RL-Glue C/C++ Codec 3.0 Manual

Brian Tanner

Contents

1 Introduction

This document describes how to use the C/C++ RL-Glue Codec, a software library that provides socket-compatibility with the RL-Glue Reinforcement Learning software library.

For general information and motivation about the RL-Glue[1] project, please refer to the documentation provided with that project.

This codec will allow you to create agents, environments, and experiment programs in C and/or C++.

1.1 Software Requirements

This project requires that RL-Glue has been installed on your computer. It has no additional requirements beyond RL-Glue: nothing more exotic than a C compiler, Make, etc. This codec 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 Codec

The codec 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-ext/downloads/list`

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

[1] `http://glue.rl-community.org/`
[2] `http://sources.redhat.com/autobook/`
[3] `http://www.cygwin.com/`

1.3 Installing the Codec

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

1.3.1 Simple Codec Install

If you are working on your own machine, it is usually easiest to install the headers and libraries 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` and the libs to `/usr/local/lib`.

1.3.2 Install Codec when RL-Glue is in a custom location

If `configure` can't find RL-Glue installed on your machine, it will give you an error like the following:

```
checking for rlConnect in -lrlgluenetdev... no
configure: error: RL-Glue library not found.
You must have RL-Glue installed to use this codec.

If you have not downloaded it please see http://glue.rl-community.org/
If you do have it installed in a non-standard location you may need to use the
--with-rl-glue=/path/to/rlglue
command line switch to specify where the rl-glue root is located.
```

If you installed RL-Glue to some place other than `/usr/local`, say `/Users/btanner/tmp/rlglue`, you could do:

```
>$ ./configure --with-rl-glue=/Users/btanner/tmp/rlglue/lib
>$ make
>$ sudo make install
```

1.3.3 Install Codec To Custom Location (without *root* access)

If you don't have *sudo* or *root* access on the target machine, you can install the codec 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 agents, environments, and experiments. See Section 2.1 for more information.

For example, maybe we want to install the codec to `/Users/btanner/tmp/rlglue`. This will **not** clobber RL-Glue if it is already installed to this location, it will install beside it. The commands are:

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

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

2 Agents

We have provided a skeleton agent with the codec that is a good starting point for agents that you may write in the future. It implements all the required functions and provides a good example of how to compile a simple agent.

The pertinent files are:

```
examples/skeleton_agent/SkeletonAgent.c
examples/skeleton_agent/Makefile
```

This agent is not particularly interesting, it does not learn anything and randomly chooses integer action 0 or 1.

If RL-Glue and this codec have been installed in the default location, `/usr/local`, then you can compile and run the agent like:

```
>$ cd examples/skeleton_agent
>$ make
>$ ./SkeletonAgent
```

You will see something like:

```
RL-Glue C Agent Codec Version 1.0-alpha-3, Build 192:208M
Connecting to host=127.0.0.1 on port=4096...
```

This means that the SkeletonAgent is running, and trying to connect to the `rl_glue` executable server on the local machine through port 4096!

You can kill the process by pressing `CTRL-C` on your keyboard.

See Section 2.1 if RL-Glue or this Codec are not installed in default locations.

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!

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

```
>$ cc SkeletonAgent.c -lrlutils -lrlagent -o SkeletonAgent
```

On some platforms, you may need to add `-lrlgluenetdev`

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

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

SkeletonAgent.c Compile the SkeletonAgent.c source file.

-lrlutils Link to the RLUtils library, which comes with this codec. This library contains convenience functions for allocating and cleaning up the structure types (Section 7.1.2). If you don't use these convenience functions, you don't need this library.

-lrlagent Link to the RLAgent library of the codec. This is where the main agent loop is defined. The main agent loop connects to the `rl_glue` executable server and dispatches commands sent by the glue.

-lrlgluenetdev Link to the RLGlueNetDev library from the RL-Glue project. This library is automatically linked through `rlagent` on most platform (except notably Cygwin). RLGlueNetDev provides implementations of the low level network code that is used by all three parts of the codec, as well as the `rl_glue` executable server.

2.1 Custom Flags for Custom Installs

If RL-Glue **or** this codec have 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. These instructions apply to agents, environments, and experiment programs.

To do it on the command line:

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

When you open a new terminal window, these values will be lost unless you put the appropriate `export` lines in your shell startup script. But that's enough about that, because we're getting well off topic.

3 Environments

We have provided a skeleton environment with the codec that is a good starting point for environments that you may write in the future. It implements all the required functions and provides a good example of how to compile a simple environment. This section will follow the same pattern as the agent version (Section 2). This section will be less detailed because many ideas are similar or identical.

The pertinent files are:

```
examples/skeleton_environment/SkeletonEnvironment.c
examples/skeleton_environment/Makefile
```

This environment is not particularly interesting. It 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.

If RL-Glue and this codec have been installed in the default location, `/usr/local`, then you can compile and run the environment like:

```
>$ cd examples/skeleton_environment
>$ make
>$ ./SkeletonEnvironment
```

You will see something like:

```
RL-Glue C Environment Codec Version 1.0-alpha-3, Build 192:208M
Connecting to host=127.0.0.1 on port=4096...
```

This means that the SkeletonEnvironment is running, and trying to connect to the `rl_glue` executable server on the local machine through port 4096!

You can kill the process by pressing `CTRL-C` on your keyboard.

See Section 2.1 if RL-Glue or this Codec are not installed in default locations.

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!

Compiling the environment is almost identical to compiling the skeleton agent, except you need to link to the `RLEnvironment` library instead of `RLAgent`.

```
>$ cc SkeletonEnvironment.c -lrlutils -lrlenvironment -o SkeletonEnvironment
```

On some platforms, you may need to add `-lrlgluenetdev`

4 Experiments

We have provided a skeleton experiment with the codec that is a good starting point for experiment that you may write in the future. It implements all the required functions and provides a good example of how to compile a simple experiment. This section will follow the same pattern as the agent version (Section 2). This section will be less detailed because many ideas are similar or identical.

The pertinent files are:

```
examples/skeleton_experiment/SkeletonExperiment.c
examples/skeleton_experiment/Makefile
```

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`.

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

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

You will see something like:

```
RL-Glue C Experiment Codec Version 1.0-alpha-3, Build 192:208M
Connecting to host=127.0.0.1 on port=4096...
```

This means that the SkeletonExperiment is running, and trying to connect to the `rl_glue` executable server on the local machine through port 4096!

You can kill the process by pressing `CTRL-C` on your keyboard.

See Section 2.1 if RL-Glue or this Codec are not installed in default locations.

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!

Compiling the experiment is almost identical to compiling the skeleton agent, except you need to link to the `RLExperiment` library instead of `RLAgent`.

```
>$ cc SkeletonExperiment.c -lrlutils -lrlexperiment -o SkeletonExperiment
```

On some platforms, you may need to add `-lrlgluenetdev`

4.1 Gotchas!

- If you are calling `RL_step`, beware that the last step (when `terminal==1`), the action will be empty.

5 Putting it all together

At this point, we've compiled and run each of the three components, now it's time to run them with the `rl_glue` executable server. The following will work from the examples directory if you have them all built, and RL-Glue installed in the default location:

```
>$ rl_glue &
>$ skeleton_agent/SkeletonAgent &
>$ skeleton_environment/SkeletonEnvironment &
>$ skeleton_experiment/SkeletonExperiment
```

You should see output like the following if it worked:

```
>$ rl_glue &
RL-Glue Version 3.0-alpha-3, Build 848:852M
RL-Glue is listening for connections on port=4096

>$ skeleton_agent/SkeletonAgent &
RL-Glue C Agent Codec Version 1.0-alpha-3, Build 192:208M
Connecting to host=127.0.0.1 on port=4096...
RL-Glue C Agent Codec :: Connected
RL-Glue :: Agent connected.

>$ skeleton_environment/SkeletonEnvironment &
RL-Glue C Environment Codec Version 1.0-alpha-3, Build 192:208M
Connecting to host=127.0.0.1 on port=4096...
RL-Glue C Environment Codec :: Connected
RL-Glue :: Environment connected.

$> skeleton_experiment/SkeletonExperiment


Experiment starting up!
RL-Glue C Experiment Codec Version 1.0-alpha-3, Build 192:208M
Connecting to host=127.0.0.1 on port=4096...
RL-Glue C Experiment Codec :: Connected
RL-Glue :: Experiment connected.
```

```
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 to your message

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 respond to your message


----------Running a few episodes----------
Episode 0  100 steps  0.000000 total reward  0 natural end
Episode 1  44 steps  -1.000000 total reward  1 natural end
Episode 2  18 steps  -1.000000 total reward  1 natural end
Episode 3  100 steps  0.000000 total reward  0 natural end
Episode 4  50 steps  1.000000 total reward  1 natural end
Episode 5  1 steps  0.000000 total reward  0 natural end
Episode 6  28 steps  1.000000 total reward  1 natural end


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

----------Summary----------
It ran for 144 steps, total reward was: -1.000000
```

6 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 (as with

this codec).

6.1 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.

6.1.1 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);
    ...
```

6.1.2 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.

6.2 Free Your Mess

When using this codec, 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.

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;
}
```

7 Codec Specification Reference

This section will explain how the RL-Glue types and functions are defined for this codec. This isn't meant to be an interesting section of this document, but it will be handy.

7.1 Types

The types used by this codec are the same as the direct-compile RL-Glue library.

7.1.1 Simple Types

The simple types are:

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

7.1.2 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;
```

7.1.3 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
```

7.2 Functions

7.2.1 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);
```

7.2.2 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();
```

7.2.3 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();
```

8 Changes and 2.x Backward Compatibility

8.1 Types

All of the types that existed in the 2.x codec (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>
```

9 Frequently Asked Questions

We're waiting to hear your questions!

10 Credits and Acknowledgements

Andrew Butcher originally wrote the RL-Glue network library and first version of this codec. Thanks Andrew.

Brian Tanner has since grabbed the torch and has continued to develop the codec.

10.1 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: 224 $
Last Updated By: $Author: brian@tannerpages.com $
Last Updated : $Date: 2008-09-28 16:08:13 -0600 (Sun, 28 Sep 2008) $
$URL: https://rl-glue-ext.googlecode.com/svn/trunk/projects/codecs/C/docs/doc.tex $