisPacket** port, but if sent too frequently (more than 10/second) some may be overwritten. Only by queuing the packets can you be sure they will (eventually) be sent down the serial line.

The **PSOSInt**, **PSOSNoArg**, **PSOSStrn**, and **PSOSStr** command makers construct general PSOS packets with the appropriate argument types.

The **PSOSSing** command maker accepts a string consisting of pairs of bytes, the first of which indicates duration of a tone and the second of which indicates the frequency of the tone, as specified in the Pioneer Software Manual. Up to 40 pairs may be included in the string, and, when it is received by the robot, the robot will "sing" (or "beep") the specified notes.

The **DigOut** command maker specifies changes to the digital output lines. The *mask* argument (0-255) specifies the pins that will be affected (1 bits), and the *vals* argument specifies the values these pins should take on.

The **ADSel** command specifies which pin will be used for analog-to-digital conversion (in Pioneer 2 robots only). The acceptable values are 4-7 and 9; the value of the specified pin is reported thereafter in the **AnalogIn** port of the SPP (see Section 5.2).

VisionCommands are strings, discussed in the Cognachrome User's Manual.

5. 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 4.6), and all of their available ports with a short description of the messages they provide or accept.

5.1 Pioneer Base Controller

Prototype: `ayDefBehaviorClass(ayiBaseControllerMaker);`

Standard instance name: `PBC`

| Ports Used for Input | |
|--|--|
| <code>ayIntPort(ControlMode, ayFAST);</code> | Motor Controller mode - can be one of: <ul style="list-style-type: none"> • <code>aySMOOTH</code> - for smoother motion • <code>ayFAST</code> - for more responsive control |
| <code>ayIntPort(DefaultVel, 200);</code> | Set velocity to be used when executing distance-based commands (see Distance port) |
| <code>ayIntPort(MaxAngVel, 75);</code> | Set maximum angular velocity to be used in execution of <code>AbsHeading</code> and <code>RelHeading</code> commands. |
| <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"> • ayGRIPOPEN - open at the bottom position • ayGRIPTOP - closed at the top position • ayGRIPCARRY - closed at the carry position |
|---|--|
| Ports Used for Output | |
| <code>ayIntPort(TransInProgress, 0);</code> | Reports status of distance-based motion (initiated through a write to port Distance). TRUE (1) indicates that the robot has not yet completed the most recent distance-based command; 0 indicates that no distance-based commands are currently in progress. Any write to Stop , Velocity , RightWheelVel , or LeftWheelVel will cancel the current distance-based translation and cause this port to return 0. <i>Works only in ayFast mode.</i> |
| <code>ayIntPort(RotInProgress, 0);</code> | Reports status of degree-based motion (initiated through a write to port AbsHeading or RelHeading). TRUE (1) indicates that the robot has not yet completed the most recent degree-based command; 0 indicates that no degree-based commands are currently in progress. Any write to Stop , Velocity , RightWheelVel , or LeftWheelVel will cancel the current distance-based translation and cause this port to return 0. <i>Works only in ayFast mode.</i> |

5.2 Server Packet Parser

Prototype: `ayDefBehaviorClass(ayiSPParserMaker);`

Standard instance name: **SPP**

| Ports Used for Input | |
|---|--|
| <p><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.</p> | |
| <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 Pioneer1 has only 7 pingers, but the user can add an eighth and connect it to pinger 7. The Pioneer 2 uses eight pingers on the front sonar, with an optional eight-pinger rear sonar ring (Sonars 8-15) |
| <code>ayIntPort(Sonar8,0);</code> | Sonar reading of optional rear pinger 8, in millimeters |
| <code>ayIntPort(Sonar9,0);</code> | Sonar reading of optional rear pinger 9, in millimeters |
| <code>ayIntPort(Sonar10,0);</code> | Sonar reading of optional rear pinger 10, in millimeters |
| <code>ayIntPort(Sonar11,0);</code> | Sonar reading of optional rear pinger 11, in millimeters |
| <code>ayIntPort(Sonar12,0);</code> | Sonar reading of optional rear pinger 12, in millimeters |
| <code>ayIntPort(Sonar13,0);</code> | Sonar reading of optional rear pinger 13, in millimeters |
| <code>ayIntPort(Sonar14,0);</code> | Sonar reading of optional rear pinger 14, in millimeters |
| <code>ayIntPort(Sonar15,0);</code> | Sonar reading of optional rear pinger 15, in millimeters |
| <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 (0-255) |
| <code>ayIntPort(DigitalIn,0);</code> | State of the Pioneer's digital input pins (8 bits) |
| <code>ayIntPort(DigitalOut,0);</code> | State of the Pioneer's digital output pins (8 bits) |
| <code>ayIntPort(GripperBeams,0);</code> | State of the breakbeams in the optional gripper. One of: <ul style="list-style-type: none"> • <code>ayGRIPFRONTBEAM</code> • <code>ayGRIPREARBEAM</code> or • <code>ayGRIPBOTHBEAMS</code> to indicate that an object is obstructing one or both beams, or • 0, if neither beam is obstructed |
| <code>ayIntPort(GripperState,0);</code> | Position of the optional gripper, one of: <ul style="list-style-type: none"> • <code>ayGRIPATCARRY</code> - closed, at carry position • <code>ayGRIPATOPEN</code> - open, at bottom • <code>ayGRIPATTOP</code> - closed, at top • <code>ayGRIPMOVING</code> - between positions |
| <code>ayIntPort(GripperContact,0);</code> | Contact sensors of the optional gripper. One of: <ul style="list-style-type: none"> • <code>ayGRIPCONTACT</code> • 0 (no contact) |
| <code>ayIntPort(XModButton,0);</code> | State of the button on the optional expansion module. One of: <ul style="list-style-type: none"> • <code>ayBUTTONPRESSED</code> • 0 |
| <code>ayIntPort(XModSwitch,0);</code> | Position of the switch on the optional expansion module. One of: <ul style="list-style-type: none"> • <code>aySWITCHUP</code> • <code>aySWITCHDOWN</code> |
| | |

| Ports Used for Input | |
|---|--|
| <code>ayIntPort(ResetPosition, 0);</code> | Resets the Pioneer's coordinate system so that the current position becomes (0, 0) with heading 0. |

5.3 Universal Sonar Interpreter

Prototype: `ayDefBehaviorClass(ayiUniversalSonarMaker);`

Standard instance name: USI

| Ports Used for Output | |
|--|----------------------------------|
| <i>"Universal" Sonar Readings</i> - these give the minimum sonar reading in a generalized direction relative to the robot's heading. Each is composed of a combination of raw sonar readings. This behavior allows portability of programs between robot platforms - the generalized directions are correct for both Pioneer 1 and Pioneer 2 sonar configurations. | |
| <code>ayIntPort(Front, 0);</code> | Minimum of forward-facing sonars |
| <code>ayIntPort(LeftFront, 0);</code> | Minimum of left front sonars |
| <code>ayInttPort(RightFront, 0);</code> | Minimum of right front sonars |
| <code>ayIntPort(Left, 0);</code> | Minimum of left-facing sonars |
| <code>ayIntPort(Right, 0);</code> | Minimum of right-facing sonars |
| The following three ports are only active on Pioneer 2 models equipped with rear sonars. Thus, for truly portable code, it is a good idea to check if these ports are outputting information before using them (or make sure the initial value of any port they are connected to is useful if they are never written). | |
| <code>ayIntPort(Rear, 0);</code> | Minimum of rear-facing sonars |
| <code>ayIntPort(LeftRear, 0);</code> | Minimum of left rear sonars |
| <code>ayInttPort(RightRear, 0);</code> | Minimum of right rear sonars |

5.4 Vision Packet Parser

Prototype: `ayDefBehaviorClass(ayiVPParserMaker);`

Standard instance name: VPP

| Ports Used for Output | |
|---|---|
| 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"> • <code>ayBLOBMODE</code> - only color-blob information is provided • <code>ayBBOXMODE</code> - color-blob and bounding-box information is provided • <code>ayLINEMODE</code> - only line mode information is provided |
| <code>ayIntPort(ChannelBMode, 0);</code> | Mode of channel B |

| | |
|---|---|
| <code>ayIntPort(ChannelCMode, 0);</code> | Mode of channel C |
| <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. | |
| <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) |
| <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. | |
| <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 |
| <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. | |
| <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 |
| Ports Used for Input | |
| <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. |

5.5 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) |

5.6 Low-Level Communication

5.6.1 Pioneer Robot Starter

Prototype: `ayDefBehaviorClass(ayiRobotStarter);`

Standard instance name: **RW**

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

5.6.2 Pioneer Packet Receiver

Prototype: `ayDefBehaviorClass(ayiPacketReceiverMaker);`

Standard instance name: **PPR**

| Ports Used for Output | |
|---|--|
| <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 |
| <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. |

5.6.3 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. Instead, there is a packet queue from which the PPS sends packets when time permits (if there are packets waiting, a minimum of two will be sent each second from the queue; as many as ten per second may be sent in the absence of other activity). The access function for this queue is `ayEnqueueGP` (Section 4.4). If the queue grows large due to too-frequent enqueueing, a warning will be issued.

| Ports Used for Input | |
|--|--|
| <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 |

6. Extended Examples

6.1 An Ayllu RoboCup Soccer System

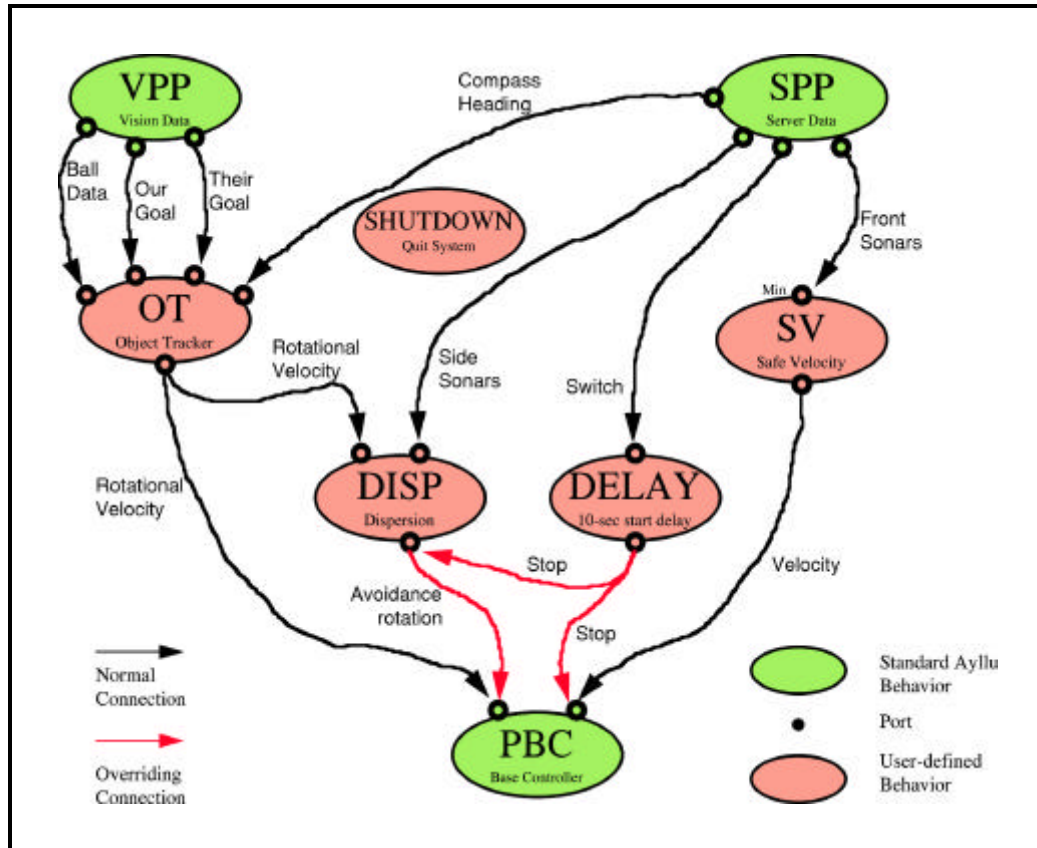


Figure 7: Behavior diagram of RoboCup soccer system

/***** 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>

Note that this code does not take advantage of Ayllu's interrobot communication; it allows non-networked robots to cooperate as a team. Watch the website above for future versions that take advantage of Ayllu network communication.

```

                                                                    */
/* 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,
                tgsz, 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); /* Reset MinPort */
  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(); /* This is the Macintosh function - */
  return;                          /* use a similar non-blocking fn   */
}

```

```

}                                     /* for other systems */
ayDefBehaviorClass(Stopper)
{ ayINTERFACE { }
  ayPROCESSES { ayInitProcess(stopper,ratepersecond(20));}
}

int main()

{
  ayInitPioControl("modem");

  ayInitBehavior(SafeVelocity, SV);      /* Instantiate behaviors */
  ayInitBehavior(Dispenser, 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, tgsizes); /* Goals are marked with */
  ayConnect(VPP, C_BlobArea, OT, ogsizes); /* 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 kbd input says to stop */

return 0;
}

```

6.2 Helicopter Tracking: Multi-Robot Coordination Strategies

Ayllu allows for a wide range of control strategies for multi-robot teams. It is clearly possible to have a traditional planner with an AylluLite interface activating and parameterizing behaviors on each robot; it is equally possible to build fully distributed systems of robots running Ayllu which do not communicate at all (as in the Soccer system of Section 6.1). Ayllu is targeted primarily, however, at fully distributed communicating systems, and the rest of this section will present some novel approaches to distributed control specifically facilitated by Ayllu.

Ayllu allows arbitrarily scalable dynamic task allocation through broadcast of various types of messages. Basically, robots are able to determine if they are the best-suited to a task, and either prevent others from performing the same task or notify others of their self-assignment to a particular task. We give as an example a task involving a robotic helicopter and some number of Pioneers on the ground. Among other tasks, the group of ground robots must keep one of its members directly below the helicopter at all times. As Figure 4 illustrates, each robot keeps a measure of its distance from the helicopter at all times, and broadcasts this distance to the rest of the group. By extracting the minimum of the broadcasts received and comparing it to its own distance, the robot closest to the helicopter can identify itself. When it does so, it activates a behavior that causes it to stay below the helicopter, and at the same time inhibits the same behavior in the other robots. If, however, another robot at some point maintains a position closer to the helicopter for some period of time (due to failure or blockage of the first robot, or the ground robots' inability to match the speed of the helicopter), this new robot will take over tracking, freeing the first robot for another task.

Strategies need not be so complex to have efficient task distribution. It would also have been possible for the first robot to perceive the helicopter to inhibit tracking in all the other robots, without the need for the determination of closeness (or even GPS data); this is preferable in tasks where continuity of roles is more important or feasible. In this case, if the robot failed or lost the helicopter, it would stop inhibiting the others, and at some point another robot would perceive the helicopter, track it, and in turn inhibit the others.

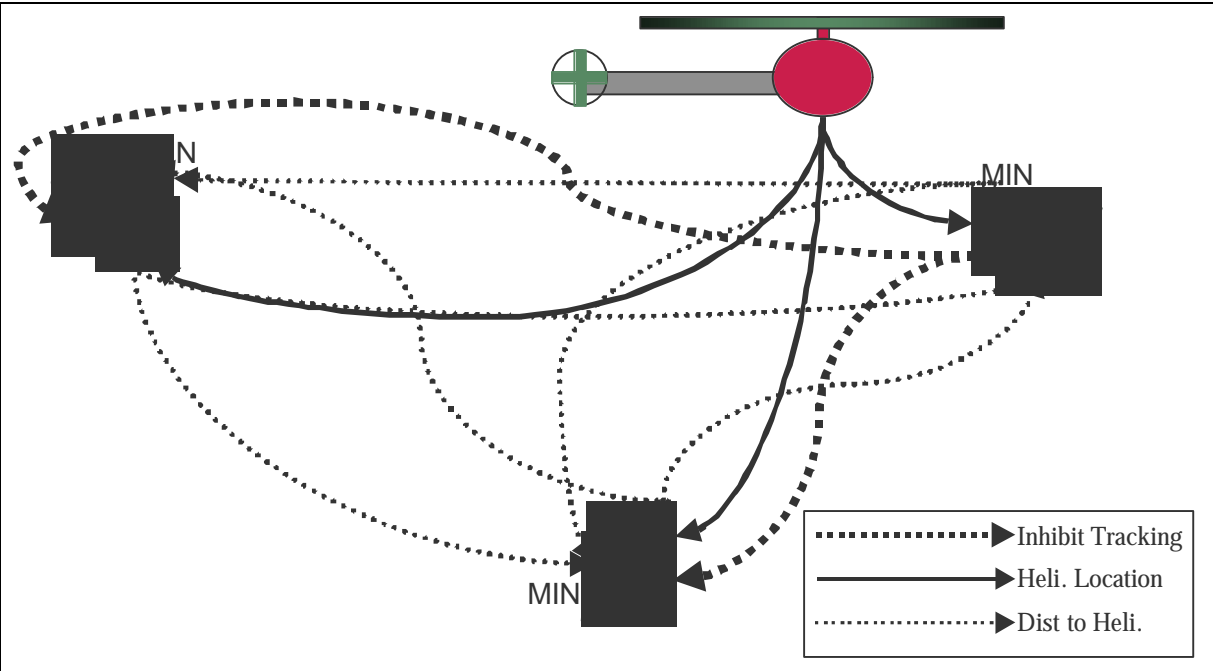


Figure 8: Communication Diagram of the Helicopter-Tracking System

6.2.1 Behavior Definitions

```
/* Behavior MinSelector - This is a general behavior that outputs a true value
   if its input MyValue is smaller than the minimum of all values written to its
   port Criterion
*/
ayDefProcess(ActivateIfMin)
{ ayLocalPort Criterion, MyValue, Activator, Inhibitor;
  ayLocalMono Activating;

  if (ayPortWritten(MyValue)) /* if not getting data, */
      /* there's a problem. Don't activate */
      /* that way, someone'll take over task */
      if (ayReadIntPort(MyValue) <= ayReadIntPort(Criterion))
          ayTrigger(Activating);

  ayWriteIntPort(Criterion, ayMAXINT); /*reset minimum */

  /* Activate iff monostable is active. */
  /* monostable smooths over glitches */
  ayWriteIntPort(Activator, ayMonoActive(Activating));
  if (ayMonoActive(Activating))
      ayWriteIntPort(Inhibitor, 1);
}

ayDefBehaviorClass(MinSelector)
{ ayINTERFACE
  { ayMinIntPort(Criterion, ayMAXINT);
    ayIntPort(MyValue, ayMAXINT);
    ayIntPort(Activator, 0);
    ayIntPort(Inhibitor, 0);
    ayMonostable(Activating, seconds(3));
  }
  ayPROCESSES
  { ayInitProcess(ActivateIfMin, ratepersecond(1)); }
}

ayDefProcess(DistanceCalc)
{ ayLocalPort x1, y1, x2, y2, dist;

  if (ayPortWritten(x1) && ayPortWritten(y1)
      && ayPortWritten(x2) && ayPortWritten(y2))
      /* Calculation is no good if we don't */
      /* have all data...*/
      ayWriteIntPort(dist, ayDistance(ayReadIntPort(x1), ayReadIntPort(y1),
                                      ayReadIntPort(x2), ayReadIntPort(y2)));
}

ayDefBehaviorClass(Distance)
{ ayINTERFACE
  { ayIntPort(x1, 0);
    ayIntPort(y1, 0);
    ayIntPort(x2, 0);
    ayIntPort(y2, 0);
    ayIntPort(dist, 0);
  }
  ayPROCESSES
```

```

    { ayInitProcess(DistanceCalc, ratepersecond(2)); }
}

```

6.2.2 Connections

Assuming that the behaviors *GoTo* (moves towards specified position, such as in Section 2.5), *Wander* (random walk), and *GPS* (reports latitude and longitude) are instantiated on the ground robots, and *GPS* is instantiated on the helicopter (*HELI*):

```

ayInitBehavior(Distance, DistToHeli);
ayInitBehavior(MinSelector, TrackIfClosest);

ayConnectFromRemote(HELI, GPS, Lat, DistToHeli, x1);
ayConnectFromRemote(HELI, GPS, Long, DistToHeli, y1);
ayConnectFromRemote(HELI, GPS, Lat, GoTo, Latitude);
ayConnectFromRemote(HELI, GPS, Long, Goto, Longitude);
ayConnect(GPS, Latitude, DistToHeli, x2);
ayConnect(GPS, Longitude, DistToHeli, y2);
ayConnect(DistToHeli, Dist, TrackIfClosest, MyValue);
for (i=0; i<n; i++)
    if (ayMyIPNum() != ayIPNum(groundbots[i]))
    {
        groundbots[i] = ayMakeRemoteHost(hosts[n]);
        ayConnectToRemote(DistToHeli, Dist,
            groundbots[i], TrackIfClosest, Criterion);
        ayInhibitRemoteOut(TrackIfClosest, Inhibitor,
            groundbots[i], TrackIfClosest, Activator, seconds(10));
    }
ayConnect(TrackIfClosest, Activator, Goto, Active);
aySuppressIn(GoTo, Active, Wander, Active, seconds(1));

```

If this code is running on all of the ground robots, the one closest to the helicopter will activate the *GoTo* behavior to stay beneath the helicopter, at the same time inhibiting the other robots from doing the same. Should there be a failure of the "closest" robot, or should by serendipity or obstacles place some other robot closer for a significant amount of time (here, 10 seconds), the new closest robot will take over, inhibiting the first one.

7. Index

A

AbsHeading
 port (in PBC) á 37
 status of degree-based motion á 38

AbsPan
 port (in PTZ) á 42

AbsTilt
 port (in PTZ) á 42

Analog input
 selection of input pin á 36

AnalogIn
 port (in SPP) á 39

AngularVel
 port (in PBC) á 37

ArcRadius
 port (in PBC) á 37

ayDefBehaviorClass á 27

ayEnqueueGP
 packet construction á 36

ayFloatMaxPort á 29

ayFloatMinPort á 29

ayFloatPort á 29

ayFloatSlot á 30

ayFloatSumPort á 29

ayInitBehavior á 27

ayInitProcess á 27

ayINTERFACE á 27

ayIntMaxPort á 29

ayIntMinPort á 29

ayIntPort á 29

ayIntSlot á 30

ayIntSumPort á 29

Ayllu
 as a language á 7, 27

ayLocalMono á 30

ayLocalPort á 30

ayLocalSlot á 30

ayMakeADSelCommand á 36

ayMakeDigOutCommand á 36

ayMakePSOSIntCommand á 36

ayMakePSOSNoargCommand á 36

ayMakePSOSSingCommand á 36

ayMakePSOSStrCommand á 36

ayMakePSOSStrnCommand á 36

ayMakeVisionCommand á 36

ayMemPort á 29

ayMemSlot á 30

ayMonoActive á 30

ayMonostable á 30

ayPortPriority á 31

ayPortWritten á 31

ayPrioritize á 29

ayPROCESSES á 27

ayReadFloatPort á 31

ayReadIntPort á 31

ayReadMemPort á 31

ayReadStrPort á 31

aySetPortPriority á 31

ayStringPort á 29

ayStringSlot á 30

ayTrigger á 30

ayWriteFloatPort á 32

ayWriteFloatPriority á 32

ayWriteIntPort á 32

ayWriteIntPriority á 32

ayWriteMemPort á 32

ayWriteMemPriority á 32

ayWriteStrPort á 32

ayWriteStrPriority á 32

B

beep á 36

beh á 9

behaviors
 definition á 27
 instantiation á 27
 peer á 9
 remote á 9
 what is an Ayllu behavior á 8

C

ChannelAMode
 port (in VPP) á 40

Compass
 port (in SPP) á 38

connections
 inhibitory á 8
 peer á 9
 remote á 9
 suppressive á 9
 what is a connection? á 8

ControlMode
 port (in PBC) á 37

D

data availability á 31

data-driven programming á 31

DefaultVel
 port (in PBC) á 37
degree-based motion á 38
delayseconds á 27
destbeh á 9
desthost á 9
Digital Output
 packet construction á 36
DigitalIn
 port (in SPP) á 39
DigitalOut
 port (in SPP) á 39
Distance
 port (in PBC) á 37
 status of distance-based commands á 38
distance-based motion á 38

E

event-driven programming á 31
extrema
 ports á 29

G

Gripper
 port (in PBC) á 37
GripperBeams
 port (in SPP) á 39
GripperContact
 port (in SPP) á 39
GripperState
 port (in SPP) á 39

H

Heading
 port (in SPP) á 38, 40
host á 9

I

incoming connections á 9
inhibitory
 connections á 8
InputTimer
 port (in SPP) á 39

L

LeftWheelVel
 port (in PBC) á 37
lightweight process á 28

LineBottomRow
 port (in VPP) á 40
LineMinMass
 port (in VPP) á 40
LineNumSlices
 port (in VPP) á 40
LineSliceSize
 port (in VPP) á 40
local
 declarations á 30
 monostables á 28, 30
 ports á 28, 30
 slots á 28, 30
 use of term á 9
localmono á 9
localport á 9
localslot á 9
LWheelVel
 port (in SPP) á 38

M

message
 prioritized á 9
messages
 priority of á 32
 propagation of á 8
 unreliability of á 31
monostables
 declaration á 30
 local á 9, 28, 30
 what is a monostable? á 8

N

name á 9

O

OtherPacket
 port (in PPR) á 42
outgoing connections á 9

P

P2OS
 packet construction á 36
packets
 construction á 36
 types á 36
PanAngle
 port (in PTZ) á 42
Pan-Tilt-Zoom Camera á 41
PBC á 9, 50, 51

- Pioneer Base Controller á 9
- peer behavior á 9
- Pioneer robot
 - Standard Behaviors for á 9
- port priority á 29
- ports
 - data availability á 31
 - extrema á 9, 29
 - local á 9, 28, 30, 32
 - peer á 9
 - priority á 9, 31, 32
 - reading á 31
 - remote á 9
 - summation á 9, 29
 - what is a port? á 8
 - writing to á 32
- PPR
 - Pioneer Packet Receiver á 10, 42
- PPS
 - Pioneer Packet Sender á 10, 43
- priority
 - of messages á 9, 32
 - of ports á 29, 31, 32
- priority ports á 29, 31
- processes*
 - definition á 28
 - instantiation á 27
 - lightweight á 28
 - reentrant á 28
- propagation á 8
- prototypes
 - behavior á 27
 - process á 28
- PSOS
 - packet construction á 36
- PTU
 - port (in SPP) á 38
- PTZ
 - Pan-Tilt-Zoom Camera Controller á 10

Q

- Quechua language á 8

R

- ratepersecond* á 27
- reentrant process* á 28
- RelHeading
 - port (in PBC) á 37
 - status of degree-based motion á 38
- RelPan
 - port (in PTZ) á 42
- RelTilt
 - port (in PTZ) á 42
- remote
 - behavior á 9

- port á 9
- remote host á 9
- ResetPosition
 - port (in SPP) á 40
- RobotClass
 - port (in SPP) á 42
- RobotName
 - port (in SPP)
 - port (in SPP) á 42
- RobotSubclass
 - port (in SPP) á 42
- RotInProgress
 - port (in PBC) á 38
- RotPacket
 - port (in PPS) á 43
- Running
 - port (in PRS) á 42
- RW
 - Pioneer Robot Starter á 10, 42
- RWheelVel
 - port (in SPP) á 38

S

- scoping
 - differences from C á 27
- semantics
 - differences from C á 7, 27
- Sing
 - packet construction á 36
- slots
 - declaration á 30
 - local á 9, 28, 30
 - what is a slot? á 8
- SonarN
 - port (in SPP) á 38
- speaker á 36
- SPP á 9, 10, 50, 51
 - Server Packet Parser á 9
- srcbeh* á 9
- srchost* á 9
- Stalls
 - port (in SPP) á 38
- status of degree-based motion á 38
- status of distance-based motion á 38
- Stop
 - port (in PBC) á 37
- summation
 - ports á 29
 - summation ports á 9
- suppressive
 - connections á 9
- syntax
 - differences from C á 7, 27
- system
 - definition of á 8

T

terminology á 9
TiltAngle
 port (in PTZ) á 42
TransInProgress
 port (in PBC) á 38
triggering
 of monostables á 8

U

unreliability
 of messages á 31
User I/O
 analog input á 36
 digital output á 36

V

Velocity
 port (in PBC) á 37
VelPacket
 port (in PPS) á 43
Vision Commands
 packet construction á 36
Vision packets á 36

VisPacket
 port (in PPS) á 43
Voltage
 port (in SPP) á 38
VPP
 Vision Packet Parser á 10, 50

X

XModButton
 port (in SPP) á 39
XModSwitch
 port (in SPP) á 39
XPos
 port (in SPP) á 38, 40

Y

YPos
 port (in SPP) á 38, 40

Z

Zoom
 port (in PTZ) á 42