1 Getting Access

1.1 Source Code

Before you can start work on the AIBOs, you need to obtain a copy of the robot control software. The software is stored in a CVS code repository which keeps track of old versions. Checking out a copy of robot code using CVS consists of setting an environment variable and then issuing the checkout command.

- Your account will most likely be set up correctly to check out the source code. To get the code, execute the following command:

  ```
  cvs checkout dogs
  ```

- If your account is not set up properly and the above command fails, you can specify where cvs needs to go in order to obtain its files. To checkout code from a repository located on your local machine, use the following command:

  ```
  cvs -d /data/dogs/code_repository checkout dogs
  ```

- To checkout code from a different computer, execute the following commands:

  ```
  export CVS_RSH=ssh
  ```

  ```
  cvs -d :ext:<server name>:/data/dogs/code_repository checkout dogs
  ```

Where `<server name>` is the name of the machine that contains the robot repository. Contact Scott Lenser to get access to the machine that has this repository.

If you get a permission denied error, please contact Scott Lenser to fix the problem.

The dogs directory contains the following files and directories:

- CSV Makefile agent docs util

- The directory CVS contains information about the version control systems and should be left alone.
- The agent directory contains the files necessary to compile the AIBO control program.
- The docs directory will contain information on how to use the AIBOs (like this document).
- The util directory contains many different utilities for loading code onto the AIBOs, as well as for debugging their software.
1.2 Maintaining Your Code Repository

If you are using a laptop or another shared computer to work on code that is located on the gs219 server, you will want to copy the contents of your home directory to the laptop. Once you are done with your modifications, you will want to copy your home directory back to the server. To do this, you will use the commands:

\texttt{gethomedir}

to copy the contents of your home directory on the server and,

\texttt{sendhomedir}

to return the contents of your home directory back to the server.

\textbf{IMPORTANT:} If you execute \texttt{gethomedir} but your local directory has a files in it that are not in the server, then those files will be renamed and their names will end with a \texttt{~} character. If you execute the command \texttt{gethomedir} again, all files ending with a \texttt{~} character will be deleted. The same effect will occur if you execute \texttt{sendhomedir} and the server has files that are not located in your local home directory.

1.3 Sony Documentation

The software development kit (SDK) for the AIBOs is provided from Sony. There are a number of web sites that you can go to in order to get more documentation about the software.

- The Sony RoboCup support information can be found at the following site:

  \texttt{https://www.openr.org/page1\_2003/index.html}

- The Sony OPENR-SDK support information can be found at the following site:

  \texttt{https://www.jp.aibo.com/openr/e\_regi/index.jsp}

1.4 Multi-Robot Lab Documentation

The remainder of the publications, documentation, and internal lab information can be found on gs104.sp.cs.cmu.edu in the following directories (not through CVS):

/\texttt{data/dogs/docs/papers/}
/\texttt{data/dogs/docs/doc/}

Please talk to Scott Lenser about getting access to this information if you are interested.
2 Compiling the AIBO Control Software

2.1 Initial Setup

- Set the DOGROOT environment variable to point to the location where you checked out the AIBO code directory. So, if you put it into your home directory, execute the following command:

  ```
  export DOGROOT=˜/dogs
  ```

  if you put the files into a subdirectory (or inside nested subdirectories), be sure to add this subdirectory to the environment variable:

  ```
  export DOGROOT=˜/<subdirs>/dogs
  ```

- Next, add the following directories to your path variable with the command (Note: Entries should all appear on the same line. They are separated here for readability):

  ```
  export PATH=/usr/local/OPEN_R_SDK/bin:
  /usr/local/OPEN_R_SDK/RP_OPEN_R/bin:
  ${DOGROOT}/util/bin:${PATH}
  ```

  These commands can be put into your .zshrc file (if they’re not there already) so that they are executed every time you open a new shell.

2.2 Compiling

To compile the code, just go into the dogs directory and type

```
make
```

If you get errors about unable to find mipsel-*, either your path is incorrect or the machine is set up wrong. Run the command

```
echo $PATH
```

and make sure that /usr/local/OPEN_R_SDK/bin is there. If not, re-run the above export commands to put it there. If you get errors about unable to find dep_process, your path is missing

```
${DOGROOT}/util/bin
```

and you need to re-run the above export commands to put it there.
3 Running the Dogs

3.1 Configuration

After the code in the agents directory has compiled, it must be loaded onto a memory stick. In addition to the AIBO runtime image, the memory stick also contains various parameter files that the AIBO binary will need to access during its execution.

Preparing the memory stick consists of the following steps:

- Setting the behavior that the robot will execute
- Configuring the network address of the memory stick so that it matches the robot’s hardware
- Copying the binary to the stick

3.1.1 Selecting the Behavior to Run

The AIBO control software contains many different behaviors that the robots can run. By allowing the binary to select which behavior to run based on the configuration of the memory stick, a robot can be quickly reconfigured for a new behavior without requiring that the binary be recompiled.

1. Put a memory stick into a USB card reader.

2. Behaviors can be configured either manually on the memory stick or by having stickit automatically copy a configuration file to the memory stick:
   
   To manually configure the behavior, follow the following steps
   
   (a) Execute the command
   
   mount /memstick
   
   (b) Execute the command
   
   cd /memstick/config
   
   (c) Edit run.cfg so that it contains the desired behavior names and priorities.
   
   (d) cd to a directory that is not on the memstick
   
   (this avoids “Device is busy” problems when unmounting)
   
   (e) Stop the stick access by running the command
   
   umount /memstick
   
   NOTE: The stickit -c and stickit -a commands will overwrite run.cfg with the contents of agent/config/config/run.cfg. If you want to make a long term change to the behavior, edit the file on agent/config/config and use stickit to transfer it to the memory stick.

3. Remove the memory stick from the reader.

3.1.2 Selecting the Robot’s Network Configuration

- Determine the host name of the robot being used. There should be a two digit number printed on the robot’s back (01, 02, 03, ...). The robot’s hostname will be ersXY where XY is the robot’s number.

- Configure the stick for the specific robot IP address. Run the program wlan_setup and specify only the host name (the domain name is not needed) of the robot:

  wlan_setup -i ersXY

  where, once again, XY is the two digit number found on the back of the robot.
3.1.3 Copying the Code to the Stick

Now that the configuration files on the stick are properly configured, you’ll need to copy your binary to the stick so it can be executed by the AIBO. Do the following:

1. Put the memory stick in the card reader.

2. Download your code to the memory stick using the stickit program (which should be in your path). Stickit has three options (you must specify at least one):
   - `-a` (copy everything)
   - `-b` (copy binaries/executables)
   - `-c` (copy configuration files)

   Normally, you’ll only ever need to execute the following:

   ```bash
   stickit -a
   ```

   Executing this command will not change your network or behavior configuration settings.

3. Remove the memory stick from the reader. Stickit starts and stops the card reader drivers as necessary, so you do not need to execute the `mount` or `unmount` commands at all.

3.2 Running Your Program on the Robots

1. Put the memory stick and a charged battery in a dog

2. Turn on the robot by pressing chest button and wait ~30 seconds for dog to boot. If the robot shuts itself off immediately, the battery is most likely low and you should try one that is fully charged.

3. To start the robot, press back button.

4. Watch as the AIBO does amazing things.

5. Stop the robot by pushing its chest button and wait for activity to cease.

6. Take out memory stick, and robot will shut itself off.

Operating notes:

- If while running the robot, you want to pause its motions, push and hold the back button for 1 second (push again to unpause). If you hold it too long, the robot will unpause again. You can tell when you have held it long enough when the tail twitches.

- Taking out the memory stick while the robot is running will also shut it down. Be very careful when doing this if the legs are in motion. It’s easy to pinch your fingers in the legs and while this shouldn’t do any permanent damage, it’s likely to hurt a lot. The best place to grab the robot is from the center of its back. This lets you turn the robot upside down and access the memory stick while keeping your fingers away from the legs.

3.3 Monitoring the State of the Robot

The robot can be configured to output debugging information that can be captured and displayed on a workstation. The utility to do this is called `chokechain` and it should have been built when you invoked make at the top level of the robot’s source directory.
3.3.1 Configuring Debug Output

Edit the file /memstick/config/spout.cfg on the memory stick. Set the values of the variables to 0 to turn that debugging information off and set them to 1 to turn it on.

3.3.2 Running the GUI

- Run chokechain with the command
  
  chokechain

- Run the network server with the command:

  wlan_connect ersXY.wv.cs.cmu.edu

  where XY is the number on the back of the dog. Whatever debug information you configured in the spout.cfg file should be displayed on the screen as well as on the text console that invoked chokechain.

4 Debugging

Debugging embedded systems such as the AIBOs is typically much more difficult than debugging programs written for Windows or *nix machines. For one, there is no visual debugger that allows one to easily step through instructions that are being executed. There are two methods of debugging that can be used:

**Viewing live output** When monitoring the execution of your program, you will often want to view the values of variables, or to graphically display robot vision or location information while the dog is running its program. When you are running chokechain, the output from printf statements will appear in the console window that invoked the program. To enable various debug information, mount the memory stick and edit the file /memstick/config/spout.cfg. Variables set to 1 will output debug information, otherwise no debug information will be set.

**Viewing saved output** All debug information is cached in the robot’s memory. This information can be dumped off to a log file that can be viewed off-line. To collect debug data for off-line viewing, do the following:

1. Set the debug information to view in the /memstick/config/spout.cfg file.
2. Run the AIBO around and collect data.
3. Push the back head button. This will cause the robot to save its data to the log file and to shut itself down.
4. Extract the memory stick and mount it. There should now be a file called /memstick/log.
5. Execute the command
   
   logextract /memstick/log $DOGROOT/util/log_extractor/extract_configs/<config>
   
   where <config> is one of:

   (a) ball_loc.cfg
   (b) ball_vis_model.cfg
   (c) gsens_move_foot.cfg
   (d) gsensor.cfg
   (e) headers.cfg
Analyze a stack dump  If your program executes an illegal instruction that causes the binary to crash, the robot will shut itself down. You will need to determine what part of your code caused the robot to crash and the correct the error. Do the following steps to debug the robot’s code.

1. Edit agents/Makefile.sdk.common and remove the -02 option from the CXXFLAGS variable. Do a make clean inside the agent directory and recompile with make.
2. After the compilation has finished, re-load your program and re-run it on the dog. When the robot crashes, pull the memory stick out and put it in your memory stick reader.
3. Mount the stick with the command mount /memstick
4. Execute the command

  emonParser /memstick/open-r/emon.log

to obtain a stack trace of your program. For instance, if your program crashed just after starting, you might see an output that looks like:

  context: 801c8420
  object:  mainObj

This means that the program crashed while executing the function mainObj at address 0x801C8420. To determine what happened in more detail, go to the dogs/agent/Main directory and execute the following command:

  emonParser /memstick/open-r/emon.log mainObj.nosnap.elf

This command may take a while, but it will produce a stack dump that should look something like this:

  context: 801c8420
  object:  mainObj
  badvaddr: 00001708
  epc: 809e4848
  ra: 80a0d4f0
  target address: 809e4848 (epc)
  gp: 80b1bec0
  _gp: 00579be0
  static addr: 00442568
  finding name for static addr: 442568
  symbol: 00442540 T SensorData::getFrame(int)
  spReg: 8051f710
  raReg: 80a0d4f0
  finding name for static addr: 442568
  stack level 0: 00442540 T SensorData::getFrame(int)
finding top of function
    found at: 00442540 (idx 113943)
finding return address
ra: 80a0d4f0 (46b210)
Warning: Couldn’t find change to stack offset, assuming 0
finding name for static addr: 46b210
stack level 1: 0046b178 T FootDog::operator()(FeatureSet*, Motion::MotionCommand*)

finding top of function
    found at: 0046b178 (idx 213266)
finding return address
ra_offset: 80 (0x00000050)
ra: 80a0dcf4 (46ba14)
stack_add: 88 (0x00000058)
finding name for static addr: 46ba14
stack level 2: 0046b964 T FootDog::get(unsigned long)

finding top of function
    found at: 0046b964 (idx 214125)
finding return address
ra_offset: 40 (0x00000028)
ra: 80a05538 (463258)
stack_add: 48 (0x00000030)
finding name for static addr: 463258
stack level 3: 0046313c T BehaviorManager::get(unsigned long)

finding top of function
    found at: 0046313c (idx 198016)
finding return address
ra_offset: 96 (0x00000060)
ra: 80a03a30 (461750)
stack_add: 104 (0x00000068)
finding name for static addr: 461750
stack level 4: 0046124c T RobotMain::processImage(unsigned long, unsigned char const*, unsigned char const*, unsigned char const*, int, int)

(...and it keeps going...)

Now you can see that the crash occurred while executing SensorData::getFrame(int) which was called from FootDog::operator()(FeatureSet*, Motion::MotionCommand*) which was called from FootDog::get(unsigned long), and so on...

Although most crashes occur in MainObject, other objects such as Logger or Motion can also crash. In those cases replace mainObj.nosnap.elf with a different appropriate file such as motion.nosnap.elf in the Motion directory.
A Using UNIX

A.1 Basic Commands

`ls` List the contents of a directory.

`cd <dir>` Change the current directory.

`pwd` Print the current directory.

`less <file>` View a text file.

`xemacs <file> &` Start the xemacs editor in a separate window on a file.

`xemacs -nw <file>` Start the xemacs editor in the current window on a file.

`man <prog name>` Get help on a program, output will come up in a pager.

`rm <file name>` Remove a file.

A.2 Using a pager

Here is a list of basic commands to use the common pagers `less` and `more`.

`<space>` Advance to next screen full of information.

`b` Go back one screen full of information.

`q` Quit pager and return to shell.

`<return>` Advance one line.

`/<regular expression>` Search for next instance of the regular expression. You can search for strings containing only letters, numbers and underscores simply by using the string as the regular expression.

A.3 Using a Shell

These instructions assume that you are using zsh. If you want to use another inferior shell, you are on your own. You can check what shell you are currently running by running `ps`. The item it lists that ends with “sh” is the shell you are currently using.

To change your shell, run `chsh`. Type in your password when prompted. When prompted for the new shell, enter `/usr/bin/zsh`.

To set an environment variable, the command is `export <var-name>=<value>`. This sets the value for the current shell.

To set an environment variable for all new shells that you create, add the line `export <var-name>=<value>` to the file `~/.zshrc`.

To avoid typing the same command over again, you can use up and down arrow to cycle through the previous commands you have entered.

You can type `<tab>` in many situations at the shell to complete things for you or get a list of possible completions in case it is ambiguous.

A.3.1 UNIX Paths

Unix paths all start at the root directory (`/`). The current directory is indicated by period (`.`). The parent of the current directory is indicated by two periods (`..`). The components of a path are separated by forward slashes (`/`).

9
A.4 Using Xemacs or Emacs

Type text using the keyboard. Use the arrow keys and such to move around.
Type C–x C–s to save the current file (or use the button or menu options). C–x means press the “Ctrl” key, press the “x” key, release the “x” key, then release the “Ctrl” key.
Type C–x C–c to quit xemacs.

A.5 Using CVS

A.5.1 Setup

Add the following lines to your /.cvsrc file.

update -dP
cHECKOUT -P

A.5.2 CVS Commands

**checkout** <module> or **co** <module> Checks out the code module specified. Use this to get an initial copy (called a working copy) of a piece of code.

**update** Updates the working copy recursing from the current directory to incorporate the latest changes in the repository. CVS will attempt to merge the changes in the repository with whatever local changes you have. CVS will display a “C” next to filenames that had conflicts. You will have to manually edit these files to resolve the conflicts. Search for “==” to find the conflict markers.

**diff** Displays differences between files under CVS control. Without options, displays all changes you have made locally to the version you checked out (Note: this will not display anything about changes other people have made to the repository since you checked out.). This command can also show differences between your working copy and any version in the repository and between any two revisions in the repository.

**commit** <filename1> <filename2> Commits the files specified to the repository where other people can get the changes using **update**. If no files are specified, . is used. Directories are recursed to pick up changes to be committed. If you have not updated to the latest version of the repository before you try to commit, CVS will print out an error and abort. CVS will bring up an editor after determining what files need committed to allow you to enter a log message. Type a message in the editor describing the changes you made and then save the file and exit the editor. Make sure to use a descriptive commit message.

**log** <filename> Displays the commit messages that have been entered for this file which describe the changes done. Also displays some summary statistics about the changes performed on the file.

**add** <filename> Add file or directory to the repository. Files are added when they are committed, directories are added immediately. Please be sure of the need for a new directory before adding a new directory to the repository.

**remove** -f <filename> Remove the file from the working copy and schedule the file for deletion in the repository. There is no way to recover the local file after performing this operation. There is no way to remove directories. If the -P option is passed to **update/co**, empty directories in the repository will be removed from the working copy on update. Without the -f option, the local file must already be removed and the command schedules the file for deletion in the repository.
A.5.3 Common CVS Options

Here are some options used before the CVS command:

-n Fake running the command. Generate all output as normal but don’t actually do anything.

-d <repository> Specify a repository to use when performing the CVS command. This option is usually not needed because CVS defaults to using the repository location stored in the working copy.

Here are some common options used by CVS commands:

-r <rev> Used to specify a revision.

-D <date> Used to specify a date for a revision.

-d Specifies that update should add directories if necessary.

-P Specifies that update and checkout should prune empty directories.

A.5.4 CVS Environment Variables

Here are some environment variables used by CVS:

CVS_RSH By default CVS uses the insecure protocol rsh to connect to :ext: repositories. Setting this to ssh makes CVS use the ssh protocol.

CVSROOT Specifies the location of the repository. This variable usually doesn’t need to be set since it is determined automatically from the working copy.

A.5.5 How to Perform Common Tasks

Here are some common tasks that are done and the way to do them using CVS:

Creating a working copy
cvs -d <repository loc> co <module name>

Finding out what changes other people have made
cvs -n update. This lists the effect that an update command will have on the working copy. The letters in front of the file names mean:

M Modified locally.
U Modified in repository.
P Modified in repository.
C Modified locally and in repository.
? Local file not present in repository.
A Locally added file.
D Locally deleted file.

Find out what changes you have made. Use cvss -n update to find any files you have added/deleted that need to be scheduled for addition/deletion from the repository. Use cvss diff to determine what changes you have made to each file.

Move a file. Use mv <old-name> <new-name> to move the file. Use cvss remove <old-name> to remove the old file name from the repository. Use cvss add <new-name> to add the new file name to the repository. Include a note in the commit message mentioning that the file was moved and the new and old names.

Remove a directory. Remove all of the files in the directory. CVS does not directly version directories.

Commit changes. Use cvss commit <filename1> <filename2> to commit your changes. The filenames can be directories in which case they will be recursed upon to look for changes. You can omit the names to indicate the current directory.
A.6 Using ssh/scp

We do not use telnet or ftp on our machines because they are insecure protocols (unless using kerberized version) which send passwords and other sensitive information in cleartext. Instead, we use the ssh tools ssh and scp.

ssh is a secure replacement for telnet that also securely forwards X11 requests. This allows access to remote shells and the ability to run X11 programs on the remote machine with the display sent to the local machine. To connect to a remote machine use ssh [<user>@]<machine name>. The user name can be omitted if it the same on both machines. The brackets indicate an optional part of the command.

scp is a secure alternative to ftp. scp uses a syntax similar to cp. To specify a remote file use the syntax [<user>@]<machine>:<filename>. The user name can be omitted if it is the same on both machines. The filename is a path relative to your home directory unless it starts with a slash (in which case it is relative to the root directory). The rest of the syntax is the same as cp. See man scp for more details.

A.6.1 Using Public/Private Key Authentication

normally ssh and scp authentic by asking for your password. You can set up ssh to use public/private key authentication instead. This creates a trust relationship between the machines, so please only use this between machines within the lab. Here are the steps needed to setup public/private key authentication so that remote machine trusts the local machine:

1. Open two xterms.
2. Connect to the remote machine using ssh in one window.
3. Run ssh-keygen -t dsa -b 2048 on the local machine (you only have to do this once per machine). Just hit enter when it asks for a pass phrase.
5. On the local machine run cat ~/.ssh/id_dsa.pub.
6. On the local machine select the print out with the mouse. The text should be completely highlighted starting with “ssh-dsa” through the end of the line containing your login name and the machine name.
8. With the remote window focussed, press the middle mouse button to paste the ssh key.
9. Type C-d, i.e. press “Ctrl”, press “d”, release “d”, release “Ctrl”.
10. You should now be able to ssh to the remote machine from the local machine without entering a password.