RL-Glue 3.0 Technical Manual

Brian Tanner

Contents

1 Introduction

This document describes how to use RL-Glue when each of the agent, environment, and experiment program is written in C/C++. This scenario is also known as the *direct-compile* scenario, because

all of the components can be compiled together into a single executable. This contrasts with the more flexible way to use RL-Glue, where the `rl_glue` executable server acts as a bridge for agents, environments, and experiment programs written in any of: Python, Lisp, Matlab, Java, or C/C++.

For general information and motivation about RL-Glue, please read the RL-Glue overview documentation. This technical manual is about explaining the finer details of installing RL-Glue and creating direct-compile projects, so we won't rehash all of the high level RL-Glue ideas.

This software project is licensed under the Apache-2.0[1] license. We're not lawyers, but our intention is that this code should be used however it is useful. We'd appreciate to hear what you're using it for, and to get credit if appropriate.

This project has a home here:
`http://rl-glue.googlecode.com`

1.1 Software Requirements

This project requires nothing more exotic than a C compiler, Make, etc. This project uses a configure script that was created by GNU Autotools[2], so it should compile and run without problems on most *nix platforms (Unix, Linux, Mac OS X, Windows using CYGWIN[3]).

1.2 Getting the Project

The RL-Glue project can be downloaded either as a tarball or can be checked out of the subversion repository where it is hosted.

The tarball distribution can be found here:
`http://code.google.com/p/rl-glue/downloads/list`

To check the code out of subversion:
`svn checkout http://rl-glue.googlecode.com/svn/trunk rl-glue`

1.3 Installing the Project

The package was made with autotools, which means that you shouldn't have to do much work to get it installed.

[1] `http://www.apache.org/licenses/LICENSE-2.0.html`
[2] `http://sources.redhat.com/autobook/`
[3] `http://www.cygwin.com/`

1.3.1 Simple Install

If you are working on your own machine, it is usually easiest to install the headers, libraries, and `rl_glue` binary into `/usr/local`, which is the default installation location but requires *sudo* or *root* access.

The steps are:

```
>$ ./configure
>$ make
>$ sudo make install
```

Provided everything goes well, the headers have now been installed to `/usr/local/include` the libs to `/usr/local/lib`, and `rl_glue` to `/usr/local/bin`.

1.3.2 Install To Custom Location (without *root* access)

If you don't have *sudo* or *root* access on the target machine, you can install RL-Glue in your home directory (or other directory you have access to). If you install to a custom location, you will need set your `CFLAGS` and `LDFLAGS` variables appropriately when compiling your projects. See Section 2.2 for more information.

For example, maybe we want to install RL-Glue to `/Users/btanner/tmp/rlglue`. The commands are:

```
>$ ./configure --prefix=/Users/btanner/tmp/rlglue
>$ make
>$ make install
```

Provided everything goes well, the headers, libraries, and binaries have been respectively installed to
`/Users/btanner/tmp/rlglue/include`
`/Users/btanner/tmp/rlglue/lib`
`/Users/btanner/tmp/rlglue/bin`

1.4 Uninstall

If you decide that you don't want RL-Glue on your machine anymore, you can easily uninstall it. The procedures varies a tiny bit depending on if you installed it to the default location, or somewhere custom.

1.4.1 RL-Glue Installed To Default Location

```
>$ ./configure
>$ sudo make uninstall
```

This will remove all of the headers, libraries, and binaries from `/usr/local`.

1.4.2 RL-Glue Installed To Custom Location

You'll need to make sure that either you haven't reconfigured the directory you downloaded from, or, if you removed/changed that already, you have to run configure again the exact same way as when you installed it. For example:

```
>$ ./configure --prefix=/Users/btanner/tmp/rlglue
>$ make uninstall
```

That's it! This will remove all of the headers, libraries, and binaries from `/Users/btanner/tmp/rlglue`.

2 Agent, Environments, and Experiments

We have provided a skeleton agent, environment, and experiment program that can be compiled together and run as an experiment. This is a good starting point for projects that you may write in the future. We'll start by explaining how to run compile and run the experiment, then we'll talk in more detail about each part.

2.1 Compiling and Running Skeleton

If RL-Glue has been installed in the default location, `/usr/local`, then you can compile and run the experiment like:

```
>$ cd examples/skeleton/
>$ make
>$ ./SkeletonExperiment
```

We will spend a little bit talking about how to compile the project, because not everyone is comfortable with using a `Makefile`. To compile the project from the command line, you could do:

```
>$ cc *.c -lrlglue -lrlutils -o SkeletonExperiment
```

It might be useful to break this down a little bit:

cc The C compiler. You could also use `gcc` or `g++`, etc.

***.c** Compile `SkeletonExperiment.c` `SkeletonAgent.c` `SkeletonEnvironment.c` sources files.

-lrlglue Link to the RLGlue library. This is where the *glue* that connects the three components is defined.

-lrlutils Link to the RLUtils library, which comes with RL-Glue. This library contains convenience functions for allocating and cleaning up the structure types (Section 4.2.4). If you don't use these convenience functions, you don't need this library.

At this point, we've compiled the project, now we just have to run the experiment:

```
>$ ./SkeletonExperiment
```

You should see output like the following if it worked:

```
>$ ./SkeletonExperiment
Experiment starting up!
RL_init called, the environment sent task spec: 2:e:1_[i]_[0,5]:1_[i]_[0,1]:[-1,1]


----------Sending some sample messages----------
Agent responded to "what is your name?" with: my name is skeleton_agent!
Agent responded to "who is your daddy and what does he do?" with: I don't know how to respond t

Environment responded to "what is your name?" with: my name is skeleton_environment!
Environment responded to "who is your daddy and what does he do?" with: I don't know how to res


----------Running a few episodes----------
Episode 0  100 steps  0.000000 total reward  0 natural end
Episode 1  90 steps  -1.000000 total reward  1 natural end
Episode 2  56 steps  1.000000 total reward  1 natural end
Episode 3  100 steps  0.000000 total reward  0 natural end
Episode 4  96 steps  -1.000000 total reward  1 natural end
Episode 5  1 steps  0.000000 total reward  0 natural end
Episode 6  106 steps  1.000000 total reward  1 natural end


----------Stepping through an episode----------
First observation and action were: 10 1
```

```
----------Summary----------
It ran for 204 steps, total reward was: -1.000000
```

That's all there is to it! You just ran a direct-compile RL-Glue experiment! Congratulations!

2.2 Custom Flags for Custom Installs

If RL-Glue has been installed in a custom location (for example: `/Users/joe/glue`), then you will need to set the header search path in `CFLAGS` and the library search path in `LDFLAGS`. You can either do this each time you call make, or you can export the values as environment variables.

To do it on the command line:

```
>$ CFLAGS=-I/Users/joe/glue/include LDFLAGS=-L/Users/joe/glue/lib make
```

That might turn out to be quite a hassle while you are developing. In that case, you can either update the `Makefile` to include these flags, or set an environment variable. If you are using the bash shell you can `export` the environment variables:

```
>$ export CFLAGS=-I/Users/joe/glue/include
>$ export LDFLAGS=-L/Users/joe/glue/lib
>$ make
```

In some cases, you may be able to compile and link your programs without incident, but you receive shared library loading errors when you try to execute them, as mentioned in Gotchas! (Section 2.6.2).

In these cases, you may also have to set `LD_LIBRARY_PATH` (Linux) or `DYLD_LIBRARY_PATH` (OS X) environment variables, like:

```
>$ export LD_LIBRARY_PATH=/Users/joe/glue/lib
```

In some cases (64-bit linux looks in `/usr/local/lib64`?) you may have to use this approach even when RL-Glue is installed in the default location:

```
>$ export LD_LIBRARY_PATH=/usr/local/lib
```

When you open a new terminal window, all of these environment variables will be lost unless you put the appropriate `export` lines in your shell startup script.

2.3 Agents

This agent implements all the required functions and provides a good example of how to create a simple agent.

The pertinent files are:

`examples/skeleton/SkeletonAgent.c`

This agent does not learn anything and randomly chooses integer action 0 or 1.

The Skeleton agent is very simple and well documented, so we won't spend any more time talking about it in these instructions. Please open it up and take a look.

POSSIBLE CONTRIBUTION: If you take a look at the agent and you think it's not easy to understand, think it could be better documented, or just that it should do some fancier things, let us know and we'll be happy to do it!

2.4 Environments

The Skeleton experiment implements all the required functions and provides a good example of how to create a simple environment.

The pertinent files are:

`examples/skeleton_environment/SkeletonEnvironment.c`

This environment is episodic, with 21 states, labeled $\{0, 1, \ldots, 19, 20\}$. States $\{0, 20\}$ are terminal and return rewards of $\{-1, +1\}$ respectively. The other states return reward of 0. There are two actions, $\{0, 1\}$. Action 0 decrements the state number, and action 1 increments it. The environment starts in state 10.

The Skeleton environment is very simple and well documented, so we won't spend any more time talking about it in these instructions. Please open it up and take a look.

POSSIBLE CONTRIBUTION: If you take a look at the environment and you think it's not easy to understand, think it could be better documented, or just that it should do some fancier things, let us know and we'll be happy to do it!

2.5 Experiments

The Skeleton experiment implements all the required functions and provides a good example of how to create a simple experiment. This section will follow the same pattern as the agent version (Section 2.3). This section will be less detailed because many ideas are similar or identical.

The pertinent files are:

`examples/skeleton_experiment/SkeletonExperiment.c`

This experiment runs `RL_Episode` a few times, sends some messages to the agent and environment, and then steps through one episode using `RL_step`.

The Skeleton experiment is very simple and well documented, so we won't spend any more time talking about it in these instructions. Please open it up and take a look.

POSSIBLE CONTRIBUTION: If you take a look at the experiment and you think it's not easy to understand, think it could be better documented, or just that it should do some fancier things, let us know and we'll be happy to do it!

2.6 Gotchas!

2.6.1 Crashes and Bus Errors in Experiment Program

If you are running an experiment using `RL_step`, beware that the last step (when `terminal==1`), the action will be empty. If you try to access the values of the actions in this case, you may crash your program.

2.6.2 Shared Library Loading Errors

On some machines we've used, RL-Glue installs without incident, but when the experiment is run, the system gives an error message similar to:

```
>$ ./SkeletonExperiment: error while loading shared libraries: librlglue-3:0:0.so.1:
cannot open shared object file: No such file or directory
```

If this happens, the operating system might have an alternate search path, and might not be looking in `/usr/local/lib` for libraries. You can troubleshoot this problem by doing:

```
>$ LD_DEBUG=libs ./SkeletonExperiment
```

If you see that `/usr/local/lib` is not in the search path, you may want to add it to your library search path using `LDFLAGS` or `LD_LIBRARY_PATH`. See Section 2.2 for more information.

3 Who creates and frees memory?

Memory management can be confusing in C/C++. It might seem especially mysterious when using RL-Glue, because sometimes the structures are passed directly from function to function (in

direct-call RL-Glue), but other times they are written and read through a network socket (with the C/C++ network codec).

3.1 Copy-On-Keep

The rule of thumb to follow in RL-Glue is what we call *copy-on-keep*. Copy-on-keep means that when you are passed a dynamically allocated structure, you should only consider it valid within the function that it was given to you. If you need a persistent copy of the data outside of that scope, you should make a copy: copy it if you need to keep it.

3.1.1 Task Spec Example

```
/********************   UNSAFE   ********************/
task_specification_t task_spec_copy=0;

void agent_init(const task_specification_t task_spec){
        /*
            Not making a copy, just keeping a pointer to the data
        */
    task_spec_copy=task_spec;
}

action_t agent_start(observation_t this_observation) {
        /*
            Behavior undefined.  Who knows if the string the task_spec
            was originally pointing to still exists?
        */
    printf("Task spec we saved is: %s\n",task_spec_copy);
    ...


/********************   SAFE   ********************/
task_specification_t task_spec_copy=0;

void agent_init(const task_specification_t task_spec){
        /*
            Allocating space (need length+1 for the terminator character)
        */
    task_spec_copy=(char *)calloc(strlen(task_spec)+1, sizeof(char));
    strcpy(task_spec_copy,task_spec);
}

action_t agent_start(observation_t this_observation) {
        /*
```

```
            This is fine, because even if
            the task_spec was deleted, we have a copy.
        */
    printf("Task spec we saved is: %s\n",task_spec_copy);
    ...
```

3.1.2 Observation Example (using helper library)

```
/*******************   UNSAFE   *******************/
observation_t last_observation;
action_t agent_start(observation_t this_observation) {
        /*
            Unsafe, points last_observation to
            this_observation's arrays!
        */
    last_observation=this_observation;
...

/*******************   SAFE   *******************/
observation_t last_observation;
action_t agent_start(observation_t this_observation) {
        /*
           This helper function actually frees the
           allocated memory inside last_observation, allocates
           new memory, and copies from this_observation!
        */
    replaceRLStruct(&this_observation, &last_observation);
...
```

These examples might make it clear why most of our samples use global variables for things that are changed often, like structures that we return, or structures keeping track of the previous action/observation etc. This allows us to only worry about re-allocating the arrays inside the struct, instead of also worrying about pointers to the structs themselves.

3.2 Free Your Mess

When using this RL-Glue, you are responsible for cleaning up any memory that you allocate. The good news is that that you can trust that between function calls, any memory you've returned to a caller has either been copied or is not necessary (it is safe to free it). Remember that in C/C++ it's not safe to return pointers to stack-based memory.

The Skeleton examples do the appropriate thing in this respect: the `intArrays` that need to be dynamically allocated are allocated in the `_init` methods, and then the memory is released in the

_cleanup methods.

3.2.1 Messaging Examples

Strings seem to be more complicated for some people, so here are a couple of examples:

```
/********************   UNSAFE   ********************/
message_t agent_message(const message_t inMessage) {
    char theBuffer[1024];
    sprintf(theBuffer,"this is an example response message\n");
        /*
            This returns the address of a local variable
            bad idea and compiler will complain
        */
    return theBuffer;
}


/****************   UNSAFE (MEMORY LEAK)   ***************/
message_t agent_message(const message_t inMessage) {
    char theBuffer[1024];
    message_t returnString=0;
    sprintf(theBuffer,"this is an example response message\n");
    returnString=(char *)calloc(strlen(theBuffer+1),sizeof(char));
    strcpy(returnString,theBuffer);
        /*
            Memory leak... every time this function is called
            a new returnString is allocated, but nobody will
            ever clean them up!
        */
    return returnString;
}


/********************   SAFE   ********************/
message_t agentReturnString=0; /*Global Variable */
message_t agent_message(const message_t inMessage) {
    char theBuffer[1024];
    sprintf(theBuffer,"this is an example response message\n");

        /*
            This code will free the memory on subsequent calls
        */
    if(agentReturnString!=0){
        free(agentReturnString);
```

```
        agentReturnString=0;
    }
    agentReturnString=(char *)calloc(strlen(theBuffer+1),sizeof(char));
    strcpy(agentReturnString,theBuffer);
    return agentReturnString;
}
```

4 RL-Glue C/C++ Specification Reference

This section will explain how the RL-Glue types and functions are defined for C/C++. This is important both for direct-compile experiments, and for components that use the C/C++ network codec.

4.1 Types

The types used here will be the same for the C/C++ network codec.

4.1.1 Simple Types

The simple types are:

```
    typedef double reward_t;
    typedef int terminal_t;
    typedef char* message_t;
    typedef char* task_specification_t;
```

4.1.2 Structure Types

All of the major structure types (observations, actions, random seed keys, and state keys) are typedef'd to `rl_abstract_type_t`.

```
typedef struct
{
    unsigned int numInts;
    unsigned int numDoubles;
    unsigned int numChars;
    int* intArray;
    double* doubleArray;
    char* charArray;
} rl_abstract_type_t;
```

The specific names and definitions of the structure types are:

```
    typedef rl_abstract_type_t observation_t;
    typedef rl_abstract_type_t action_t;
    typedef rl_abstract_type_t random_seed_key_t;
    typedef rl_abstract_type_t state_key_t;
```

The composite structure types returned by `env_step` are:

```
typedef struct{
    observation_t o;
    action_t a;
} observation_action_t;

typedef struct{
    reward_t r;
    observation_t o;
    terminal_t terminal;
} reward_observation_t;

typedef struct {
    reward_t r;
    observation_t o;
    action_t a;
    terminal_t terminal;
} reward_observation_action_terminal_t;
```

4.1.3 Summary

The type names are:

```
    reward_t
    terminal_t
    message_t
    task_specification_t
    observation_t
    action_t
    observation_action_t
    reward_observation_t
    reward_observation_action_t
```

4.2 Functions

4.2.1 Agent Functions

All agents **should implement** these functions, located in `rlglue/Agent_common.h`

```
void agent_init(const task_specification_t task_spec);
action_t agent_start(observation_t o);
action_t agent_step(reward_t r, observation_t o);
void agent_end(reward_t r);
void agent_cleanup();
message_t agent_message(const message_t message);
```

4.2.2 Environment Functions

All environments **should implement** these functions, located in `rlglue/Environment_common.h`

```
task_specification_t env_init();
observation_t env_start();
reward_observation_t env_step(action_t a);
void env_cleanup();
message_t env_message(const message_t message);
void env_set_state(state_key_t sk);
void env_set_random_seed(random_seed_key_t rsk);
state_key_t env_get_state();
random_seed_key_t env_get_random_seed();
```

4.2.3 Experiments Functions

All experiments **can call** these functions, located in `rlglue/RL_glue.h`

```
task_specification_t RL_init();
observation_action_t RL_start();
reward_observation_action_terminal_t RL_step();
void RL_cleanup();

terminal_t RL_episode(unsigned int num_steps);

message_t RL_agent_message(message_t message);
message_t RL_env_message(message_t message);

reward_t RL_return();
```

```
    int RL_num_steps();
    int RL_num_episodes();

    void RL_set_state(state_key_t sk);
    void RL_set_random_seed(random_seed_key_t rsk);
    state_key_t RL_get_state();
    random_seed_key_t RL_get_random_seed();
```

4.2.4 RLUtils Library Functions

You can get access to these functions by linking to libRLUtils (`-lrlutils`) and by including the appropriate header:

```
#include <rlglue/utils/C/RLStruct_util.h>


/* Copies all of the data from src to dst, freeing and allocating only if necessary*/
void replaceRLStruct(const rl_abstract_type_t *src, rl_abstract_type_t *dst);

/*
    Frees the 3 data arrays if they are not NULL, sets them to NULL,
    and sets numInts, numDoubles, numChars to 0
*/
void clearRLStruct(rl_abstract_type_t *dst);

/*  calls clearRLStruct on dst, and then frees the dst pointer */
void freeRLStructPointer(rl_abstract_type_t *dst);

/*
    Given a pointer to a rl_abstract_type_t, allocate arrays of the requested sizes,
    set the contents of the arrays to 0, and set numInts, numDoubles,
    numChars in the struct appropriately.
*/
void allocateRLStruct(rl_abstract_type_t *dst,
                      const unsigned int numInts,
                      const unsigned int numDoubles,
                      const unsigned int numChars);

/*
    Create a new rl_abstract_type_t, allocate its arrays
    and its numInts/Doubles/Chars using allocateRLStruct,
    return the pointer
*/
rl_abstract_type_t *allocateRLStructPointer(const unsigned int numInts,
```

```
                                        const unsigned int numDoubles,
                                        const unsigned int numChars);
```

5 Changes and 2.x Backward Compatibility

There were many changes from RL-Glue 2.x to RL-Glue 3.x. Most of them are at the level of the API and project organization, and are addressed in the RL-Glue project documentation, not this technical manual.

5.1 Types

All of the types that existed in the RL-Glue 2.x (ex: `Observation` instead of `observation_t`) are still supported through a definition file called `legacy_types.h`. If you don't want to update your old agents to the new types, you can use the old names by doing the following in your source files:

```
#include<rlglue/legacy_types.h>
```

6 Frequently Asked Questions

We're waiting to hear your questions!

7 Credits and Acknowledgements

Andrew Butcher originally wrote the RL-Glue library and network library . Thanks Andrew.

Brian Tanner has since grabbed the torch and has continued to develop RL-Glue and the codecs.

7.1 Contributing

If you would like to become a member of this project and contribute updates/changes to the code, please send a message to rl-glue@googlegroups.com.

Document Information

```
Revision Number: $Rev: 873 $
Last Updated By: $Author: brian@tannerpages.com $
```

Last Updated : $Date: 2008-10-01 12:58:14 -0600 (Wed, 01 Oct 2008) $
$URL: https://rl-glue.googlecode.com/svn/trunk/docs/TechnicalManual.tex $