GMI Agent, API

Application Interface for the GMI Agent Program

General Management Interface Foundation

http://www.gmi-foundation.org

Application Program Interface, Software Description

The GMI Agent program permits users to create applications, which can be remotely uploaded to a GMI Agent program to provide support for general end-point management operations. The specific software and steps to construct an application are described in this document, in a format suitable for C-language and other programmers to begin constructing agent applications immediately.

For a discussion of Agent features, start with the "GMI System Overview" documentation, available within a separate manual. For a discussion of the command API, which can be used to communicate with any application installed at an agent, consult the "GMI Command API" documentation.

The document herein will be of interest to application developers, system managers, and other people wanting to provide special end-point functions for any site running the GMI Agent program, by creating their own management applications that are uploaded to an existing GMI Agent installation.

GMI Development Concepts

GMI Agent programs are typically referred to as "GMI Applications", or "GMI Apps", or (in the context of the GMI project) simply "Apps". These Apps can be simple programs that gather special data or apply special control over the managed platform, or can be somewhat complex systems relying on multiple other applications, running in a multi-threaded environment, possibly with inter-process communication required, and interfaces to third-party systems that are quite complex by themselves.

The basic steps in creating an application are as follows:

1. The programmer creates a DLL, Dynamic Link Library, using the App-src templates that come with the API. This software is likely (but not necessarily) linked with the "GMI-App32.lib" library of functions (listed at the end of this document) to create the DLL.

2. The programmer tests the DLL using a simple stub program: "GMI-Test.exe". This test program, available from the GMI Foundation website, can be run in

debug mode, and allows the end-user to completely test the package design using a symbolic debugger such as provided by Microsoft Visual Studio, or the "Eclipse" IDE.

3. When the DLL is completely tested, the programmer needs to "certify" the application, which creates a "signed" GMI package. This step simply entails contacting GMI Foundation (or one of its technical partners). The programmer provides contact information for the author, and standard documentation on the purpose of the application. The certification process typically takes between one hour and one day, depending upon the accuracy of the information supplied.

4. The certified GMI package is then given back to the developer, where it can be redistributed and uploaded to any number of GMI agent programs on the network to provide new folders and functions to the agent program.

Certification Process

Note that step #3 above, the certification process, is an essential part of the GMI concepts. Certification and signing of your package, with the actual verified author name, contact information, and waivers of responsibility is a required step in the process of creating a redistributable GMI application. This provides essential security against malicious and unidentifiable software hackers.

If you are not prepared to submit your final DLL object code for certification, or identify yourself with proper credentials (i.e. your real name and contact information) you should not attempt to develop any GMI application. Note that GMI Foundation will not certify any anonymous applications. Further note that GMI Foundation verifies all contact information for correctness as part of the certification process.

Scope of This Document

The information in this document is intended to support the first step in the process, which allows the programmer to develop the Agent App using a source code templates and the GMI-App32.lib utility library, to achieve i a particular objective in an end-point management strategy.

For additional information, regarding the GMI-Test.exe test program, certification procedures, and general guidelines to create applications, refer to the GMI Foundation website, or consult web resources available to application developers. Additionally, see the end of this document for specific contact information that will be useful in the development process.

Application Development

The GMI Agent API is intended to be as simple, flexible, and comprehensive as possible. GMI Foundation recognizes that the biggest obstacle to any development activity is the sheer complexity of the activity. Rather than define thousands of obscure functions (as has been a traditional solution of frameworks) GMI supports application development through a small set of commands, only a few of which are essential. This greatly simplifies the development activity.

During normal operation the GMI agent program calls the GMI application when a GMI command program references a folder: Three basic commands must be supported by the application. (1) The "ls" command is used to list directories and fetch object values; (2) the "set" command is used to set object values; (3) the "upload" command is used to upload files to the agent.

To create a working application, the programmer simply creates a DLL that supports these three main functions. When a GMI management program lists a directory, accesses an object, or installs and uninstalls the package, the appropriate function within the DLL is called.

Basic GMI Application Functions

To create a GMI application, the user provides the software logic for five different functions. These function names are formal and invariant, and are introduced below. Later sections of this manual will define these functions in greater detail, including examples to illustrate their basic usage. The principle GMI App functions are:

- **GMI_list_directory()** - This function is required, and populates the directory structure, adding new directories and objects in the management information database. The function should return a pointer to the static directory structure defined in the "gmi-app.h" header file.

- **GMI_get_object() -** This function is required if any object can be read from the agent. It returns a value whenever an object (referenced by a directory element) is accessed. The object will typically be listed in the directory listing defined by the "GMI_list_directory()" function, described above.

- **GMI_set_object() -** This function is required if any object in the directory structure is settable. It is called whenever a value is set in the management information base. The directory element can store a value in static storage, or the set command can initiate an action (such as launch a program.)

- **GMI_put_object() -** This function is required if any object in the directory structure is uploadable. It called whenever an upload command is executed. If no uploadable objects are available, the function should simply return (0).

- **GMI_dwn_object() -** This function is required if any object in the directory structure is downloadable. It called whenever a download command is executed. If no download objects are available, the function should simply return (0).

All of the above functions should exist in the application package. The "GMI_list_directory()" function typically requires simple fill-out of a directory structure (as discussed in later pages). Likewise, depending on the complexity of the application, the other methods may be implemented quickly and with minimal fuss. for the agent to be useful, at least one of the other three functions (or any combination thereof) should have some logic provided.

Other Common GMI Application Functions

In addition to the above functions, several other functions may be useful, and may be required to fulfill the objectives of the developer. These function names are formal and invariant, and are listed below:

Note that not all of the above functions will be necessary for every application. For good programming purposes, the developer may wish to include these functions with the application as stubs, to permit easy modifications to the package.

- **GMI_init() -** This function is optional. If the function exists in the package, it is the first function called when the GMI agent starts the package. The function accepts the name of the GMI module assigned by the agent, in the form "module.gmi". This module typically initializes the data store for the application, as discussed in later sections. This function usually starts any working thread in the app.

- **GMI_term() –** This function is optional. If the function exists in the package, it is called whenever the agent exits or is re-initialized (where re-initialization occurs whenever any application is installed or uninstalled.) In particular, if a working thread exists in the app, this function MUST terminate the thread and wait for the thread to terminate.

- **GMI_install() –** This is function is optional, and is executed when the application is first installed. The function can perform any arbitrary action. If the install is successful the program should return true to permit the application to be initialized by the agent program. If the function is not present in the package, the agent program executes no special installation function.

- **GMI_uninstall() -** This function is optional, and is executed when the application is uninstalled. The function can cleanup temporary files or other files on the system before exiting.

The above functions comprise the entire GMI Agent API Interface (excluding utility libraries). The programmer should provide software logic for each function above (if applicable to the objectives of agent package.) The developer may add any other

functions that are called from the above functions, without limit. The above GMI functions are stubbed out and available in the "gmi-app.c" file, which is a standard part of the GMI Agent API package.

GMIA Link Library

In addition to the above main GMI functions, the Agent API package contains a number of utility functions (which are not strictly required) which may be useful in most programming activities. These utilities are documented in the latter parts of this manual. The utility functions are contained within the "gmi-app32.lib" link library, provided with the API package. All of these utility functions begin with a "GMIA" prefix.

In particular, most applications are designed as a front-end to a datastore, so that the main functions simply set and access data items. Although variations of this methodology exist, most GMI Apps will simply fetch or store data at the request of the manager, with some logic provided to automatically update the datastore with data via a thread, or via logic contained in one of the GMI functions.

Because of this, the GMIA library has multiple useful utilities to interact with a private datastore that is created and maintained by the App. This datastore interface consists of some very simply function calls, introduced below:

- **GMIA_init_dstore().** This function is typically called by the "GMI_init()" function, and creates or defines the data store used by the program. The data store will be an encrypted file in the "Agent\.data" directory of the system.

- **GMIA_get_dstore().** This function retrieves a value from the application data store. The programmer passes a key string to the function, which is almost always (but not necessarily) the "apath" agent path value. The function returns the value associated with the agent path / key in the data store.

- **GMIA_set_dstore().**This function sets a value in the application data store. The value The user passes a key string to the function, which is almost always (but not necessarily) the "apath" agent path value. The function sets the value, and returns.

Description of Functions

The remaining pages of this manual discuss all of the above functions, and additional ancillary (less critical) library functions that are available to the programmer. When reviewing these functions, it may be useful to the developer to study the example programs herein and also the online open source projects that exist for GMI applications. Guidance by GMI Foundation is available on request.

Function: GMI_list_directory ()

A non-trivial "GMI_list_directory ()" function is required for all applications. The function is called when the package is first installed, and subsequently each time the agent starts. This function defines the directory structure of the application through a simple and stylized technique illustrated here, returning a pointer to the calling program that defines the directory structure.

The program simply fills in the "GMI_dirlist_s" array of structures, and returns a pointer to the resulting value. This function typically operates as follows, coded in the highly stylistic manner shown below without much variation between applications except for the actual values assigned to the "GMI_dirlist_s" array:

```
#include "gmi-app.h"

struct GMI_dirlist_s *GMI_list_directory()
{
   /* Return a pointer to the directory structure. */
   /* The directory structure is populated as shown below. */

   static struct GMI_dirlist_s dirlist[GMI_MAX_DIRLIST];
   int i;

   i = 0;

   dirlist[i].path   = "your-app-name";
   dirlist[i].type   = "Folder";
   dirlist[i].access = GMI_ACCESS_FOLDER;
   i++;

   dirlist[i].path   = "your-app-name/value_1";
   dirlist[i].type   = "Text";
   dirlist[i].access = GMI_ACCESS_READ | GMI_ACCESS_WRITE;
   i++;

   /* Add any additional folders or objects here. */

   dirlist[i].path   = NULL;
   dirlist[i].type   = NULL;
   dirlist[i].access = 0;
   i++;

   /* Return a pointer to the static directory structure */
   /* populated  above. */

   return(dirlist);
}
```

Function: GMI_list_directory () (continued)

The "GMI_dirlist_s" structure is defined in the "gmi-app.h" include file. A pointer to a static array of these structures is returned by the function, as shown in the code snipped included above. The members of the structure are described below.

- **char *path;** This structure member is assigned the pathname to the object, which is the unique object identifier for the system. The path should contain simple characters with no spaces, and should not begin with a "/" slash character. (The final path will consist of the name selected by the developer, preceded with the absolute path provided by the "Certify" process discussed earlier.)

- **char *type;** This structure member is a somewhat arbitrary and short text string describing the general type of the path. (The type appears when a management program issues an "ls" command, or fetches a directory list.) The GMI foundation provides consistent names for these types, either "Text", "File", "Integer", "Counter", "Time", "IPaddr", etc. The actual name is completely arbitrary.

- **int access;** This value is an integer value created by a logical "OR" combination of various access flags. The flags indicate the type of operation that will be accepted or rejected by the agent program when the object is requested by a GMI manager program. (The access appears when a management program issues an "ls" command, or fetches a directory list.)

The "access" function is simply a combination of one or more of the following defined flags. The program uses these flags to indicate how the object is handled. For example, attempting to "set" an object without "write" access causes the agent to respond with an error indication. Flags are defined as follows:

- **GMI_ACCESS_FOLDER.** This access indicates that the object is simply a container or folder that contains other objects.

- **GMI_ACCESS_READ.** This access type indicates that the object is readable. The user should supply logic that is executed within the GMI_get_object() function, documented in the next section.

- **GMI_ACCESS_WRITE.** This access type indicates the object is writable. The user should supply logic that is executed within the GMI_set_object() function, documented in later sections.

- **GMI_ACCESS_UPLOAD.** This access type indicates the object is uploadable. The user should supply logic that is executed within the GMI_put_object() function, documented in later sections.

Function: GMI_get_object ()

If one or more objects are readable, then the "GMI_get_object ()" function is required for the application. The function is called when a GMI manager program reads or lists an object value. The function accepts one of the paths defined in the "GMI_list_directory()" function (described earlier) and simply returns the value to the "retval" string, to a maximum of GMI_MAX_STRING (1000 characters.)

Typically, the GMI_get_object() function simply decodes the *apath value (which is one of the paths defined in the GMI_list_directory() function that has been assigned GMI_ACCESS_READ access) and then executes a function. The function returns a value which is copied to the *retval return value. An example is provided below:

```
#include "gmi-app.h"

int GMI_get_object(char *apath, char *retval)
{
   /* For each supported path, when called, return a value. */
   /* More than one value can be returned. The last value that */
   /* is returned should be terminated with an EOF. */

   /* Return stat: 0=unhandled; 1=continue; -1=finished */

   if (stricmp(apath, "your-app-name/value_1") == 0)
   {
      strcpy(retval, "This is value number 1);
      return(EOF);
   }
   else if (stricmp(apath, "your-app-name/value_2") == 0)
   {
      strcpy(retval, "This is value number 2);
      return(EOF);
   }
   else if (stricmp(path, "your-app-name/value_3") == 0)
   {
      strcpy(retval, "This is value number 3);
      return(EOF);
   }
   return(0); /* Object is not handled. */
}
```

As shown above, the function accepts the "path" argument, which is one of the paths that was defined in the "GMI_list_directory()" function presented earlier. The program simply compares the path to an expected value, and executes the particular "case" leg of the if / else if structure. If a path is matched, the "retval" return value is updated with the fetched value. If the path is not matched, the function returns zero.

Function: GMI_get_object () (continued)

Note that the return status value of the function is important, because the "GMI_get_object()" function can return multiple lines of output (not just a single line.) Specifically, the following return status values must be specified.

- **return(EOF)** – The function should return EOF, -1 if there are no other values to be processed. Specifically, if an object folder contains only one line of text, then the function should return EOF. Failure to do so will cause the management system to continuously call the function again until program limits of 1 million lines of output are returned.

- **return(1)** – The function should return positive integer 1 if there are more values to be fetched. Specifically, if the data resides in a file, or there are multiple lines of output, then the function should return positive integer 1 until the last object in the list. When the last object is reached, the function should return EOF as described above.

- **return(0)** – The function should return zero if there is no match for the case select. This causes the agent program to return an error to the management program indicating that the object has no value or is not supported.

A typical implementation of the "GMI_get_object()" function is as follows: when a particular case leg is matched, the "GMI_get_object()" function executes a command, redirects command output to a temporary file, then opens the file and read the contents. Each time the function is called, the next line is returned (and the return status of the function is positive integer 1). When the file is exhausted, the temporary file is deleted and the last line is returned with an EOF flag.

This is easily accomplished by defining a static FILE* type, and executing the "system" command for any command, such as the following: (1) If on entry to the function, the file descriptor is NULL, the "GMI_get_object()" function executes a command and redirects output to a file, and then opens the file and reads the first line; (2) If on entry the file is opened, the program simply gets the next line until EOF, and; (2) when an EOF is encountered, the "GMI_get_object()" function closes the FILE* descriptor and assigns the static file descriptor back to NULL, returning an EOF (and any final value) to the calling program.

Other common variations of the "GMI_get_object()" function can exist. For example, the GMI function can return a value from a data store using the "GMIA_get_dstore()" function, or can return information from an uploaded file using the "GMIA_is_gmi_file()" function. These different variants are documented in online source code available from the GMI foundation, and documented in later sections of this manual.

Function: GMI_set_object ()

If one or more objects are writable then the "GMI_set_object ()" function is required for the application. (Note that if the object is "uploadable", the "GMI_put_object()" function is required also.) The function is called when a GMI manager program sets an object value. The function accepts one of the paths defined in the "GMI_list_directory()" function (described earlier), accepts a value, and simply performs an action such as storing the value.

Typically, the "GMI_set_object()" function simply decodes the *apath value (which is one of the paths defined in the "GMI_list_directory()" function that has been assigned GMI_ACCESS_WRITE access) and then executes a function. The precise action performed depends upon the *apath and the *setvalue passed arguments. An example of usage is shown below:

```
#include "gmi-app.h"

/* Static storage. */

char ST_value_1[GMI_MAX_STRING];
char ST_value_2[GMI_MAX_STRING];
char ST_value_3[GMI_MAX_STRING];

int GMI_set_object(char *apath, char *setval)
{
   /* For each supported path, when called, set a value. */
   /* Only one value can be set at a time. */

   /* Return stat: 0=unhandled; 1=handled; -1=handled w error */

   if (stricmp(path, "your-app-name/value_1") == 0)
   {
      strcpy(ST_value_1, value);
      return(1);
   }
   else if (strcmp(path, "your-app-name/value_2") == 0)
   {
      stricpy(ST_value_2, value);
      return(1);
   }
   else if (strcmp(path, "your-app-name/value_3") == 0)
   {
      stricpy(ST_value_3, value);
      return(1);
   }
   return(0); /* Object is not handled. */
}
```

Function: GMI_set_object () (continued)

As shown above, the function accepts the "apath" argument, which is one of the paths that was defined in the "GMI_list_directory()" function presented earlier. The program simply compares the path to an expected value, and executes the particular "case" leg of the if / else if structure. If a path is matched, the program takes action (in this case by simply storing the value into static storage.) If the path is not matched, the function returns zero.

As with the "GMI_get_object()" function, described earlier, the return status value of this function is significant, and should be used correctly. The set command has the ability to flag a value that is wrong, or out-of-expected range. Specifically, the following return status values must be specified.

- **return(-1)** – The function should return -1 if the value of "setvalue" is not appropriate for the type of operation. For example, if the application expects an IP address, and the management program provides an integer number, the function can reject the value by returning a negative 1. Similarly, if the atoi(setvalue) result is not between 1 and 100, this can be rejected as may be appropriate for the application.

- **return(1)** – The function should return positive integer 1 if and only if the action completed successfully. The management program will then proceed, knowing that the value was successfully set within the agent program.

- **return(0)** – The function should return zero if there is no match for the case select. This causes the agent program to return an error to the management program indicating that the object has no value or is not supported.

The "GMI_set_object()" function effects an action. For example, the system can simply store the value in a data store using the "GMIA_set_dstore()" function (discussed elsewhere) – or the function can perform some real action such as removing a file, executing a program, terminating a program, or any other action that is required by the application.

Note that objects can be settable without being readable, and also note that a typical action may return a value (via the "GMI_get_object()" function) that is independent of the value that was set. Further note that the "set" command can only set a single value; to set an object with multiple values (such as the SysInfo object of the native agent implementation) the "GMI_put_object()" function is required.

Finally, the "GMI_set_object()" function may often operate on an object that is readable, writable, and uploadable. In the case of an object that is both "settable" and "uploadable", the set function should first check to see if an uploaded file exists, and delete this file as appropriate. See additional notes, provided later in this manual.

Function: GMI_put_object ()

Within the GMI construct, there are two main ways to assign an object a value. The first is to set the object to the value using the GMI "set" command (described in the preceding section.) The second way of setting an object to a value is to "upload" a file to the object. The "upload" operation is similar to a "set" command (in that the object can be listed via the "GMI_get_object()" function). However, instead of setting a single line of text, an entire file can be saved.

Typically, the "GMI_put_object()" function accepts an agent path and the name of a temporary file on the system. The function simply decodes the *apath value (which is one of the paths defined in the "GMI_list_directory()" function that has been assigned GMI_ACCESS_UPLOAD access) and then processes the file in some manner, typically storing the file on the system using the "GMIA_ map_new_gmi_filename()" function.

```
#include "gmi-app.h"

int GMI_put_object (char *apath, char *infile)
{
   FILE *fdin, *fdout;
   char outfile[MAX_PATH];

   /* Process the specified file. The *apath value is the */
   /* agent path to be decoded. The *infile value is the */
   /* pathname to a temporary file containing the data. */

   if (stricmp(path, "your-app-name/value_1") == 0)
   {
      /* Create a new GMI file to hold the data. */
      /* This function maps and return the name of a GMI */
      /* file that will contain the data. */

      GMIA_map_new_gmi_file(apath, outfile);

      fdin  = fopen(infile , "rb");
      fdout = fopen(outfile, "wb");
      while (! feof(fdin))
      {
         fputc(fgetc(fdin), fdout));
      }
      fclose(fdin);
      fclose(fdout);
      return(1)
   }
   return(0); /* Object is not handled. */
}
```

Function: GMI_put_object () (continued)

The above function allows the user to upload a file of information. The file is stored in a private directory within the GMI ".data" directory. (the precise name of the file stored is derived by the "GMIA_map_new_gmi_file()" function, documented in later sections, and linked with the program via the "GMI-App32.lib" library.

The contents of the uploaded file can be fetched via the "download" or the "ls" command. (In both cases, the file is fetched using the "GMI_get_object()" function, described earlier.)

The "GMI_get_object()" function accesses the file through a stylistic method described in a later section. The file may be either a text file, or some sort of binary file (such as a PDF file or a compressed file, or even an executable program.)

The return status value of this function is identical to the "GMI_set_object()" function. The "upload" command has the ability to flag a value that is wrong, or out-of-expected range. Specifically, the following return status values must be specified.

- **return(-1)** – The function should return -1 if the contents of the file is not appropriate for the type of operation. For example, if a specific type of configuration file is being uploaded, the "GMI_put_object()" function can check the syntax of the config file and reject it..

- **return(1)** – The function should return positive integer 1 if and only if the action completed successfully. The management program will then proceed, knowing that the file was successfully uploaded to the agent program.

- **return(0)** – The function should return zero if there is no match for the case select. This causes the agent program to return an error to the management program indicating that the object is not uploadable.

Not all applications require "uploadable" objects. However, the ability to upload an entire file of any type is an important supported feature of the GMI-Agent, and expands the role of the program enormously to include config files, executable files, and raw data.

If the application supports the "upload" operation, then simple but very specific constraints are placed on the "GMI_get_object()" and "GMI_set_object()" to support this operation. Specifically, if the uploaded object is to be fetched, the "GMI_get_object()" function must test to see if the current value for the object is a mapped GMI file (using the "GMIA_is_gmi_file(): function) and handle this case appropriately.

Likewise, if the object is both settable and uploadable, the "GMI_set_object()" function must be prepared to unmap the GMI file. These constraints are introduced here, and are documented in a later chapter of this manual.

Function: GMI_dwn_object ()

The "GMI_dwn_object()" function complements the "GMI_put_object()" function, and is called when a file is to be downloaded. The function is usually quite simple, and simply returns the name of the GMI file (saved in the datastore) that is to be downloaded. If the function returns a valid file, the GMI-Agent program performs the download operation for the file, returning the file to the remote command program.

The "GMI_dwn_object()" function accepts an agent path, and returns the name of a GMI file. The file contains the data to be downloaded. Typically, the file is simply a value created via the "GMIA_map_new_gmi_filename()" function, fetched from the datastore. An example of how to implement this function is as follows:

```
#include "gmi-app.h"

int GMI_dwn_object (char *apath, char *file_to_download)
{
   /* Based on the agent path value, return the name */
   /* of the file to download. */

   GMI_get_dstore(apath, file_to_download);
   return(1); /* Object is handled. */
}
```

The above function is quite simple, because the agent program handles all the details of the download operation. The GMI app only needs to supply the name of the file to be downloaded, given by this function.

Since only the application program is immediately aware of the particular filename (and only the application can access its own datastore) the "GMI_dwn_object()" function simply needs to return the GMI file.

Note that the file must be a valid GMI file, ending in a ".dat" suffix, residing in the ".gmi-data" directory of the agent installation. If the file does not exist in the ".gmi-data

Good practice is to return 1 if the object is handled, and zero if otherwise. However, the agent itself will check for the existence of the returned value, and handle the error. Hence, the return value is actually not necessary or used.

Function: GMI_init () and GMI_term()

The "GMI_init()" and "GMI_term()" functions are provided as functions that can be exported to the application. The "GMI_init()" function is called agent startup, and can perform specific tasks associated with the startup process. The "GMI_term()" function is called on agent termination or re-initialization. Not all applications will require any initialization.

The most common use of the "GMI_init()" function is to define a private data store for the program. Data stores are a convenient construct used to store data and flags for the application, as defined in a later section.

The typical implementation of "GMI_init()" function is as follows, which is suitable for most applications, including those that do not necessarily use the GMI data store construct.

```
#include "gmi-app.h"

int GMI_init (char *module_name)
{
   /* Optional. Initialize a data store for this application. */
   /* The function below defines the data store by name. */

   /* Note: The function below performs no action unless */
   /* the applicatiob uses the GMIA_get_dstore() or the */
   /* GMIA_set_dstore()functions in the "GMI-App32.lib" */

   GMIA_init_dstore(module_name);

   return(0);
}
```

The "GMI_init()" function accepts as a single argument the module name, which (in this particular case) uniquely defines the private data store used by the application.

Another common function of the "GMI_init()" function is to start threads, cleanup files, or start processes that communicate with the other application calls through the file system or via other mechanisms, as may be required for the intent and purpose of the application.

The "GMI_init_dstore()" function, referenced above, is documented in a later section, and permits the program to save and retrieve values from non-volatile memory, saved between invocations of the program, and frequently used for special processing. See the section on data stores, later in this manual.

Function: GMI_install() and GMI_uninstall()

The "GMI_install()" and GMI_uninstall() functions are provided as one of the several functions exported to the agent process. These functions are called whenever the agent is installed and uninstalled.

Each function is called without any arguments, and the return value of the application is ignored.

Specifically, the "GMI_install()" function is called whenever the application is first installed via the "install" command of a GMI manager. The function can perform specific tasks associated with the installation.

Similarly, the "GMI_uninstall(): function is called whenever application is uninstalled via the "uninstall" command of the GMI manager. The function can perform specific tasks associated with the uninstallation of the applcation, such as test for and remove configuration data, or other data files on the system required by the application.

Not all applications will require any special installation or uninstallation procedure. For may apps, these two functions will consist of empty stubs, or will be removed from the final DLL.

GMI Application Data Store Functions

Experience shows that many GMI applications make heavy use of a "data store" (also known as an "object store") to save values. The symmetry of the GMI application functions imply the need and utility for some data store to save values (via the "GMI_set_object()" and "GMI_put_object()" functions) which are later retrieved and or operated on.

In particular, values that are saved are often intended to remain non-volatile, i.e. values are often intended to be saved between agent restarts. Therefore, it is useful to have a data store repository as part of an application to permanently save values for the application.

One way to accomplish this is for the application to simply create a file, write values to the file, and read from the file. This is not difficult for any application programmer to implement, and is a very transparent way of handling data issues.

However, the GMI application link library contains multiple functions to manage a private data store for the application for special convenience. These functions maintain encrypted files in the ".data" directory of the agent installation, promoting security and data privacy. Hence, these functions are often used by GMI applications and programmers. The specific GMI Data Store functions are introduced below:

- **GMIA_init_dstore(char *modulename);** This function creates a data store. The function is almost always called exclusively by the "GMI_init()" function, which passes the name of the module to the function to create a unique data store for the application, or reference the existing data store (if a data store already exists.)

- **int GMIA_get_dstore(char *key, char *getvalue);** This function fetches a value from the data store. The "*key" value is almost always an agent path (passed into one of the GMI functions documented earlier.) The value of *getvalue is the text string associated with the key.

- **int GMI_set_dstore(char *key, char *setvalue);** This function sets a value in the data store. The "*key" value is almost always an agent path (passed into one of the GMI functions documented earlier.) The value of *setvalue is the text string associated with the key, which is stored.

- **int GMIA_map_new_gmi_filename(char *apath, char *filename);** This function is used when a file is uploaded to the agent, and an agent path is associated with a file. In practice, this function is used exclusively by the "GMI_put_object()" function to create a file to hold data, and to update the data store with the agent path and filename.

Example: Using Data Store Functions To Save And Get Data

After a data store has been created by the "GMI_init_dstore()" function, as part of the "GMI_init()" function, it becomes a simple procedure to store data in the data store when an object is set, and retrieve the data when the object is listed. A typical implementation is as shown below.

```
#include "gmi-app.h"

/* First, initialize the data store on app startup. */

int GMI_init(char *module)
{
   GMI_init_dstore(module);
   return(0);
}
/* Set the object in the data store when called. */

int GMI_set_object(char *apath, char *setval)
{
   if (stricmp(path, "your-app-name/value_1") == 0)
   {
      GMIA_set_dstore(apath, setval);
      return(1); /* Object successfully set. */
   }
   return(0); /* Object is not handled. */
}
/* Get the object from the data store when called. */

int GMI_get_object(char *apath, char *getval)
{
   if (stricmp(path, "your-app-name/value_1") == 0)
   {
      GMIA_get_dstore(apath, getval);
      return(EOF); /* Object successfully retrieved. */
   }
   return(0); /* Object is not handled. */
}
```

As shown above implementation of get and set commands becomes quite trivial. When an object is to be saved (via a manager "set" command) the "GMI_set_object()" function simply passes the arguments to the "GMIA_set_dstore()" function. When the object is to be retrieved (via a manager "ls" command) the "GMI_get_object()" function simply passes the arguments to the "GMIA_get_dstore()" function. No further action is needed to save and fetch values.

Example: Data Store Functions to Upload Data

The upload functions can also make use of data store functions. In this case, the program saves the data into a file, and the name of the file (which is irrelevant to the programmer, but is a pathname to a time stamped file in the Agent\.data" directory) is saved in the data store. This permits the file to be easily retrieved by the "ls" or "download" functions. Without saving the name of the file in the data store, the unique GMI file name would be lost.

The "GMIA_map_new_gmi_file()" function was previously introduced during the discussion of the "GMI_put_object()" function, and is illustrated below for completeness.

```
#include "gmi-app.h"

int GMI_put_object (char *apath, char *infile)
{
   FILE *fdin, *fdout;
   char outfile[MAX_PATH];

   /* Process the specified file. The *apath value is the */
   /* agent path to be decoded. The *infile value is the */
   /* pathname to a temporary file containing the data. */

   if (stricmp(path, "your-app-name/value_1") == 0)
   {
      /* Create a new GMI file to hold the data. */
      /* This function maps and return the name of a GMI */
      /* file that will contain the data. */

      GMIA_map_new_gmi_file(apath, outfile);

      /* Just copy the infile to the outfile. */

      copy_file(infile, outfile);
      return(1)
   }
   return(0); /* Object is not handled. */
}
```

Specifically, the "GMIA_map_new_gmi_file()" function creates a new time stamped file in the agent ".data" directory, and then assigns this filename to the "apath" key in the data store. Any function that wants to retrieve the file associated with "apath" simply calls "GMIA_get_dstore(apath, file)" to get the name of the file associated with the agent path. This technique permits the "GMI_get_object()" function to access the file if it exists, as illustrated in the next section.

Example: Data Store Functions to Download Data

The "ls" and "download" commands are both implemented by the "GMI_get_object()" function. The function tests to see if the value of the data store for an agent path corresponds to a pathname. If the value is not a pathname, the function simply returns the value of the data store. Otherwise, the function opens a file and reads the data one line at a time until the file has been read and transmitted to the requesting management program.

Consider the complicated case where an agent value can be uploaded, set, downloaded, or listed. The following code snippet (the most complicated and comprehensive so far in this manual) describes an agent called "my-file" that can support all of these functions.

```
#include "gmi-app.h"

int GMI_get_object(char *apath, char *retval)
{
   /* Static open file descriptor. */

   static FILE *fd = NULL;
   char filetype [GMI_MAX_STRING];

   if (  stricmp(apath, "My-App/My-File") != 0))
   {
      /* Not handled. */

      return(0);
   }
   else if (fd != NULL)
   {
      /* A file is opened. Return the next line. */

      if (GMIA_getline(fd, retval) == EOF)
      {
         fclose(fd);
         fd = NULL;
         return(EOF);
      }
      return(1); /* Continue with read. */
   }
```

Example: Download Data (continued)

```
   /* No file is currently opened. */

   GMIA_get_dstore(apath, retval);
   if (*retval == '\0')
   {
      strcpy(retval, "No value."); return(EOF);
   }
   else if (! GMIA_is_gmi_file(retval))
   {
      /* Not an external file. Just return the scalar value. */
      /* Note - the scalar value was provided by the */
      /* GMIA_get_dstore() function above. */

      return(EOF);
   }
   /* This is a GMI file. Get the file type. */

   GMIA_detect_file_type(retval, filetype);
   if (stricmp(filetype, "text") != 0)
   {
      /* Not a text file. */

      strcpy(retval, filetype);
      return(EOF);
   }
   else if ((fd = fopen(retval, "rb")) == NULL)
   {
      /* Anomalous. File has been removed. */

      strcpy(retval, "No value.");
      return(EOF);
   }
   else
   {
      /* File is now opened. Get first line. */

      GMIA_getline(fd, retval);
      return(1); /* Continue. */
   }
   return(0); /* Anomalous. Not reached. */
```

The above example introduces multiple new functions. The operation is explained in detail below.

Example: Download Data (continued)

1. On entry, the function sees if the agent path is handled or not. If the agent path is something other than "My-App/my-file", then the function simply returns zero to indicate the object is not handled. In this case, the agent will continue looking for the agent path, (perhaps supported by some other application) or will return an error to the management program if the agent path is not found.

2. If the apath value is supported by the application, the application checks to see if a file descriptor is already opened from a previous call. If a file descriptor is already opened, the function simply returns the next line of the file (using the "GMIA_getline()" utility function) Note that a file descriptor can only be opened if the file exists, as found later in the function.

3. If the file descriptor is not opened, then the program next gets any value from the application data store by calling "GMIA_get_dstore()". If no value exists (because no value was ever set) the return value is simply the string "No value", which is sent back to the management program.

4. If the "GMIA_get_dstore()" function returns a value, the next test is to see if the value corresponds to a GMI pathname. This can be accomplished with a variety of techniques (such as a system "access()" call on the return value) but the "GMIA_is_gmi_file()" provides convenience to the programmer; the function returns true if the passed argument is a GMI file in the "Agent\.data" directory.

5. If the program is a GMI file, the next step is to determine whether the file can actually be opened or not. (Note that the GMI file can be a text file, but can also be a binary file suitable for download, but not necessarily suitable for listing.) The "GMIA_detect_file_type()" function provides utility in determining the file type; the function returns the keyword "text" if the file is actually a test file, otherwise returns a description of the file and its size.

6. If the file is not a text file, the return value of the "GMIA_detect_file_type()" is returned. The file is not opened. However, if the file is a text file, the program simply opens the text file, assigning the static file descriptor to a non-NULL value, and then gets the first line of the file.

This entire sequence covers the entire range of possibilities associated with uploading, downloading a file, as well as being able to set and get scalar values. The actual software is implemented via the "File" application, available from the GMI Foundation as an open-source programming example.

Multi-Threaded Apps

GMI Apps are quite often multi-threaded, and the GMI-Agent permits the user to create a multi-threaded app (which updates the local datastore, or performs some scheduled activity.) Several simple considerations apply:

1. The working thread in the application should be started in the "GMI_init()" function, which is called when the App is first started.

2. The "GMI_term()" function MUST shutdown the thread when called by the agent. This is required; failure to incorporate a shutdown mechanism for the thread will result in the agent abnormally exiting whenever ANY app is installed or uninstalled in the agent.

An example of implementing a multi-threaded application is shown below:

```
#include "gmi-app.h"
#include <process.h>

int GLOBAL_terminate_thread = 0;
void worker_thread(*void iargs);

int GMI_init (char *module)
{
  /* Start a persistent worker thread */

   GLOBAL_terminate_thread = 0;
   beginthread(worker_thread, (unsigned) 0, (void*) NULL);
   return(0);
}
int GMI_term ()
{
  /* End the worker thread */

   GLOBAL_terminate_thread = 1;
   while (GLOBAL_terminate_thread) Sleep(500);
   return;
}
void worker_thread(*void iargs)
{
   while (! GLOBAL_terminate_thread)
   {
      /* Perform work. */
   }
   GLOBAL_terminate_thread = 0
   return;
}
```

GMIA – GMI-App32.lib Function Calls

To use any function that contains GMIA requires that the programmer link with the "GMI-app32.lib" or "gmi-app.obj" file. These files are available as part of the GMI Foundation, and are small files of basic utility functions. The functions are documented in the "gmi-app.h" include file, and further documented in this section to provide additional explanation.

The "GMI-App32.lib" is available for Windows 32 bit applications, and is linked with the other programmer files to create the final application DLL to be certified.

Programmers should consult the "gmi-app.h" header file, and look at open source examples that illustrate their usage in typical GMI Agent application programming activities.

Function: GMIA_getline()

This function provides general utility in fetching a complete text line from the specified open file. The function is provided for completeness, and does not perform any special processing except fetching up to GMI_MAX_STRING (1000) characters of data from an open file up to but not including the newline delimiter for the line. The function returns EOF at the end of the file, otherwise returns zero. The size of the "line" argument should be GMI_MAX_STRING characters in length, or longer.

```
int GMIA_getline
(
FILE *fd,
char *line
);
```

Function: GMIA_strip_blanks()

This function provides general utility in stripping the leading and trailing blanks from a string, and any multiple blanks found in the string. The function is provided for completeness, to assist with cleanly formatting strings used by the system. The value of "string" is replaced by the formatted value. The size of the "string" argument should be GMI_MAX_STRING characters in length or longer.

```
int GMIA_strip_blanks
(
char *string
);
```

GMIA – GMI-App32.lib Function Calls (continued)

Function: GMIA_is_gmi_file()

This function provides general utility in determining whether a value, typically fetched via the "GMIA_get_dstore()" function, is the pathname to a GMI file that has been uploaded on the system. The function returns true(1) if the "filename" argument is a GMI file residing in the "Agent\.data" directory, otherwise returns false (0). The value of the "filename" argument is not modified at all.

```
int GMIA_is_gmi_file
(
char *filename
);
```

Function: GMIA_init_dstore()

This function is typically called by the "GMI_init()" function, and creates or defines the data store used by the program. The data store will be an encrypted file in the "Agent\.data" directory of the system. The function below takes a single argument, which is the unique GMI application module name, passed directly from the "GMI_init()" function to the function below. This function must be executed once for any other data store function to work.

```
int GMIA_init_dstore
(
char *module_name
);
```

Function: GMIA_get_dstore()

This function retrieves a value from the application data store. The value must be GMI_MAX_STRING (1000) characters or longer. The user passes a key string to the function, which is almost always (but not necessarily) the "apath" agent path value. The function returns the value associated with the agent path / key in the data store. This function will fail if the "GMIA_init_dstore()" function (above) has not been called. On failure, the function returns a zero length value.

```
int GMIA_get_dstore
(
char *key,
char *getvalue
);
```

GMIA – GMI-App32.lib Function Calls (continued)

Function: GMIA_set_dstore()

This function sets a value in the application data store. The value must be GMI_MAX_STRING (1000) chars or less. The user passes a key string to the function, which is almost always (but not necessarily) the "apath" agent path value. The function sets the value, and returns. The function silently fails if the "GMIA_init_dstore()" function (above) as not been called.

```
int GMIA_set_dstore
(
char *key,
char *value
);
```

Function: GMIA_match_path()

This function provides general utility in matching paths, where the *p1 value must be completely contained in the *p2 value. Specifically, if *p1 is the path "misc/test" then the function returns true if *p2 is the path "misc/test/val1" or "misc/test/folder1/val1" but returns false if *p2 is "misc/testvalue" or "misc01/test". This provides a convenient way and reliable method of decoding the "apath" values passed to the various functions and branching the program to handle certain folders.

```
int GMIA_match_path
(
char *p1,
char *p2
);
```

Function: GMIA_map_new_gmi_filename()

This function provides general utility in creating a new file, and then setting the association between the "apath" agent path and that new file. The function is mainly used in the "GMI_put_object()" function, when the user is uploading a file to the system. The function will unlink / delete any existing GMI file associated with the path, create a new GMI file in the "Agent\.data" directory, and then assign the pathname to the specified apath. This facilities handling of the upload process for the programmer by taking care of all house keeping and cleanup activities related to uploading a file.

```
int GMIA_map_new_gmi_filename
(
char *apath,   /* In. */
char *filename /* Out. */
);
```

GMIA – GMI-App32.lib Function Calls (continued)

Function: GMIA_get_tempfile()

This function provides general utility in creating a temporary file in the "Agent\.data" directory, such as a file to temporarily handle the standard output of a program, or operate as an intermediate storage area. Programmers are free to create their own temporary files without this command, but the agent program automatically cleans up temporary files created by this function, when the agent program exits in a normal fashion. Hence, this particular function offers a high degree of convenience to programmers.

```
int GMIA_get_tempfile
(
char *filename /* Out. */
);
```

Function: GMIA_detect_file_type()

This function provides general utility in determining the type of a GMI file. The program looks at the specified diskpath (which is typically a value returned by the "GMIA_get_dstore()" function) and then checks the file to determine the file type. If the file path is a text file, the type is "text". Otherwise, the program attempts to determine the type of file (such as Windows executable, PDF", "GIF", etc.) and then returns with the file type and the size. This provides convenience to the "GMI_get_object()" function in determining how to treat the file, such as whether to open the file and return its contents to the manager (one line at a time) or simply report the file type to the manager program.

```
int GMIA_detect_file_type
(
char *diskpath,
char *filetype
);
```

Function: GMIA_get_client_address()

This function furnishes general utility in returning the IP address of the client that is accessing the data, useful for audit and security purposes. The accepts a pointer to a text string of 20 characters or more, and fills in the last IP address of the client that accessed the system, in standard "N.N.N.N" format.

```
int GMIA_get_client_address
(
char *client_address /* Out. */
);
```

Function: GMIA_get_site_ident()

This function furnishes general utility in returning the "Site Identifier" for the program. This is unique string in the form "AAAA-BBBB-CCCC", which identifies the site: (1) The first segment is a hex checksum of the current host name; (2) the second segment is a hex number indicating the agent install time; (3) the third number is a hex checksum of the current agent working directory. The value can be used by App developers to create a site licensing mechanism for their apps (where the "Site Identifier" is sent to the vendor, who encrypts the value and uses it as a "key" to license an application.)

```
int GMIA_get_site_ident
(
char *site_identifier /* Out. */
);
```

Additional Information

Any programming activity can be frustrating. GMI Foundation is devoted to reducing this frustration by answering programmer questions to the extent feasible in order to promote this technology.

Further information on the Agent API, libraries, programming techniques, code samples, and additional application notes are available at the GMI-Foundation.org website. Application developers are encouraged to contact our organization at the information below, or consult with one of our technical partners. Questions, suggestions, and reasonable criticism are always welcome.

GMI Foundation
http://www.gmi-foundation.org
mailto: info@gmi-foundation.org

About Our Technical Partners

Additional information on the GMI Software, including a selection of useful GMI Apps, as well as professional and support services, is available from various members of the GMI Foundation.

A central clearinghouse for GMI applications is provided by Vallum Software, LLC, which provide support, certification, and development solutions, as well as the "Halo Management System", based on the GMI agent. Visit the link below.

Vallum Software, LLC
http://www.vallumsoftware.com
mailto: info@vallumsoftware.com