RETSINA MicroAFC for Linux
version: 1.01
Author: Martin van Velsen vvelsen@cs.cmu.edu
Editor: Michael Rectenwald mdr2@cs.cmu.edu

Table of Contents

1. Introduction

For several years, the Intelligent Software Agents Lab at Carnegie Mellon University's Robotics Institute has researched, developed, demonstrated and tested Multi-Agent Systems (MASs). To extend the testing and demonstration of the technologies and principles of MASs, an implementation was written in C++ called the Agent Foundation Classes (AFC). This software development kit allows developers to rapidly create software agents that can participate in robust, reliable and scalable agent systems. The full version of the AFC encompasses numerous infrastructure components and provides a great many tools and utilities for agent developers.

When working with the AFC, one is essentially developing from within an agent's perspective. Although this approach is suitable for most agent applications, there are situations in which a more lightweight solution is required. For example, one of the motivations for a smaller, stripped down version of the AFC was the requirement of adding agent capabilities to existing monolithic software applications where the luxury of wrapping does not obtain. Instead, an additive process takes place, whereby agent capabilities are added to the application, instead of the application becoming an agent. The libraries and tools resulting from the adaptation of the AFC to a small micro version has resulted in MicroAFC. MicroAFC uses principles and mechanisms similar to those of the full AFC. The main difference is that only a small subset of AFC functionality is provided in MicroAFC. This functionality is enough, however, to allow an application become part of an MAS.

In this manual we will explain and demonstrate how to add the MicroAFC functionality to your Linux applications. The documentation assumes a working knowledge of the C programming language and a familiarity with the gcc compiler environment. This is a work in progress; we do not provide full install scripts and packages at this time. You will manually have to install and configure your MicroAFC system.

2. Installation

1. Download a copy of the MicroAFC distribution. For this example we will be using: microafc-v1.01.tgz

2. Unpack the archive by using the command:

```
tar -xvf microafc-v1.01.tgz
```

This command should create a subdirectory that holds the distribution.

3. Copy the library file *.lib to your local library directory:

```
cd microafc
cp *.lib /usr/local/lib
```

4. Copy the header files to your local include directory:

```
cp *.h /usr/local/include
```

That should be the extent of the installation. In case you do not have root access, you will have to configure your system to use local directories in your personal space. We advise that you keep the libraries and include the files in the distribution directory. Also, we suggest that you set your library path, including a path to the distribution directory.

3. Testing the Installation

To make sure the package was properly installed and configured, we will now build the examples and test them. From within the distribution directory type:

```
make examples
```

This should compile three basic examples used for demonstration purposes later in this manual. To test them, navigate to the examples directory. You will find a number of object (*.o) files there, three executables and three script files (*.sh).

4. Agent API

Before we describe any detailed API for this release of the MicroAFC, we will need to explain some of the differences between the MicroAFC and the full AFC. From a functional perspective, each version works in a slightly different way. The AFC is event-driven and uses OOP mechanisms to abstract agent behaviors away from low-level network interaction. In the MicroAFC we use a polling system, since we assume that you will be adding the code to existing applications. This means that developers using the MicroAFC will be working at a much lower more direct level than AFC developers. We will explain more about this difference in the next chapter.

4.1 Basics

Since the MicroAFC is a stripped-down version of the AFC, it has only a small fraction of the AFC's capabilities. However, there is enough functionality to create what, from the MAS community standpoint, amounts to a fully functional agent. The following core capabilities are present in this distribution:

- KQML parser
- ANS client
- Communicator
- Agent management API
- Debugging tools

We will elaborate and give a full description of each of the modules and their functionality. Even though the AFC is written in C++ and uses OOP as its core principle, we tried to maintain API consistency for the MicroAFC. In cases where we need persistence between functions, we use a struct to encapsulate information. For example in the AFC you would write:

```
c_parser->parse_message ("(Hello World)");
```

In the MicroAFC, there is no notion of an object, and therefore we cannot store local state information about a parser. Instead, to breach this gap, we have devised the API to use pointers to *structs*. The method above was translated to look like:

```
struct kqml_message *a_message=parse_kqml_message ("(Hello World)");
```

The decomposed message is stored in a kqml_message struct and should be used afterwards as a reference to this message. For example:

```
char *receiver=find_receiver (a_message);
```

Here the parser uses the kqml_message struct to reference a previously parsed message. You can think of these structs as classes without methods. The methods have been replaced by functions that take references to objects (structs).

There are a number of standards that govern the API described in this manual. First of all we use the same Boolean variables in the AFC as we do in the MicroAFC. A Boolean variable is defined using the 'BOOL' statement and the contents of one such instance can either be 'TRUE' or 'FALSE'. Most functions will return a Boolean indicating that the operation they represent has succeeded or failed. Here is an example of a Boolean variable and it's instance:

```
BOOL truth=FALSE;
```

4.2 Execution Model

Each agent will go through an execution lifecycle. This lifecycle is state-driven and results in the agent going through very specific states during its operation. The MicroAFC is polling-, not event-based. This means that you will have to keep track of what the agent is doing in each state. You cannot directly set the state of an agent; instead, you call different functions to activate parts of your agent that will result in state changes. Be aware, however, that the agent may refuse to enter a state if a previous state has not been reached. For example you can enter the 'shutdown' stage directly after the 'init' stage, but you can't go directly from the

’create’ stage to the ’running’ stage. In order to further examine the agent states we will use the actual definitions in the code, the are defined as:

```
#define __AGENT_STATE_CONSTRUCTOR__   0
#define __AGENT_STATE_CREATE__        1
#define __AGENT_STATE_INIT__          2
#define __AGENT_STATE_RUNNING__       3
#define __AGENT_STATE_SHUTDOWN__      4
#define __AGENT_STATE_DESTRUCTION__   5
#define __AGENT_STATE_TOP_LEVEL_END__ 6
```

At any point during the execution of your agent, you can call the function ’get_state’ to find out where your agent is in the cycle. The full definition is:

```
int   get_state (void);
```

Even though you do not have direct control over the states, you do change them by moving from one primary function to the next. Primary functions are those that directly influence the control flow of the agent. The MicroAFC follows more or less the same control flow as the full AFC, but leaves out one startup function. Below is a listing of the declarations of the main control functions:

```
BOOL  agent_create       (int argc,char **argv);
BOOL  agent_init         (void);
char *agent_message_loop (void);
BOOL  agent_shutdown     (void);
BOOL  agent_delete       (void);
```

The functions are listed in order of appearance. This means that you will use them in the same order in your agent as they appear above. Keep in mind, however, that if you use them in a large pre-existing application, the transitioning between function calls may take quite some time. It is important to know this fact, because certain parts of the ANS code need to be called every 10 minutes by the communicator. Normally this happens automatically when you call the function in the main message loop. Although under most circumstances you will not be confronted with any timing issues, it is important to be aware of them. For example, the ’agent_message_loop’ should be called at fixed intervals to ensure that messages are properly extracted from the lower TCP/IP layers. If the gap between calls is too great, you may end up with a congested agent. Let us now examine the main functions in more detail.

4.2.1 Agent Creation

Since the MicroAFC was derived from the full AFC, you will see aspects of C++ programming cast into C formats. The creation function is a good example of this casting. Normally, an agent would be created by calling the constructor. In C, of course, we cannot do that. Although the notion of a constructor does not exist in C, we still need to have the agent setup a number of important variables. When you call this function in your code, the agent will do the following:

- Obtain a number of parameters from the operating system regarding the network. At this point it will determine its hostname and IP address.
- Investigate how much of a RETSINA infrastructure is available on disk. For example, the agent will see if there is a RETSINA environment variable and validate the directory it points to.
- Process any arguments that are given on the command line that configure important variables within the agent. See Appendix B for a list of command line arguments that the MicroAFC can take.

You are encouraged to examine the return value of this function since your agent will not proceed with the next steps if the create function returned a ’FALSE’ value.

4.2.2 Agent Initialization

In the initialization phase, your agent will become active on the network. Before this stage, all that happened was a setup and enquiry set of actions to prepare your agent for operation. After the initialization phase, your agent will be able to receive messages and will be a part of a running RETSINA agent system. This function does not take any arguments and all activity is automated. You will be able to obtain the result of the init function by examining the return value. If the return value was FALSE, then subsequent calls to any primary agent functions will fail. If you find yourself in this situation, it is best to quit your application. You will not need to call any shutdown or delete methods. In short, the following steps are taken with the initialization stage:

- Start a server on the listening port specified by the command line parameters or suggested by the operating system.
- Register your agent with an ANS server
- Prepare the message queues for incoming messages.

4.2.3 Agent Message Loop

If your code has arrived at this point, you are up and running and ready to process events and messages from outside the agent. It is very important to understand the mechanics that this method uses to make the agent run. We assume here that your main code is something like a “while” loop within your main function. Remember that the agent code can be part of a large application where you may not directly see the main loop. No matter how the actual code is designed, you will need an entry point that will be called at regular intervals. Below is the simplest version of such a loop:

```
char *result=NULL;

 while (0)
 {
   result=agent_message_loop ();

   if (result!=NULL)
   {
    if ((strcmp (result,"panic")==0) || (strcmp (result,"exit")==0))
     exit (1);
    else
     printf ("Incoming data: %s",result);
   }
 }
```

Call the 'agent_message_loop' from an endless loop and examine the result to see what action to take. At the writing of this document there are two pre-defined results that must be examined to ensure proper functioning. If the resulting string equals either 'panic' or 'exit' the agent code will no longer continue to function. The 'panic' result occurs if the agent detects any problems with the network layer or internal data consistency. The 'exit' value can result from an authorized external agent telling your agent to shut down. This is part of the FIPA agent management standards.

4.2.4 Agent Shutdown

When your application or agent is ready to leave the system you should call: 'agent_shutdown'. This will take the appropriate steps to remove your agent from a running agent system. The agent will take the following steps when this function is called:

- Unregister the agent from the ANS.
- Stop the server and close all open sockets.

4.2.5 Agent Delete

At this point your agent is no longer part of the agent network. Other agents will not be able to reach you. During the course of execution a large amount of variables were assigned and filled. This includes message buffers and incoming queues. Agent deletion will free all memory used by your agent. The core variables will still be intact in case you want to restart your agent without leaving the application. In other words, you could call the 'create', 'init' and 'message_loop' functions again.

4.3 Agent Management

In this section, we will discuss a number of functions that will let you set parameters within the agent. These parameters can only be set during the creation and initialization phase. All of the functions listed below will return a 'FALSE' value if called from any other agent stage. Here are the functions as defined so far:

```
BOOL  set_agent_name     (char *);
BOOL  set_agent_port     (unsigned int);
BOOL  set_ans_host       (char *);
BOOL  set_ans_port       (unsigned int);
```

If you want to force the agent's name to a specific string and not use command line arguments, then use the first function above to provide an alternative name. Call this function after the 'creation' function, since the agent will pre-set the name to an internal string if no other values are specified.

The second function is used to specify a listening port for your agent. Again, this value is normally obtained through command line arguments, but can be fixed after creation and before initialization. If this value is set to 0 either using the function listed here or through command line parameters, the communicator will ask the operating system to assign the first available listening port.

The next two functions will determine which ANS to use. You can specify a hostname and port number to use. Instead of a hostname you can also give an IP address. If no values are specified through either command line arguments or directly through the functions listed here, the agent will default these values to:

kriton.cimds.ri.cmu.edu 6677

See http://www.softagents.ri.cmu.edu/ans/javaANS.PDF for documentation on the RETSINA ANS.

5. Communicator Usage

Traditionally, the communicator was the part of the agent whose task was the management of agent-to-agent dialog. In the MicroAFC, the set of tasks are reduced in such way that the code is responsible for maintaining incoming queues of messages and providing functions to talk to remote agents. At this point in time, in MicroAFC there is one function to send messages to other agents:

```
BOOL comm_sendmessage (char *, // performative
                       char *, // receiver
                       char *, // ontology
                       char *, // language
                       char *, // reply-with
                       char *, // forward-to
                       char *, // content);
```

As you can see, the send_message takes a fair number of parameters. Not all of them are needed and many can be set to NULL. The parameters that are needed for this function to actually send a message are performative,receiver and content. All other strings can be NULL and will be filled in appropriately by the system. If no ontology is provided, then the string sent to a remote agent defaults to: "default-ontology." Consequently, if the language string is not provided, a "default-language" will be sent. In the MicroAFC version of the AFC there is no automatic generation of reply-with fields. This means that if this parameter is not filled in, no string will be sent. If the message was successfully sent as indicated by the TCP/IP layer, the function will return TRUE.

6. KQML Parser

The functions you will be working with most are the ones related to message parsing. In the MicroAFC we hard-wired the parser; the KQML parser is set as the parser. This setting represents another difference between the MicroAFC and the full AFC, where you are not aware of the ACL parser used. (Note the difference in function calls between 'parse_kqml_message' in the MicroAFC and 'parse_message' in the full AFC). When working with the KQML parser you will have to keep pointers to structs of type:

```
struct kqml_message
```

Every time you make a call to one of the parser functions you will need to supply a pointer to an instance of this struct. You will obtain this pointer when you first call the parser function:

```
struct kqml_message *parse_kqml_message (char *);
```

Here is an example of a call:

```
struct kqml_message *message;

message=parse_kqml_message ("(Hello World)");
if (message==NULL)
{
 debug ("Error parsing message");
}
```

Now that you can parse messages, you will need to extract information from them. The parser code provides a number of specific functions to retrieve the message envelope fields such as sender and receiver. The MicroAFC has the following access functions for envelope fields:

```
char *find_performative (struct kqml_message *);
 char *find_sender       (struct kqml_message *);
 char *find_ontology     (struct kqml_message *);
 char *find_language     (struct kqml_message *);
 char *find_receiver     (struct kqml_message *);
 char *find_reply_with   (struct kqml_message *);
 char *find_in_reply_to  (struct kqml_message *);
 char *find_contents     (struct kqml_message *);
 char *find_forward      (struct kqml_message *);
```

Each of these functions takes a pointer to a message struct and will return a valid string that contains the field requested. Keep in mind that you should not try to free any of the memory associated with these fields. Allocated messages can be freed by calling:

```
 void delete_message (struct kqml_message *);
```

When you start to parse content fields and other KQML structures other than the content field you will need a function to find specific fields within a message.You can use the following function to find a field within a message structure:

```
 char *find_token (struct kqml_message *,char *);
```

Here is a small snippet to illustrate the function above:

```
 struct kqml_message *envelope;
 struct kqml_message *string;

 envelope=parse_kqml_message ("(tell :string (Hello World))");
 if (envelope!=NULL)
 {
  string=find_token (envelope,"string");
  if (string!=NULL)
   printf ("String: %s",string);
 }

 delete_message (envelope);
```

This is all there is too it. Keep in mind that the result of the operation listed above can be used by the kqml_parse_message function again. Any number of nestings can exist in a message.

7. ANS

Although in most cases the Communicator and core agent code will use the ANS functions, there might be times where you will want to have direct control over the control flow of the interaction with an Agent Name Server. The MicroAFC provides a number of functions to interact with an ANS and to obtain information from the ANS your agent is currently registered with. The following functions are defined for the MicroAFC:

```
 BOOL ans_lookup           (char *,char *,unsigned int *);
 BOOL ans_register         (void);
 BOOL ans_unregister       (void);
 BOOL ans_unregister_agent (char *);
```

7.1 Lookup

When a message is sent to another agent, the communicator code will use the 'ans_lookup' method to obtain the hostname and port number for the recipient agent. This function takes three parameters. The first parameter is the name of the agent you want to lookup. The second parameter is a pointer to a string where the hostname will be stored in. You will have to make sure there is enough space to hold the hostname/IP address. The last parameter you will have to supply is a pointer to an unsigned integer. This parameter will hold the port number the target agent is listening on. For example:

```
char         hostname [128];
unsigned int portnumber;

if (ans_lookup ("InformationAgent",hostname,&portnumber)==FALSE)
{
 debug ("Unable to lookup: InformationAgent");
}
else
{
 debug ("Location for InformationAgent is: [%s][%d]",hostname,portnumber);
}
```

7.2 Registration

Under normal circumstances, your agent will automatically register with an ANS upon startup. This will happen when the agent enters the __AGENT_STATE_INIT__ state, or when you call the 'agent_init' function. After the proper variables have been set the agent calls 'ans_register'. See the section on 'Agent API' for more information on important variables used by your agent. When the agent starts shutting down, it will automatically call 'ans_unregister'. This only happens if the agent is in the __AGENT_STATE_SHUTDOWN__ state, or when the function 'agent_shutdown' is called. Even though there are some restrictions as to when ANS registrations and unregistrations can take place, you the developer can freely call these methods at any time when the agent is in the __AGENT_STATE_RUNNING__ state. This version of the AFC does not keep track of a registration state. It is therefore possible to call the two methods out of order. It is up to the developer to maintain a proper registration state.

7.3 Unregistering other agents

There might be a situation where for development and debug purposes you may want to unregister an agent from the ANS other then your own running agent. For example, you could use the MicroAFC API to create a test agent that monitors and maintains other agents. We've provided the 'ans_unregister_agent' method for testing purposes. This does not mean that the ANS will actually honor the request. If you are running an ANS that uses authentication for registrations, you might not be able to use this function. The function takes only one parameter, which is the name of the agent to be unregistered. If the action succeeds, a TRUE value will be returned; otherwise a FALSE value will be returned.

8. Debugging Tools

The same debugging methods that are present in the AFC are also available in MicroAFC. You should be able to use the exact same methods to add debugging text to the log file. A number of advanced features from the AFC were left out and you will have to call two methods by hand to configure logging. The three methods related to debugging are defined as:

```
void debug_init (char *);
void debug      (char *, ...);
void debug_exit (void);
```

The first method is used before the agent starts and will setup file logging. Make a call to this method as the very first function call. Supply the name of a file to log to. If this parameter is NULL, the logfile will be called: "Logfile.txt" and can be found in the same directory from which you started the agent/application. If the ’debug_init ()’ function is not called, then subsequent calls to ’debug’ will result in debugging information on the console. After the agent is shutdown, call: ’debug_exit ()’ to flush and close the logfile. During any part of the agent’s usage you can call the ’debug’ function to log text-strings to a logfile. The code will add a: [RETSINA] tag and a line number field. A regular debugging call could look like:

```
 debug ("Hello World");
```

```
This would appear in the logfile as:
```

```
 [RETSINA][00001] Hello World
```

Notice that we do not add new lines to the debugging string. The debugging code will automatically do that for you. You can also add additional parameters to the debugging string, which makes this function call virtually identical to ’printf’. The following is an example of this usage:

```
 debug ("Agent uses socket: %d",0);
```

You would see the following in the logfile:

```
 [RETSINA][00002] Agent uses socket: 0
```

Appendix A. License Agreement

CARNEGIE MELLON UNIVERSITY
NON-EXCLUSIVE END-USER
SOFTWARE LICENSE AGREEMENT

RETSINA(tm)
Reusable Environment for Task Structured Intelligent Network Agents(tm)

IMPORTANT:
PLEASE READ THIS SOFTWARE LICENSE AGREEMENT ("AGREEMENT") CAREFULLY.
Unpacking, examining, or using the attached software and documentation constitutes acceptance of this license agreement.

LICENSE

The CMU Software, together with any fonts accompanying this Agreement, whether on disk, in read only memory or any other media or in any other form (collectively, the "CMU Software") is never sold. It is non-exclusively licensed by Carnegie Mellon University ("CMU") to you solely for your own internal, non-commercial research purposes on the terms of this Agreement. CMU retains the ownership of the CMU Software and any subsequent copies of the CMU Software. The CMU Software and any copies made under this Agreement are subject to this Agreement.

YOU MAY:

1. LOAD and USE the CMU Software as long as the CMU Software is only used on one (1) computer by one (1) user at a time. This license does not allow the CMU Software to exist on more than one (1) computer at a time or on a computer network, including without limitation an intranet network or a local area network.

2. USE the CMU Software solely for your own internal, non-commercial research purposes.

3. COPY the CMU Software for back-up purposes only. You may make one (1) copy of the CMU Software in machine-readable form for back-up purposes. The back-up copy must contain all copyright notices contained in the original CMU Software.

4. TERMINATE this Agreement by destroying the original and all copies of the CMU Software in whatever form.

YOU MAY NOT:

1. Assign, delegate or otherwise transfer the CMU Software, the license (including this Agreement), or any rights or obligations hereunder or thereunder, to another person or entity. Any purported assignment, delegation or transfer in violation of this provision shall be void.

2. Loan, distribute, rent, lease, give, sublicense or otherwise transfer the CMU Software (or any copy of the CMU Software), in whole or in part, to any other person or entity.

3. Copy, alter, translate, de-compile, disassemble, reverse engineer or create derivative works from the CMU Software, including but not limited to, modifying the CMU Software to make it operate on non-compatible hardware.

4. Remove, alter or cause not to be displayed, any copyright notices or startup messages contained in the CMU Software.

5. Export the CMU Software or the product components in violation of any United States export laws.

Title to the CMU Software, including the ownership of all copyrights, patents, trademarks and all other intellectual property rights subsisting in the foregoing, and all adaptations to and modifications of the foregoing shall at all times remain with CMU. CMU retains all rights not expressly licensed under this Agreement. The CMU Software, including any images, graphics, photographs, animation, video, audio, music and text incorporated therein is owned by CMU or its suppliers and is protected by United States copyright laws and international treaty provisions. Except as otherwise expressly provided in this Agreement, the copying, reproduction, distribution or preparation of derivative works of the CMU Software is strictly prohibited by such laws and treaty provisions. Nothing in this Agreement constitutes a waiver of CMU's rights under United States copyright law.

This Agreement and your rights are governed by the laws of the Commonwealth of Pennsylvania. If for any reason a court of competent jurisdiction finds any provision of this Agreement, or portion thereof, to be unenforceable, the remainder of this Agreement shall continue in full force and effect.

THIS LICENSE SHALL TERMINATE AUTOMATICALLY if you fail to comply with the terms of this Agreement.

DISCLAIMER OF WARRANTY ON CMU SOFTWARE

You expressly acknowledge and agree that your use of the CMU Software is at your sole risk.

THE CMU SOFTWARE IS PROVIDED "AS IS" AND WITHOUT WARRANTY OF ANY KIND, AND CMU EXPRESSLY DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF THIRD PARTY RIGHTS. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE CMU SOFTWARE IS BORNE BY YOU. THIS DISCLAIMER OF WARRANTIES, REMEDIES AND LIABILITY ARE FUNDAMENTAL ELEMENTS OF THE BASIS OF THE AGREEMENT BETWEEN CMU AND YOU. CMU WOULD NOT BE ABLE TO PROVIDE THE CMU SOFTWARE WITHOUT SUCH LIMITATIONS.

LIMITATION OF LIABILITY

THE CMU SOFTWARE IS BEING PROVIDED TO YOU FREE OF CHARGE. UNDER NO CIRCUMSTANCES, INCLUDING NEGLIGENCE, SHALL CMU BE LIABLE UNDER ANY THEORY OR FOR ANY DAMAGES INCLUDING, WITHOUT LIMITATION, DIRECT, INDIRECT, GENERAL, SPECIAL, CONSEQUENTIAL, INCIDENTAL, EXEMPLARY OR OTHER DAMAGES) ARISING OUT OF THE USE OF OR INABILITY TO USE THE CMU SOFTWARE OR OTHERWISE RELATING TO THIS AGREEMENT (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION OR ANY OTHER PECUNIARY LOSS), EVEN IF CMU HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THIS LIMITATION MAY NOT APPLY TO YOU.

ADDITIONAL PROVISIONS YOU SHOULD BE AWARE OF

This Agreement constitutes the entire agreement between you and CMU regarding the CMU Software and supersedes any prior representations, understandings and agreements, either oral or written. No amendment to or modification of this Agreement will be binding unless in writing and signed by CMU.

U.S. GOVERNMENT RESTRICTED RIGHTS

If the CMU Software or any accompanying documentation is used or acquired by or on behalf of any unit, division or agency of the United States Government, this provision applies. The CMU Software and any accompanying documentation is provided with RESTRICTED RIGHTS. The use, modification, reproduction, release, display, duplication or disclosure thereof by or on behalf of any unit, division or agency of the Government is subject to the restrictions set forth in subdivision (c)(1) of the Commercial Computer Software-Restricted Rights clause at 48 CFR 52.227-19 and the restrictions set forth in the Rights in Technical Data-Non-Commercial Items clause set forth in 48 CFR 252.227-7013. The contractor/manufacturer of the CMU Software and accompanying documentation is Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pennsylvania 15213, U.S.A.