The Nagios 2.X Event Broker Module API

Introduction

The purpose of this document is three-fold:

1. Catalog and explain the API used for writing Nagios Event Broker (NEB) Modules and,
2. Touch upon what can and can’t be done with the “stock” NEB Module API and,
3. Identify key Nagios structures and internal Nagios “Helper Routines” that can be used to manipulate Nagios from within an NEB Module.

This document assumes that the reader is familiar with the Nagios Event Broker (NEB) concept and the basic structure of an NEB Module. If not, Taylor Dondich (of OpenGroundWork fame) has created an excellent two-part introduction, available on his company’s website.

Also, while not strictly required, it is very beneficial to have at least a passing knowledge of the C programming language, in order to be able to follow the example code.

Finally, this document will (hopefully) be a continuing work-in-progress. It is currently by no means exhaustive in its treatment of what tricks, hacks and other functionality can be derived via the NEB Module mechanism. Any errors, omissions or bad spelling are mine and I would appreciate all (constructive) feedback on this subject. I can be reached via e-mail: bobi-AT-netshel-DOT-net.

Author, Copyright and License

This document was written by and is copyright © Robert W. Ingraham.

This work is licensed under the Creative Commons Attribution-ShareAlike 2.5 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/2.5/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Author</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>2006-07-28</td>
<td>Robert W. Ingraham</td>
<td>First publication date.</td>
</tr>
</tbody>
</table>
Overview of the NEB Communication Model

From a software point-of-view, Nagios communicates with user-written NEB Modules using a Publish-Subscribe model.

In this model, modules are first identified to Nagios via the “module” directive in the Nagios configuration file. When the Nagios process starts, one of its startup tasks is to load identified modules into its address-space using the dynamic linker facility (much like a DLL under Windows.)

After a module is loaded into Nagios’ memory space, Nagios searches the module for the module’s initialization function, which must be named “nebmodule_init”. The module-writer uses this function to initialize any private data structures and, principally, to subscribe to specific Nagios Events. For example, a module might be only interested in Host and Service Checks and would subscribe to Nagios’ Host Check and Service Check “channels”. The actual mechanism for subscribing to a Nagios Event Channel is that a module provides to Nagios the name of a subroutine defined within the module (known as a Call-Back routine.) When the desired event occurs, Nagios will “call-back” the subscriber’s registered Call-Back routine with the details of the Service Check Event. More details will be given on this shortly.

After initializing itself and subscribing to Nagios Event Channels of interest, the nebmodule_init then returns control back to Nagios.

It is important to know that any number of modules may be loaded and subscribe to the same events. Nagios builds Subscriber Lists for each Nagios Event Channel.

When an event occurs within the Nagios Scheduler – let’s say that it is time to run a particular Service Check as an example – Nagios “publishes” that Service Check Event to it’s Service Check Event Channel; that is, it will walk the list of Service Check Channel subscribers and invoke each subscriber’s Call-Back Routine, one-at-a-time, with the Service Check Event details. In the case of a Service Check Event, the Call-Back Routine will actually be invoked twice:
- Just before Nagios executes the Service Check and,
- Just after the Service Check’s results are processed by Nagios.

The Call-Back routine does with the Service Check information just about whatever it wants to do with it: store it in a database, trigger some external event, attempt to modify Nagios configuration or operation, etc.

After processing the event, the Call-Back routine immediately returns control back to Nagios.

At any point during its operation, a Call-Back routine may unsubscribe itself (or another Call-Back routine) from a Nagios Event Channel.

Finally, when Nagios is getting ready to shutdown, it will invoke each NEB Module’s “de-initialization” routine. Each NEB Module implements a routine called nebmodule_deinit, for this purpose. The primary function of the Module’s nebmodule_deinit routine is to unsubscribe all of its currently-subscribed Call-Back routines, and then de-allocate or clean-up any internal resources that it has used.
**Call Back Routines**

The main purpose of Event Broker call-back routines is to allow an event broker module to register to receive notification of certain pre-defined events from Nagios, as they occur. These events are called “call-back types” within Nagios.

Currently, there are 31 call-back types defined for which an NEB module can register:

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NEBCALLBACK_RESERVED0</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>1</td>
<td>NEBCALLBACK_RESERVED1</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>2</td>
<td>NEBCALLBACK_RESERVED2</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>3</td>
<td>NEBCALLBACK_RESERVED3</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>4</td>
<td>NEBCALLBACK_RESERVED4</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>5</td>
<td>NEBCALLBACK_RAW_DATA</td>
<td>Not implemented</td>
</tr>
<tr>
<td>6</td>
<td>NEBCALLBACK_NEB_DATA</td>
<td>Not implemented</td>
</tr>
<tr>
<td>7</td>
<td>NEBCALLBACK_PROCESS_DATA</td>
<td>Information from the main nagios process. Invoked when starting-up, shutting-down, restarting or abending.</td>
</tr>
<tr>
<td>8</td>
<td>NEBCALLBACK_TIMED_EVENT_DATA</td>
<td>Timed Event</td>
</tr>
<tr>
<td>9</td>
<td>NEBCALLBACK_LOG_DATA</td>
<td>Data being written to the Nagios logs</td>
</tr>
<tr>
<td>10</td>
<td>NEBCALLBACK_SYSTEM_COMMAND_DATA</td>
<td>System Commands</td>
</tr>
<tr>
<td>11</td>
<td>NEBCALLBACK_EVENT_HANDLER_DATA</td>
<td>Event Handlers</td>
</tr>
<tr>
<td>12</td>
<td>NEBCALLBACK_NOTIFICATION_DATA</td>
<td>Notifications</td>
</tr>
<tr>
<td>13</td>
<td>NEBCALLBACK_SERVICE_CHECK_DATA</td>
<td>Service Checks</td>
</tr>
<tr>
<td>14</td>
<td>NEBCALLBACK_HOST_CHECK_DATA</td>
<td>Host Checks</td>
</tr>
<tr>
<td>15</td>
<td>NEBCALLBACK_COMMENT_DATA</td>
<td>Comments</td>
</tr>
<tr>
<td>16</td>
<td>NEBCALLBACK_DOWNTIME_DATA</td>
<td>Scheduled Downtime</td>
</tr>
<tr>
<td>17</td>
<td>NEBCALLBACK_FLAPPING_DATA</td>
<td>Flapping</td>
</tr>
<tr>
<td>18</td>
<td>NEBCALLBACK_PROGRAM_STATUS_DATA</td>
<td>Program Status Change</td>
</tr>
<tr>
<td>19</td>
<td>NEBCALLBACK_HOST_STATUS_DATA</td>
<td>Host Status Change</td>
</tr>
<tr>
<td>20</td>
<td>NEBCALLBACK_SERVICE_STATUS_DATA</td>
<td>Service Status Change</td>
</tr>
<tr>
<td>21</td>
<td>NEBCALLBACK_ADAPTIVE_PROGRAM_DATA</td>
<td>Adaptive Program Change</td>
</tr>
<tr>
<td>22</td>
<td>NEBCALLBACK_ADAPTIVE_HOST_DATA</td>
<td>Adaptive Host Change</td>
</tr>
<tr>
<td>23</td>
<td>NEBCALLBACK_ADAPTIVE_SERVICE_DATA</td>
<td>Adaptive Service Change</td>
</tr>
<tr>
<td>24</td>
<td>NEBCALLBACK_EXTERNAL_COMMAND_DATA</td>
<td>External Command Processing</td>
</tr>
<tr>
<td>25</td>
<td>NEBCALLBACK_AGREGATED_STATUS_DATA</td>
<td>Aggregated Status Dump</td>
</tr>
<tr>
<td>26</td>
<td>NEBCALLBACK_RETENTION_DATA</td>
<td>Retention Data Loading and Saving</td>
</tr>
<tr>
<td>27</td>
<td>NEBCALLBACK_CONTACT_NOTIFICATION_DATA</td>
<td>Contact Notification Change</td>
</tr>
</tbody>
</table>
Each call back type is accompanied with an event-specific data structure.

For example, the NEBCALLBACK_SERVICE_CHECK_DATA call-back type is always accompanied by a `nebstruct_service_check_data` structure:

```c
/* service check structure */
typedef struct nebstruct_service_check_struct{
    int type;
    int flags;
    int attr;
    struct timeval timestamp;
    char *host_name;
    char *service_description;
    int check_type;
    int current_attempt;
    int max_attempts;
    int state_type;
    int state;
    int timeout;
    char *command_name;
    char *command_args;
    char *command_line;
    struct timeval start_time;
    struct timeval end_time;
    int early_timeout;
    double execution_time;
    double latency;
    int return_code;
    char *output;
    char *perf_data;
}nebstruct_service_check_data;
```

So, when your NEB modules registers a call-back routine with Nagios to receive notifications about service check events, your call-back routine will receive two pieces of information:

1. The Call-Back Type (In this case NEBCALLBACK_SERVICE_CHECK_DATA) and,
2. A pointer to a `nebstruct_service_check_data` structure, containing some relevant details about the service check.

We’ll discuss this data structure in some detail, further on. If you’re curious, Appendix A is a catalog of Call-Back Types and their respective data structures.

The Nagios call-back mechanism is one-way, informational-only. That is, there is currently no way for a call-back routine to alter the operation of Nagios through the call-back mechanism itself. To alter the operation of Nagios, a call-back routine must alter global Nagios data structures while it has control from Nagios. For example, to dynamically add a new service definition to Nagios, a call-back routine would invoke the “add_service()” helper function, among other things.

Since Nagios is currently a single, monolithic scheduling process with global control structures, a call-back routine must observe the following rules of “good citizenship”:

- Always return control back to Nagios.
- Spend as little time as possible in the call-back routine; i.e., return control to Nagios as quickly as possible.
- Be careful when modifying the global control structures.
- Where possible, always use the existing Nagios helper functions provided to interact with the global control structures.

**Call-Back Registration (Subscribing to a Nagios Event Channel):**

Call back routines are registered with Nagios usually within the module’s initialization function (`nebmodule_init`). Here is an example initialization routine which registers for service checks:

```c
static nebmodule *my_module_handle;

int nebmodule_init (int flags, char *args, nebmodule *handle) {
    my_module_handle = handle; // Save our module handle in our own global variable - we’ll need it later

    // Register our service check event handler
    neb_register_callback(NEBCALLBACK_SERVICE_CHECK_DATA, handle, 0, ServiceCheckHandler);

    // Always return OK (zero) if your module initialized properly;
    // Otherwise, your module will not be loaded by Nagios.
    return OK;
}

// Our Service Check Call-Back Routine:
static int ServiceCheckHandler (int callback_type, void *data) {

    // Cast the data structure to the appropriate data structure type
    nebstruct_service_check_data *ds = (nebstruct_service_check_data *)data;

    // Now we can access information about this service check that Nagios
```
// is about to execute. For example:
//
// ds->host_name
// ds->command_name
// ds->command_args
// Etc...
//
// Appendix A contains a catalog of call-back-type-specific data structures.

// Always return OK (zero) for success. Although the call-back return code
// is currently ignored by Nagios, it may be utilized in the future.
return OK;
}

There are a couple of things to notice about the above call back registration:

The same event handler may be registered for multiple events. For example, we could have registered one event handler, say ObjectEventHandler, for both Host and Service checks, among others. What makes this possible is the fact that the call back routine receives the call-back type as the first parameter. This allows you to write a multi-event handler in the following manner:

// Our Multi-Event Call-Back Routine:
static int ObjectEventHandler (int callback_type, void *data) {
//
// Invoke call-back-type-specific handling for this event:
switch (callback_type) {
    case NEBCALLBACK_SYSTEM_COMMAND_DATA:
        handleMessageCommand((nebstruct_system_command_data *)data);
        break;
    case NEBCALLBACK_EVENT_HANDLER_DATA:
        handleEventHandler((nebstruct_event_handler_data *)data);
        break;
    case NEBCALLBACK_NOTIFICATION_DATA:
        handleNotification((nebstruct_notification_data *)data);
        break;
    case NEBCALLBACK_SERVICE_CHECK_DATA:
        handleServiceCheck((nebstruct_service_check_data *)data);
        break;
    case NEBCALLBACK_HOST_CHECK_DATA:
        handleHostCheck((nebstruct_host_check_data *)data);
        break;
    default: // Unknown: Did we register for this?
        write_to_logs_and_console("ObjectEventHandler: Unhandled event", NSLOG_RUNTIME_WARNING, TRUE);
}
// Always return OK (zero) for success. Although the call-back return code
// is currently ignored by Nagios, it may be utilized in the future.
return OK;
}

When the `nebmodule_init` routine registers a call-back function (i.e., subscribes to a Nagios Event Channel), it uses the following registration function:

```c
int neb_register_callback(int callback_type, void *mod_handle, int priority, int (*callback_func)(int,void *));
```

The parameters are:

- `int callback_type;`  
  One of the thirty-one pre-defined callback types defined in the preceding Table of Call-Back Types.

- `void *mod_handle;`  
  The module handle pointer that is passed into the `nebmodule_init` function by Nagios.

- `int priority;`  
  An integer priority. This interesting item allows module writers to prioritize the chain of callback routines registered for a given event. That is, it lets you specify which callback routine gets called first, then second, third and so forth. For example, a callback routine registered for service checks with a priority of 1 will be invoked before another callback routine with priority 2.

  There is no min/max limitation on the range of priority values, except for the min/max size of an integer as defined by your OS (i.e., 32-bit ints vs. 64-bits ints).

  Priorities can be positive, zero or negative.

- `int (*callback_func)(int,void *);`  
  This is a pointer to your callback routine. Notice that the callback routine is expected to return an integer result code; although it is currently neither examined nor used by Nagios.

  Also note that the call-back routine should expect to receive two input values: an integer `callback_type` (as discussed above,) and a void pointer which must be cast to the relevant, callback-type-specific data structure.
Appendix A contains a catalog of call-back-type-specific data structures.

Also notice that the call-back routine is declared as “static”. In C programming, this ensures that the call-back function name is not visible outside of the source file in which it is declared. The reason for this is to avoid conflicts with function names within the “global” Nagios name space; i.e., it reduces global name space pollution and eliminates the possibility of a conflict between the name of your call-back functions and the names of any internal Nagios functions.

**Call-Back Routine Invocation:**

Earlier, we discussed the fact that when a call-back routine is invoked, it receives two parameters:

```c
static int myCallBackroutine (int callback_type, void *data);
```

Since we’ve already discussed the meaning and values of the `callback_type` parameter, let’s now dig a little deeper into the call-back type-specific data structure that is passed into each call-back routine as the second parameter:

Although each data structure is unique to the call-back type it accompanies, there are several variables at the beginning of each data structure that are common to all of them. Looking at a subsection of the `nebstruct_service_check_data` structure as an example, we see that these variables are:

```c
/* service check structure */
typedef struct nebstruct_service_check_struct{
    int    type;
    int    flags;
    int    attr;
    struct timeval timestamp;

    (service-check-specific variables omitted..)
}nebstruct_service_check_data;
```

The meaning and use of these common variables is detailed in the following table:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>int</td>
<td>This is arguably the most useful of the common variables. The purpose of the type variable is to give more detailed information about the call-back-type event.</td>
</tr>
</tbody>
</table>

For example, when your call-back routine is registered for and receives the NEBCALLBACK_SYSTEM_COMMAND_DATA call-back type, the “type” variable will tell you whether Nagios is about to execute the system command (type ==
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>flags</strong></td>
<td><strong>int</strong></td>
</tr>
<tr>
<td>Currently, the flags variable is only used in conjunction with the NEBCALLBACK_PROCESS_DATA call-back type, usually to let you know whether a shutdown/restart was Nagios or User initiated. All other call-back types currently set this value to NEBFLAG_NONE (zero).</td>
<td></td>
</tr>
</tbody>
</table>

| **attr**   | **int** |
| The attr variable is used to provide further information about the event type specified in the “type” variable. It is currently only used in conjunction with three call-back types: |
| 1. NEBCALLBACK_PROCESS_DATA – to tell you whether a shutdown/restart was normal or abnormal. |
| 2. NEBCALLBACK_FLAPPING_DATA – to tell you whether flapping stopped normally or was disabled. |
| 3. NEBCALLBACK_DOWNTIME_DATA – to tell you whether scheduled downtime stopped normally or was disabled. All other call-back types currently set this value to NEBATTR_NONE (zero). |

| **struct timeval** | **timestamp** |
| This is the time stamp that Nagios places on the event just prior to passing it to the call-back routines. It represents the current time in “UNIX time”. The timeval structure looks like: |
| struct timeval { |
|     long tv_sec; /* seconds */ |
|     long tv_usec; /* microseconds */ |
| } |
| and gives the number of seconds and microseconds since the Epoch. |

As an example of how one might use these common variables, let’s re-visit our original service check call-back routine:

```c
// Our Service Check Call-Back Routine, Second Version:
static int ServiceCheckHandler (int callback_type, void *data) {
```
// Cast the data structure to the appropriate data structure type
nebstruct_service_check_data *ds = (nebstruct_service_check_data *)data;

char logMsg[1024]; // Used for formatting log messages

// You can use the following Nagios global variable to identify
// how many active service checks are currently running.
extern int currently_running_service_checks;

// Many of the members of the nebstruct_service_check_data structure are
// simply copied from Nagios’ internal service structure. However, there
// is other useful service information which is *not* copied. So, to
// obtain direct access to this structure, we do the following:

service *svc;

if ((svc = find_service (ds->host_name, ds->service_description)) == NULL) {
    // ERROR - This should never happen here: The service was not found.
    sprintf(logMsg, “ServiceCheckHandler: Could not find service %s for host %s”,
           ds->host_name, ds->service_description);
    write_to_logs_and_console(logMsg, NSLOG_RUNTIME_WARNING, TRUE);
    return OK;
}

// Now, we can dynamically examine (or twiddle with,) the service definition.
//
// For example, let’s see if this service check is accepting passive checks:

if (svc->accept_passive_service_checks == FALSE) {
    // Nope, so let’s change it.
    svc->accept_passive_service_checks = TRUE;
}

// Examples of other interesting items in the internal service structure:
//
// svc->next_check - UNIX timestamp of when this service is next scheduled to execute
// svc->checks_enabled - TRUE/FALSE
// svc->check_interval
// svc->latency - service latency (represented as a “double” variable)

// Now, use the “type” common variable to see if we are being notified before or after
// the service check execution:
switch (ds->type) {

    case NEBTYPENET_SVCHECK_INITIATE:
        // Now let’s do something naughty and change the service check command
        // just BEFORE Nagios executes it. Note that at this point, Nagios has
        // already substituted-in all of the service check arguments.
        //
        // WARNING: The command_line buffer has a max size of MAX_COMMAND_BUFFER
        // (currently 8,192) bytes, so be sure not to overrun it!
        //
        // CAVEAT: Since multiple call-back routines may be registered for this
        // event, all call-back routines “down-stream” from us will now see this
        // modified command (instead of the original.) Furthermore, any one of
        // these down-stream call-back routines can also modify the command line
        // string, so unless you know for sure what all of your loaded NEB modules
        // are doing with this event, your command line changes may not survive!

        strncpy(ds->command_line, "/usr/local/nagios/libexec/naughty.pl", MAX_COMMAND_BUFFER);
        ds->command_line[MAX_COMMAND_BUFFER-1] = ‘\’;  // Null-terminate for safety

        break;

    case NEBTYPENET_SVCHECK_PROCESSED:
        // The service check command has been executed and its result has
        // been retrieved from the child process. No we can examine it.
        //
        // However, there is nothing we can do at this point, in this call-
        // back routine, to override the result that Nagios will use in
        // it’s processing (unless we modify the Nagios source file checks.c)
        //
        // The following example code does nothing useful, but provides some
        // examples of using the service data structure elements.

        // See if our current state is soft or hard.
        if (ds->state_type == SOFT_STATE)
            // do something about a soft state...
        else
            // we’re in a hard state

        // Examine our current service state
        switch (ds->state) {
            case STATE_CRITICAL:
                // handle critical state
break;
case STATE_WARNING:
    // handle warning state
    break;
case STATE_UNKNOWN:
    // handle unknown state
    break;
case STATE_OK:
    // Everything’s okie-dokie
    break;
default:
    // Should never happen...
}

// See if there’s been a state change:

// NOTE: The data structure supplied by the NEB only
// contains the current service state. To compare
// against the previous service state, we have to appeal
// to Nagios’ internal service structure (which we
// located previously in this code example.)

if (ds->state != svc->last_state) {
    // We’ve had a state change!
}
break;

case NEBTYPE_SERVICECHECK_RAW_START:
    // This has not been implemented as of Nagios 2.3
    break;

case NEBTYPE_SERVICECHECK_RAW_END:
    // This has not been implemented as of Nagios 2.3
    break;

default:
    // ERROR - We’ve received an unknown (to us) event type
    sprintf(logMsg, "ServiceCheckHandler: Unknown event type for service %s for host %s",
            ds->host_name, ds->service_description);
    write_to_logs_and_console(logMsg, NSLOG_RUNTIME_WARNING, TRUE);
    return OK;
}

// Always return OK (zero) for success. Although the call-back return code
// is currently ignored by Nagios, it may be utilized in the future.
return OK;
}

**Call-Back De-Registration (Unsubscribing to a Nagios Event Channel):**

When the `nebmodule_deinit` routine de-registers a call-back function (i.e., unsubscribes to a Nagios Event Channel), it uses the following de-registration function:

```c
int neb_deregister_callback(int callback_type, int (*callback_func)(int,void *));
```

The parameters are:

- `int callback_type;` One of the thirty-one pre-defined callback types defined in the preceding Table of Call-Back Types.
- `int (*callback_func)(int, void *);` This is a pointer to your callback routine. Notice that the callback routine is expected to return an integer result code; although it is currently neither examined nor used by Nagios.

Also note that the call-back routine should expect to receive two input values: an integer `callback_type` (as discussed above,) and a void pointer which must be cast to the relevant, callback-type-specific data structure.

Appendix A contains a catalog of call-back-type-specific data structures.

As noted earlier, call-back routines can also be registered and de-registered at any time within a call-back routine in your module.

**The Module Information Function**

There is one other NEB Module function that has not yet been discussed. It is used to register information about your module with Nagios:

```c
int neb_set_module_info(void *handle, int type, char *data);
```

The parameters are:
void *mod_handle;  
/* The module handle pointer that you received from Nagios in your nag_sys_modinfo function. */

int type;  
/* This integer specifies which (of the six possible) pieces of module information you wish to set: Title, Author, Copyright, Version, License, Description. The table below gives both the integer and mnemonic representations you can use for this parameter. */

char *data;  
/* This is a string containing the information you wish Nagios to associate with your module. */

<table>
<thead>
<tr>
<th>Index</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NEBMODULE_MODINFO_TITLE</td>
</tr>
<tr>
<td>1</td>
<td>NEBMODULE_MODINFO_AUTHOR</td>
</tr>
<tr>
<td>2</td>
<td>NEBMODULE_MODINFO_COPYRIGHT</td>
</tr>
<tr>
<td>3</td>
<td>NEBMODULE_MODINFO_VERSION</td>
</tr>
<tr>
<td>4</td>
<td>NEBMODULE_MODINFO_LICENSE</td>
</tr>
<tr>
<td>5</td>
<td>NEBMODULE_MODINFO_DESC</td>
</tr>
</tbody>
</table>

*Table of Module Information Types*

It would seem that a module writer would use this function in the <code>nag_sys_modinfo</code> routine. Here are some examples of its use:

```c
neb_set_module_info(my_module_handle, NEBMODULE_MODINFO_TITLE, "Demo NEB Module");
neb_set_module_info(my_module_handle, NEBMODULE_MODINFO_AUTHOR, "Joe Module-Writer");
neb_set_module_info(my_module_handle, NEBMODULE_MODINFO_COPYRIGHT, "© 2006 JMW & Friends");
neb_set_module_info(my_module_handle, NEBMODULE_MODINFO_VERSION, "1.0.0");
neb_set_module_info(my_module_handle, NEBMODULE_MODINFO_LICENSE, "GPL Version 2");
neb_set_module_info(my_module_handle, NEBMODULE_MODINFO_DESC, "A demonstration module");
```

Currently, there appears to be no API function to enumerate the list of loaded modules nor is there a function to look-up a module by information type.

However, the list of loaded modules is also global so, by way of example, we can perform the enumeration function in the following manner:

```c
static int neb_enum_modules (void) {
    extern nebmodule *neb_module_list;  // The linked-list of NEB modules
    nebmodule *temp_module;  // Temp pointer
    int nebmod_count = 0;  // Count the number of active modules
```
char logMsg[1024]; // Used for formatting log messages

// Traverse the NEB Module List
for(temp_module=neb_module_list; temp_module; temp_module=temp_module->next) {
    // Skip modules that are not loaded
    if(temp_module->is_currently_loaded==FALSE)
        continue;

    // Skip modules that do not have a valid handle
    if(temp_module->module_handle==NULL)
        continue;

    // Increment the active module counter
    nebmod_count++;

    // Log the module title - *if* it’s been set
    if (temp_module->info[NEBMODULE_MODINFO_TITLE] != NULL)
        sprintf(logMsg, “Found module: %s”, temp_module->info[NEBMODULE_MODINFO_TITLE]);
    else
        sprintf(logMsg, “Found module: NoTitle #%d”, nebmod_count);

    write_to_logs_and_console(logMsg, NSLOG_INFO_MESSAGE, TRUE);
}

return nebmod_count; // Return the number of active modules

Likewise, we can implement a module lookup function in the following manner:

static nebmodule *neb_find_module (int type, char *data) {

    extern nebmodule *neb_module_list; // The linked-list of NEB modules
    nebmodule *temp_module; // Temp pointer

    // Validate our parameters
    if (type < 0 || type > NEBMODULE_MODINFO_NUMITEMS || !data) {
        write_to_logs_and_console(“neb_find_module: Invalid type or data parameter”, NSLOG_RUNTIME_WARNING, TRUE);
        return OK;
    }

    // Traverse the NEB Module List
    for(temp_module=neb_module_list; temp_module; temp_module=temp_module->next) {
// Skip modules that are not loaded
if (temp_module->is_currently_loaded==FALSE)
    continue;

// Skip modules that do not have a valid handle
if (temp_module->module_handle==NULL)
    continue;

// Compare the desired module information type
if (temp_module->info[type] != NULL && !strcmp(temp_module->info[type], data))
    break; // Success: We have a match

    return temp_module; // Return a pointer to the module, if found; NULL otherwise

**NEB API Summary**

In summary, the current NEB Module writer’s API “officially” consists of the following five functions:

<table>
<thead>
<tr>
<th>NEB Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>nemodule_init</td>
<td>Your Module’s Initialization Routine</td>
</tr>
<tr>
<td>nebmodule_deinit</td>
<td>Your Module’s De-Initialization Routine</td>
</tr>
<tr>
<td>neb_register_callback</td>
<td>Used to subscribe to Nagios Event Channels</td>
</tr>
<tr>
<td>neb_deregister_callback</td>
<td>Used to Unsubscribe to Nagios Event Channels</td>
</tr>
<tr>
<td>neb_set_module_info</td>
<td>Used to register information about your module with Nagios</td>
</tr>
</tbody>
</table>

I say “officially” since, unofficially, the module writer is free to invoke any of the internal Nagios routines to examine, modify or otherwise interact with the Nagios engine.

Obviously, extreme care should be taken when utilizing any of the internal Nagios functions or directly accessing global Nagios data structures since, without a reasonable knowledge of Nagios’ inner-workings, ‘bad-things can happen with regard to the stability and integrity of the Nagios engine.

In order to present a view of the available Nagios global data structures and internal functions, I have created Appendix B in this document; which serves to catalog and discuss both the global Nagios data structures and the internal “Helper Routines” that operate on them.
Appendix A: Catalog of NEB Call-Back Types and their Associated Data Structures

Purpose

This table presents a catalog of each of the NEB Call-Back Types (Nagios Event Channels) and any relevant information associated with them.

Note that, when describing the associated data structure for a Call-Back Type, the common variables (i.e., common to all call-back data structures,) are not explained, since they have the same purpose regardless of Call-Back Type:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>int</td>
<td>This is arguably the most useful of the common variables. The purpose of the type variable is to give more detailed information about the call-back-type event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For example, when your call-back routine is registered for and receives the NEBCALLBACK_SYSTEM_COMMAND_DATA call-back type, the “type” variable will tell you whether Nagios is about to execute the system command (type == NEBTYPE_SYSTEM_COMMAND_START) or has just completed execution of the system command (type == NEBTYPE_SYSTEM_COMMAND_END). This is useful for perhaps dynamically modifying the command just before it is executed; or for receiving the results of the completed/timed-out command before Nagios acts upon them (although, with the way Nagios currently handles this call-back, there isn’t really much you can do to override the result status of the command without modifying the Nagios sources directly.)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>As a further example, the NEBCALLBACK_DOWNTIME_DATA call-back type will set this type variable to let you know if the scheduled downtime is being added, deleted, loaded, started or stopped.</td>
</tr>
<tr>
<td>flags</td>
<td>int</td>
<td>Currently, the flags variable is only used in conjunction with the NEBCALLBACK_PROCESS_DATA call-back type, usually to let you know whether a shutdown/restart was Nagios or User initiated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other call-back types currently set this value to NEBFLAG_NONE (zero).</td>
</tr>
<tr>
<td>attr</td>
<td>int</td>
<td>The attr variable is used to provide further information about the event type specified in the “type” variable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>It is currently only used in conjunction with three call-back types:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4. NEBCALLBACK_PROCESS_DATA – to tell you whether a shutdown/restart was normal or abnormal.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5. NEBCALLBACK_FLAPPING_DATA – to tell you whether flapping stopped normally or was disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6. NEBCALLBACK_DOWNTIME_DATA – to tell you whether scheduled downtime stopped normally or was disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other call-back types currently set this value to NEBATTR_NONE (zero).</td>
</tr>
<tr>
<td>struct timeval</td>
<td>timestamp</td>
<td>This is the time stamp that Nagios places on the event just prior to passing it to the call-back routines. It represents the current time in “UNIX time”.</td>
</tr>
</tbody>
</table>
The timeval structure looks like:

```c
struct timeval {
    long tv_sec;  /* seconds */
    long tv_usec; /* microseconds */
};
```

and gives the number of seconds and microseconds since the Epoch.
A.0 NEBCALLBACK_RESERVED0

Description
This Call-Back type is reserved for future use.

Data Structure
N/A

Invocation
N/A

Relevant Internal Structures
N/A

Examples
N/A
A.1 NEBCALLBACK_RESERVED1

Description

This Call-Back type is reserved for future use.

Data Structure

N/A

Invocation

N/A

Relevant Internal Structures

N/A

Examples

N/A
A.2  NEBCALLBACK_RESERVED2

Description

This Call-Back type is reserved for future use.

Data Structure

N/A

Invocation

N/A

Relevant Internal Structures

N/A

Examples

N/A
A.3  NEBCALLBACK_RESERVED3

Description

This Call-Back type is reserved for future use.

Data Structure

N/A

Invocation

N/A

Relevant Internal Structures

N/A

Examples

N/A
A.4  NEBCALLBACK_RESERVED4

Description

This Call-Back type is reserved for future use.

Data Structure

N/A

Invocation

N/A

Relevant Internal Structures

N/A

Examples

N/A
A.5  NEBCALLBACK_RAW_DATA

Description

This Call-Back type is not implemented.

Data Structure

N/A

Invocation

N/A

Relevant Internal Structures

N/A

Examples

N/A
A.6  NEBCALLBACK_NEB_DATA

Description

This Call-Back type is not implemented.

Data Structure

N/A

Invocation

N/A

Relevant Internal Structures

N/A

Examples

N/A
A.7  NEBCALLBACK_PROCESS_DATA

Description

This Call-Back Type delivers events relevant to the operation of the main Nagios process (e.g., startup, initialization, shutdown, abend, etc.)

Data Structure

/* process data structure */
typedef struct nebstruct_process_data{
  int type;
  int flags;
  int attr;
  struct timeval timestamp;
}nebstruct_process_data;

Invocation

Note: The following Event Types have been arranged in the chronological order in which they will be delivered to your Call-Back routine during a “normal” start-up (i.e., no abends.)

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBTYPE_PROCESS_PRELAUNCH</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td>Called prior to reading/parsing the object configuration files.</td>
</tr>
<tr>
<td>NEBTYPE_PROCESS_START</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td>Called after reading all configuration objects and after passing the pre-flight check. Called before entering daemon mode, opening command pipe, starting worker threads, initializing the status, comments, downtime, performance and initial host/service state structures.</td>
</tr>
<tr>
<td>NEBTYPE_PROCESS_DAEMONIZE</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td>Called right after Nagios successfully “daemonizes”; that is, detaches from the controlling terminal and is running in the background.</td>
</tr>
<tr>
<td>NEBTYPE_PROCESS_EVENTLOOPSTART</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td>Called immediately prior to entering the main event execution</td>
</tr>
<tr>
<td>Event Type</td>
<td>NEBFLAG</td>
<td>NEBATTR</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------------------</td>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NEBTYPE_PROCESS_EVENTLOOPEND</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td>Called immediately after exiting the main event execution loop (due to either a shutdown or restart.)</td>
</tr>
<tr>
<td>NEBTYPE_PROCESS_SHUTDOWN</td>
<td>NEBFLAG_PROCESS_INITIATED</td>
<td>NEBATTR_SHUTDOWN_NORMAL</td>
<td>Invoked if exiting due to either a process initiated (abnormal) or a user-initiated (normal) shutdown.</td>
</tr>
<tr>
<td>NEBTYPE_PROCESS_RESTART</td>
<td>NEBFLAG_USER_INITIATED</td>
<td>NEBATTR_RESTART_NORMAL</td>
<td>Invoked if exiting due to a user-initiated restart. Always invoked after NEBTYPE_PROCESS_EVENTLOOPEND.</td>
</tr>
</tbody>
</table>

**Relevant Internal Structures**

N/A

**Examples**

None.
A.8 NEBCALLBACK_TIMED_EVENT_DATA

Description

Notifies a call-back routine of timed-event events.

Since Nagios is, at its core, one big timed-event-driven loop, all actions taken by Nagios are considered “timed events”. Therefore, a call-back routine registered for this Call-Back Type will be invoked quite often.

Data Structure

```c
/* timed event data structure */
typedef struct nebstruct_timed_event_struct{
  int type;
  int flags;
  int attr;
  struct timeval timestamp;

  int event_type;
  int recurring;
  time_t run_time;
  void *event_data;
}nebstruct_timed_event_data;
```

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>event_type</td>
<td>int</td>
<td>Defines the type of event being added, deleted, executed, etc. See the “Table of event_types” below.</td>
</tr>
<tr>
<td>recurring</td>
<td>int</td>
<td>Boolean (TRUE or FALSE). Determines whether the event is automatically re-scheduled by Nagios after it is executed.</td>
</tr>
<tr>
<td>run_time</td>
<td>time_t</td>
<td>The time at which this event is next scheduled to run (in UNIX time).</td>
</tr>
<tr>
<td>event_data</td>
<td>void *</td>
<td>Point to an event-specific data that the event will use when it executes. See the “Table of event_types” below.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Event Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>EVENT_SERVICE_CHECK</td>
<td>active service check</td>
<td>Pointer to internal Nagios service structure</td>
</tr>
<tr>
<td>1</td>
<td>EVENT_COMMAND_CHECK</td>
<td>external command check</td>
<td>None</td>
</tr>
<tr>
<td>2</td>
<td>EVENT_LOG_ROTATION</td>
<td>log file rotation</td>
<td>None</td>
</tr>
<tr>
<td>3</td>
<td>EVENT_PROGRAM_SHUTDOWN</td>
<td>program shutdown</td>
<td>None</td>
</tr>
<tr>
<td>4</td>
<td>EVENT_PROGRAM_RESTART</td>
<td>program restart</td>
<td>None</td>
</tr>
<tr>
<td>5</td>
<td>EVENT_SERVICE_REAPER</td>
<td>reaps results from service checks</td>
<td>None</td>
</tr>
<tr>
<td>Event Type</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVENT ORPHAN_CHECK</td>
<td>checks for orphaned service checks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVENT RETENTION_SAVE</td>
<td>save (dump) retention data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVENT STATUS_SAVE</td>
<td>save (dump) status data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVENT SCHEDULED_DOWNTIME</td>
<td>scheduled host or service downtime</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVENT SFRESHNESS_CHECK</td>
<td>checks service result &quot;freshness&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVENT_EXPIRE_DOWNTIME</td>
<td>checks for (and removes) expired scheduled downtime</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVENT_HOST_CHECK</td>
<td>active host check</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVENT HFRESHNESS_CHECK</td>
<td>checks host result &quot;freshness&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVENT RESCHEDULE_CHECKS</td>
<td>adjust scheduling of host and service checks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVENT_EXPIRE_COMMENT</td>
<td>removes expired comments</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVENT_SLEEP</td>
<td>asynchronous sleep event that occurs when event queues are empty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVENT_USER_FUNCTION</td>
<td>USER-defined function (modules)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table of event_types**

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBTYPE_TIMEDEVENT_ADD</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td>A timed event has just been added to one of the global event lists (high priority or low priority)</td>
</tr>
<tr>
<td>NEBTYPE_TIMEDEVENT_REMOVE</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td>A timed event has been removed from one of the global event lists</td>
</tr>
<tr>
<td>NEBTYPE_TIMEDEVENT_EXECUTE</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td>A timed event is just about to execute.</td>
</tr>
<tr>
<td>NEBTYPE_TIMEDEVENT_DELAY</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td>Not implemented</td>
</tr>
<tr>
<td>NEBTYPE_TIMEDEVENT_SKIP</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td>Not implemented</td>
</tr>
<tr>
<td>NEBTYPE_TIMEDEVENT_SLEEP</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td>The Nagios scheduler is about to go into a timed sleep due to idleness.</td>
</tr>
</tbody>
</table>

**Relevant Internal Structures**

Nagios keeps two timed event lists: a high priority list and a low priority list. They are defined globally as:

```c
    timed_event *event_list_low;
    timed_event *event_list_high;
```

A timed_event structure is defined as:
typedef struct timed_event_struct{
  int event_type;
  time_t run_time;
  int recurring;
  unsigned long event_interval;
  int compensate_for_time_change;
  void *timing_func;
  void *event_data;
  void *event_args;
  struct timed_event_struct *next;
}timed_event;

Examples

Scheduling an event on one of these two lists is accomplished via the following internal function:

    int schedule_new_event(int event_type, int high_priority, time_t run_time, int recurring, unsigned long event_interval, void *timing_func, int compensate_for_time_change, void *event_data, void *event_args);

Removing a scheduled event is accomplished via the following internal function:

    int deschedule_event(int event_type, int high_priority, void *event_data, void *event_args)
A.9 NENCALLBACK_LOG_DATA

Description

Provides a copy of log entries to call-back routines. Note that this call-back is invoked just after the entry has been written to the Nagios log file.

Data Structure

/* log data structure */
typedef struct nebstruct_log_struct{
  int      type;
  int      flags;
  int      attr;
  struct timeval  timestamp;
  time_t   entry_time;
  int      data_type;
  char     *data;
}nebstruct_log_data;

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry_time</td>
<td>time_t</td>
<td>Time stamp of entry in the log file (in UNIX time)</td>
</tr>
<tr>
<td>data_type</td>
<td>int</td>
<td>Used to classify the source and/or severity of the log entry. See the Log Data Type table below. Notice that the log data types are defined as bit fields. This allows Nagios to filter which types of messages get written to the Nagios log file. Usually, the data_type variable will hold just one of the defined log data types; but it is possible that you may see multiple log data type values bitwise OR-ed together (if there’s an error from the service check result worker thread.)</td>
</tr>
<tr>
<td>data</td>
<td>char *</td>
<td>Message string that was written to the log file</td>
</tr>
</tbody>
</table>

Table of Log Data Types

<table>
<thead>
<tr>
<th>ID</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NSLOG_RUNTIME_ERROR</td>
</tr>
<tr>
<td>2</td>
<td>NSLOG_RUNTIME_WARNING</td>
</tr>
<tr>
<td>4</td>
<td>NSLOG_VERIFICATION_ERROR</td>
</tr>
<tr>
<td>8</td>
<td>NSLOG_VERIFICATION_WARNING</td>
</tr>
<tr>
<td>16</td>
<td>NSLOG_CONFIG_ERROR</td>
</tr>
<tr>
<td>32</td>
<td>NSLOG_CONFIG_WARNING</td>
</tr>
<tr>
<td>64</td>
<td>NSLOG_PROCESS_INFO</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>256</td>
<td>Unused</td>
</tr>
<tr>
<td>512</td>
<td>NSLOG_EXTERNAL_COMMAND</td>
</tr>
<tr>
<td>1024</td>
<td>NSLOG_HOST_UP</td>
</tr>
<tr>
<td>2048</td>
<td>NSLOG_HOST_DOWN</td>
</tr>
<tr>
<td>4096</td>
<td>NSLOG_HOST_UNREACHABLE</td>
</tr>
<tr>
<td>8192</td>
<td>NSLOG_SERVICE_OK</td>
</tr>
<tr>
<td>16384</td>
<td>NSLOG_SERVICE_UNKNOWN</td>
</tr>
<tr>
<td>32768</td>
<td>NSLOG_SERVICE_WARNING</td>
</tr>
<tr>
<td>65536</td>
<td>NSLOG_SERVICE_CRITICAL</td>
</tr>
<tr>
<td>131072</td>
<td>NSLOG_PASSIVE_CHECK</td>
</tr>
<tr>
<td>262144</td>
<td>NSLOG_INFOMESSAGE</td>
</tr>
<tr>
<td>524288</td>
<td>NSLOG_HOST_NOTIFICATION</td>
</tr>
<tr>
<td>1048576</td>
<td>NSLOG_SERVICE_NOTIFICATION</td>
</tr>
</tbody>
</table>

**Invocation**

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBTYPE_LOG_DATA</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_LOG_ROTATION</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
</tbody>
</table>

**Relevant Internal Structures**

<table>
<thead>
<tr>
<th>Global Structure</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern char *log_file</td>
<td>string</td>
<td>Name of the Nagios log file</td>
</tr>
<tr>
<td>extern char *log_archive_path</td>
<td>string</td>
<td>Path to log archive directory</td>
</tr>
<tr>
<td>extern int use_syslog</td>
<td>boolean</td>
<td>Enable/Disables writing log entries to the syslog</td>
</tr>
<tr>
<td>extern int log_service_retries</td>
<td>boolean</td>
<td>Enables/Disable logging soft service states</td>
</tr>
<tr>
<td>extern int log_initial_states</td>
<td>boolean</td>
<td>Enables/Disables logging initial host/service states</td>
</tr>
<tr>
<td>extern unsigned long logging_options</td>
<td>bit-field</td>
<td>A filter which defines which log data types will be written to the Nagios log file</td>
</tr>
<tr>
<td>extern unsigned long syslog_option</td>
<td>bit-field</td>
<td>A filter which defines which log data types will be written to the syslog file</td>
</tr>
<tr>
<td>extern time last_log_rotation</td>
<td>time</td>
<td>Time of last log file rotation</td>
</tr>
<tr>
<td>extern int log_rotation_method</td>
<td>enum</td>
<td>Log rotation method – See the Log Rotation Method table below.</td>
</tr>
<tr>
<td>ID</td>
<td>Mnemonic</td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>---------------------------</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>LOG_ROTATION_NONE</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>LOG_ROTATION_HOURLY</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>LOG_ROTATION_DAILY</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>LOG_ROTATION_WEEKLY</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>LOG_ROTATION_MONTHLY</td>
<td></td>
</tr>
</tbody>
</table>

**Examples**

None.
A.10 NEBCALLBACK_SYSTEM_COMMAND_DATA

Description

Notifies a call-back routine both before and after each system command is executed.

A system command is an external command that is run by Nagios to satisfy one of the following events:

- Executing an obsessive service check command
- Executing an obsessive host check command
- Executing the global service event handler
- Executing a service event handler
- Executing the global host event handler
- Executing a host event handler
- Executing a service contact notification command
- Executing a host contact notification command

Data Structure

```c
/* system command structure */
typedef struct nebstruct_system_command_struct{
    int type;
    int flags;
    int attr;
    struct timeval timestamp;

    struct timeval start_time;
    struct timeval end_time;
    int timeout;
    char *command_line;
    int early_timeout;
    double execution_time;
    int return_code;
    char *output;
} nebstruct_system_command_data;
```

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start_time</td>
<td>timeval</td>
<td>Time that the command started (in UNIX time)</td>
</tr>
<tr>
<td>end_time</td>
<td>timeval</td>
<td>Time that the command ended (in UNIX time). Only valid for event_type NEBTYPE_SYSTEM_COMMAND_END.</td>
</tr>
</tbody>
</table>
timeout | int | Maximum number of seconds to allow for this command to execute.
--- | --- | ---
command_line | char * | The command to be executed.
early_timeout | int | Boolean. Set to TRUE if there was a critical return code and no output AND the command time exceeded the timeout thresholds. Only valid for event_type NEBYTE_SYSTEM_COMMAND_END.
execution_time | double | Elapsed execution in milliseconds. Only valid for event_type NEBYTE_SYSTEM_COMMAND_END.
return_code | int | Return Code: STATE_OK (0), STATE_WARNING (1), STATE_CRITICAL (2) or STATE_UNKNOWN (3). Only valid for event_type NEBYTE_SYSTEM_COMMAND_END.
output | char * | Command output string. Only valid for event_type NEBYTE_SYSTEM_COMMAND_END.

Invocation

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBYTE_SYSTEM_COMMAND_START</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td>Invoked just prior to starting a new process (fork) to execute a system command.</td>
</tr>
<tr>
<td>NEBYTE_SYSTEM_COMMAND_END</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td>Invoked after the systems command has completed and its results collected.</td>
</tr>
</tbody>
</table>

Relevant Internal Structures

None.

Examples

System commands are executed via the internal function:

```c
int my_system(char *cmd, int timeout, int *early_timeout, double *exec_time, char *output, int output_length);
```

Where:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd</td>
<td>char *</td>
<td>[INPUT] Fully-qualified command to execute.</td>
</tr>
<tr>
<td>timeout</td>
<td>int</td>
<td>[INPUT] Maximum number of seconds to allow for this command to execute.</td>
</tr>
<tr>
<td>early_timeout</td>
<td>int *</td>
<td>[OUTPUT] Boolean. Set to TRUE if there was a critical return code and no output AND the command time exceeded the timeout thresholds.</td>
</tr>
<tr>
<td>exec_time</td>
<td>double *</td>
<td>[OUTPUT] Elapsed execution in milliseconds.</td>
</tr>
<tr>
<td>output</td>
<td>char *</td>
<td>[OUTPUT] Command output string. Note: This is expected to be a pre-allocated string buffer.</td>
</tr>
<tr>
<td>output_length</td>
<td>int</td>
<td>[INPUT] Pre-allocated size of the output buffer in bytes.</td>
</tr>
</tbody>
</table>
The return value is the result code from the executed command and is one of: STATE_OK (0), STATE_WARNING (1), STATE_CRITICAL (2) or STATEUNKNOWN (3).
A.11 NEBCALLBACK_EVENT_HANDLER_DATA

Description

Notifies a call-back routine before and after an event handler is executed.

Nagios invokes this call-back for the following four event handlers:

- Global Service Event Handler
- Service Event Handler
- Global Host Event Handler
- Host Event Handler

Data Structure

    /* event handler structure */
    typedef struct nebstruct_event_handler_struct{
        int          type;
        int          flags;
        int          attr;
        struct timeval timestamp;

        int          eventhandler_type;
        char        *host_name;
        char        *service_description;
        int          state_type;
        int          state;
        int          timeout;
        char        *command_name;
        char        *command_args;
        char        *command_line;
        struct timeval start_time;
        struct timeval end_time;
        int          early_timeout;
        double       execution_time;
        int          return_code;
        char        *output;
    }nebstruct_event_handler_data;
<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eventhandler_type</td>
<td>int</td>
<td>Identifies which of the four types of event handler. See the Event Handler Type table below.</td>
</tr>
<tr>
<td>host_name</td>
<td>char *</td>
<td>Host name</td>
</tr>
<tr>
<td>service_description</td>
<td>char *</td>
<td>Service description</td>
</tr>
<tr>
<td>state_type</td>
<td>int</td>
<td>Host/Service State Type: SOFT_STATE (0) or HARD_STATE (1)</td>
</tr>
<tr>
<td>state</td>
<td>int</td>
<td>Host/Service State. See the Host/Service State table below.</td>
</tr>
<tr>
<td>timeout</td>
<td>int</td>
<td>Maximum number of seconds to allow for this command to execute.</td>
</tr>
<tr>
<td>command_name</td>
<td>char *</td>
<td>The command’s name.</td>
</tr>
<tr>
<td>command_args</td>
<td>char *</td>
<td>The command’s arguments (separated by exclamation points)</td>
</tr>
<tr>
<td>command_line</td>
<td>char *</td>
<td>The full, processed command line (i.e., after parameter substitution) Only valid for event_type NEBTYPE_EVENTHANDLER_END.</td>
</tr>
<tr>
<td>start_time</td>
<td>timeval</td>
<td>Time that the command started (in UNIX time)</td>
</tr>
<tr>
<td>end_time</td>
<td>timeval</td>
<td>Time that the command ended (in UNIX time). Only valid for event_type NEBTYPE_EVENTHANDLER_END.</td>
</tr>
<tr>
<td>early_timeout</td>
<td>int</td>
<td>Boolean. Set to TRUE if there was a critical return code and no output AND the command time exceeded the timeout thresholds. Only valid for event_type NEBTYPE_EVENTHANDLER_END.</td>
</tr>
<tr>
<td>execution_time</td>
<td>double</td>
<td>Elapsed execution in milliseconds. Only valid for event_type NEBTYPE_EVENTHANDLER_END.</td>
</tr>
<tr>
<td>return_code</td>
<td>int</td>
<td>Return Code: STATE_OK (0), STATE_WARNING (1), STATE_CRITICAL (2) or STATE_UNKNOWN (3). Only valid for event_type NEBTYPE_EVENTHANDLER_END.</td>
</tr>
<tr>
<td>output</td>
<td>char *</td>
<td>Command output string. Only valid for event_type NEBTYPE_EVENTHANDLER_END.</td>
</tr>
</tbody>
</table>

### Event Handler Type Table

<table>
<thead>
<tr>
<th>ID</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>HOST_EVENTHANDLER</td>
</tr>
<tr>
<td>1</td>
<td>SERVICE_EVENTHANDLER</td>
</tr>
<tr>
<td>2</td>
<td>GLOBAL_HOST_EVENTHANDLER</td>
</tr>
<tr>
<td>3</td>
<td>GLOBAL_SERVICE_EVENTHANDLER</td>
</tr>
</tbody>
</table>

### Host State Table

<table>
<thead>
<tr>
<th>ID</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>HOST_UP</td>
</tr>
<tr>
<td>1</td>
<td>HOST_DOWN</td>
</tr>
<tr>
<td>2</td>
<td>HOST_UNREACHABLE</td>
</tr>
</tbody>
</table>

### Service State Table

<table>
<thead>
<tr>
<th>ID</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STATE_OK</td>
</tr>
<tr>
<td>1</td>
<td>STATE_WARNING</td>
</tr>
<tr>
<td>2</td>
<td>STATE_CRITICAL</td>
</tr>
</tbody>
</table>


**Invocation**

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBTYPE_EVENTHANDLER_START</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td>An event handler is about to be executed.</td>
</tr>
<tr>
<td>NEBTYPE_EVENTHANDLER_END</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td>An event handler has completed execution</td>
</tr>
</tbody>
</table>

**Relevant Internal Structures**

<table>
<thead>
<tr>
<th>Global Structure</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern int enable_event_handlers</td>
<td>boolean</td>
<td>Enable/Disable event handlers</td>
</tr>
<tr>
<td>extern int log_event_handlers</td>
<td>boolean</td>
<td>Enable/Disable logging of event handler events</td>
</tr>
<tr>
<td>extern int event_handler_timeout</td>
<td>value</td>
<td>Maximum number of seconds to allow for event handlers to execute.</td>
</tr>
<tr>
<td>extern char *global_host_event_handler</td>
<td>string</td>
<td>Global host event handler command string.</td>
</tr>
<tr>
<td>extern char *global_service_event_handler</td>
<td>string</td>
<td>Global service event handler command string.</td>
</tr>
</tbody>
</table>

**Examples**

None.
A.12 NEBCALLBACK_NOTIFICATION_DATA

Description

Data Structure

```c
/* notification data structure */
typedef struct nebstruct_notification_struct{
    int type;
    int flags;
    int attr;
    struct timeval timestamp;
    int notification_type;
    struct timeval start_time;
    struct timeval end_time;
    char *host_name;
    char *service_description;
    int reason_type;
    int state;
    char *output;
    char *ack_author;
    char *ack_data;
    int escalated;
    int contacts_notified;
} nebstruct_notification_data;
```

Invocation

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBTYPE_NOTIFICATION_START</td>
<td>NEBFLAG NONE</td>
<td>NEBATTR NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_NOTIFICATION_END</td>
<td>NEBFLAG NONE</td>
<td>NEBATTR NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_CONTACTNOTIFICATION_START</td>
<td>NEBFLAG NONE</td>
<td>NEBATTR NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_CONTACTNOTIFICATION_END</td>
<td>NEBFLAG NONE</td>
<td>NEBATTR NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_CONTACTNOTIFICATIONMETHOD_START</td>
<td>NEBFLAG NONE</td>
<td>NEBATTR NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_CONTACTNOTIFICATIONMETHOD_END</td>
<td>NEBFLAG NONE</td>
<td>NEBATTR NONE</td>
<td></td>
</tr>
</tbody>
</table>

Relevant Internal Structures
A.13 NEBCALLBACK_SERVICE_CHECK_DATA

Description

Data Structure

/* service check structure */
typedef struct nebstruct_service_check_struct{
    int type;
    int flags;
    int attr;
    struct timeval timestamp;

    char *host_name;
    char *service_description;
    int check_type;
    int current_attempt;
    int max_attempts;
    int state_type;
    int state;
    int timeout;
    char *command_name;
    char *command_args;
    char *command_line;
    struct timeval start_time;
    struct timeval end_time;
    int early_timeout;
    double execution_time;
    double latency;
    int return_code;
    char *output;
    char *perf_data;
}nebstruct_service_check_data;

Invocation

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBTYPE_SERVICECHECK_INITIATE</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_SERVICECHECK_PROCESSED</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_SERVICECHECK_RAW_START</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td>Not implemented.</td>
</tr>
</tbody>
</table>
Relevant Internal Structures

Examples
A.14 NEBCALLBACK_HOST_CHECK_DATA

Description

Data Structure

```c
/* host check structure */
typedef struct nebstruct_host_check_struct{
    int       type;
    int       flags;
    int       attr;
    struct timeval timestamp;
    char      *host_name;
    int       current_attempt;
    int       check_type;
    int       max_attempts;
    int       state_type;
    int       state;
    int       timeout;
    char      *command_name;
    char      *command_args;
    char      *command_line;
    struct timeval start_time;
    struct timeval end_time;
    int       early_timeout;
    double    execution_time;
    double    latency;
    int       return_code;
    char      *output;
    char      *perf_data;
}nebstruct_host_check_data;
```

Invocation

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBTYPE_HOSTCHECK_INITIATE</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td>A check of the route to the host has been initiated</td>
</tr>
<tr>
<td>NEBTYPE_HOSTCHECK_PROCESSED</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td>The processed/final result of a host check</td>
</tr>
<tr>
<td>NEBTYPE_HOSTCHECK_RAW_START</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td>The start of a &quot;raw&quot; host check</td>
</tr>
<tr>
<td>NEBTYPE_HOSTCHECK_RAW_END</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td>A finished &quot;raw&quot; host check.</td>
</tr>
</tbody>
</table>
Relevant Internal Structures

Examples
A.15 NEBCALLBACK_CONFIG_DATA

Description

Data Structure

```c
/* comment data structure */
typedef struct nebstruct_comment_struct{
    int type;
    int flags;
    int attr;
    struct timeval timestamp;

    int comment_type;
    char *host_name;
    char *service_description;
    time_t entry_time;
    char *author_name;
    char *comment_data;
    int persistent;
    int source;
    int entry_type;
    int expires;
    time_t expire_time;
    unsigned long comment_id;
}nebstruct_comment_data;
```

Invocation

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBTYPE_COMMENT_ADD</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_COMMENT_DELETE</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_COMMENT_LOAD</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
</tbody>
</table>

Relevant Internal Structures

Examples
A.16 NEBCALLBACK_DOWNTIME_DATA

*Description*

*Data Structure*

    /* downtime data structure */
    typedef struct nebstruct_downtime_struct{
        int      type;
        int      flags;
        int      attr;
        struct timeval timestamp;
        ...
    }nebstruct_downtime_data;

*Invocation*

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBTYPE_DOWNTIME_ADD</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_DOWNTIME_DELETE</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_DOWNTIME_LOAD</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_DOWNTIME_START</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_DOWNTIME_STOP</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
</tbody>
</table>

*Relevant Internal Structures*

*Examples*
A.17 NEBCALLBACK_FLAPPING_DATA

Description

Data Structure

/* flapping data structure */
typedef struct nebstruct_flapping_struct{
    int type;
    int flags;
    int attr;
    struct timeval timestamp;

    int flapping_type;
    char *host_name;
    char *service_description;
    double percent_change;
    double high_threshold;
    double low_threshold;
    unsigned long comment_id;
} nebstruct_flapping_data;

Invocation

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBTYPE_FLAPPING_START</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_FLAPPING_STOP</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
</tbody>
</table>

Relevant Internal Structures

Examples
A.18 NEBCALLBACK_PROGRAM_STATUS_DATA

**Description**

**Data Structure**

```c
/* program status structure */
typedef struct nebstruct_program_status_struct{
    int    type;
    int    flags;
    int    attr;
    struct timeval timestamp;
    time_t program_start;
    int    pid;
    int    daemon_mode;
    time_t last_command_check;
    time_t last_log_rotation;
    int    notifications_enabled;
    int    active_service_checks_enabled;
    int    passive_service_checks_enabled;
    int    active_host_checks_enabled;
    int    passive_host_checks_enabled;
    int    event_handlers_enabled;
    int    flap_detection_enabled;
    int    failure_prediction_enabled;
    int    process_performance_data;
    int    obsess_over_hosts;
    int    obsess_over_services;
    unsigned long modified_host_attributes;
    unsigned long modified_service_attributes;
    char    *global_host_event_handler;
    char    *global_service_event_handler;
}nebstruct_program_status_data;
```

**Invocation**

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBTYPE_PROGRAMSTATUS_UPDATE</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
</tbody>
</table>
Relevant Internal Structures

Examples
A.19  NEBCALLBACK_HOST_STATUS_DATA

Description

Data Structure

    /* host status structure */
    typedef struct nebstruct_host_status_struct{
        int type;
        int flags;
        int attr;
        struct timeval timestamp;
        void *object_ptr;
    }nebstruct_host_status_data;

Invocation

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBTYPE_HOSTSTATUS_UPDATE</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
</tbody>
</table>

Relevant Internal Structures

Examples
A.20  NEBCALLBACK_SERVICE_STATUS_DATA

Description

Data Structure

    
    typedef struct
    
        struct_service_status_struct{
            int type;
            int flags;
            int attr;
            struct timeval timestamp;

            void *object_ptr;
        } struct_service_status_data;

Invocation

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBTYPE_SERVICESTATUS_UPDATE</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
</tbody>
</table>

Relevant Internal Structures

Examples
A.21  NEBCALLBACK_ADAPTIVE_PROGRAM_DATA

Description

Data Structure

/* adaptive program data structure */
typedef struct nebstruct_adaptive_program_data_struct{
  int     type;
  int     flags;
  int     attr;
  struct timeval timestamp;

  int     command_type;
  unsigned long modified_host_attribute;
  unsigned long modified_host_attributes;
  unsigned long modified_service_attribute;
  unsigned long modified_service_attributes;
  char    *global_host_event_handler;
  char    *global_service_event_handler;
}nebstruct_adaptive_program_data;

Invocation

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBTYPE_ADAPTIVEPROGRAM_UPDATE</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
</tbody>
</table>

Relevant Internal Structures

Examples
A.22 NEBCALLBACK_ADAPTIVE_HOST_DATA

Description

Data Structure

/* adaptive host data structure */
typedef struct nebstruct_adaptive_host_data_struct{
    int type;
    int flags;
    int attr;
    struct timeval timestamp;

    int command_type;
    unsigned long modified_attribute;
    unsigned long modified_attributes;
    void *object_ptr;
}nebstruct_adaptive_host_data;

Invocation

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBTYPE_ADAPTIVEHOST_UPDATE</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
</tbody>
</table>

Relevant Internal Structures

Examples
A.23  NEBCALLBACK_ADAPTIVE_SERVICE_DATA

Description

Data Structure

/* adaptive service data structure */
typedef struct nebstruct_adaptive_service_data_struct{
    int type;
    int flags;
    int attr;
    struct timeval timestamp;
    int command_type;
    unsigned long modified_attribute;
    unsigned long modified_attributes;
    void *object_ptr;
} nebstruct_adaptive_service_data;

Invocation

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBTYPE_ADAPTIVESERVICE_UPDATE</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
</tbody>
</table>

Relevant Internal Structures

Examples
A.24  NEBCALLBACK_EXTERNAL_COMMAND_DATA

Description

Data Structure

/* external command data structure */
typedef struct nebstruct_external_command_struct{
  int    type;
  int    flags;
  int    attr;
  struct timeval timestamp;
  int    command_type;
  time_t entry_time;
  char   *command_string;
  char   *command_args;
}nebstruct_external_command_data;

Invocation

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBTYPE_EXTERNALCOMMAND_START</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_EXTERNALCOMMAND_END</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
</tbody>
</table>

Relevant Internal Structures

Examples
A.25 NEECALLBACK_AGGREGATED_STATUS_DATA

Description

Data Structure

    /* aggregated status data structure */
    typedef struct nebstruct_aggregated_status_struct{
        int        type;
        int        flags;
        int        attr;
        struct timeval  timestamp;
    }nebstruct_aggregated_status_data;

Invocation

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBTYPE_AGGREGATEDSTATUS_STARTDUMP</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_AGGREGATEDSTATUS_ENDDUMP</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
</tbody>
</table>

Relevant Internal Structures

Examples
A.26  NEBCALLBACK_RETENTION_DATA

Description

Data Structure

/* retention data structure */
typedef struct nebstruct_retention_struct{
  int type;
  int flags;
  int attr;
  struct timeval timestamp;
} nebstruct_retention_data;

Invocation

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBTYPE_RETENTIONDATA_STARTLOAD</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_RETENTIONDATA_ENDLOAD</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_RETENTIONDATA_STARTSAVE</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_RETENTIONDATA_ENDSAVE</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
</tbody>
</table>

Relevant Internal Structures

Examples
A.27  NEBCALLBACK_CONTACT_NOTIFICATION_DATA

Description

Data Structure

/* contact notification data structure */
typedef struct nebstruct_contact_notification_struct{
    int    type;
    int    flags;
    int    attr;
    struct timeval timestamp;

    int    notification_type;
    struct timeval start_time;
    struct timeval end_time;
    char   *host_name;
    char   *service_description;
    char   *contact_name;
    int    reason_type;
    int    state;
    char   *output;
    char   *ack_author;
    char   *ack_data;
    int    escalated;
}nebstruct_contact_notification_data;

Invocation

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBTYPE_NOTIFICATION_START</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_NOTIFICATION_END</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_CONTACTNOTIFICATION_START</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_CONTACTNOTIFICATION_END</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
</tbody>
</table>

Relevant Internal Structures

Examples
A.28  NEBCALLBACK_CONTACT_NOTIFICATION_METHOD_DATA

Description

Data Structure

/* contact notification method data structure */
typedef struct nebstruct_contact_notification_method_struct{
  int type;
  int flags;
  int attr;
  struct timeval timestamp;

  int notification_type;
  struct timeval start_time;
  struct timeval end_time;
  char *host_name;
  char *service_description;
  char *command_name;
  char *command_args;
  int reason_type;
  int state;
  char *output;
  char *ack_author;
  char *ack_data;
  int escalated;
} nebstruct_contact_notification_method_data;

Invocation

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBTYPE_CONTACT_NOTIFICATIONMETHOD_START</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_CONTACT_NOTIFICATIONMETHOD_END</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
</tbody>
</table>

Relevant Internal Structures

Examples
A.29 NEBCALLBACK_ACKNOWLEDGEMENT_DATA

Description

Data Structure

    /* acknowledgement structure */
    typedef struct nebstruct_acknowledgement_struct{
        int type;
        int flags;
        int attr;
        struct timeval timestamp;

        int acknowledgement_type;
        char *host_name;
        char *service_description;
        int state;
        char *author_name;
        char *comment_data;
        int is_sticky;
        int persistent_comment;
        int notify_contacts;
    }nebstruct_acknowledgement_data;

Invocation

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBYTE_ACKNOWLEDGEMENT_ADD</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
<tr>
<td>NEBYTE_ACKNOWLEDGEMENT_REMOVE</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
<tr>
<td>NEBYTE_ACKNOWLEDGEMENT_LOAD</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
</tbody>
</table>

Relevant Internal Structures

Examples
A.30 NEBCALLBACK_STATE_CHANGE_DATA

Description

Data Structure

```c
typedef struct nebstruct_statechange_struct{
  int     type;
  int     flags;
  int     attr;
  struct timeval timestamp;
  int     statechange_type;
  char    *host_name;
  char    *service_description;
  int     state;
  int     state_type;
  int     current_attempt;
  int     max_attempts;
  char    *output;
} nebstruct_statechange_data;
```

Invocation

<table>
<thead>
<tr>
<th>Event Types</th>
<th>Flags</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEBTYPE_STATECHANGE_START</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
<tr>
<td>NEBTYPE_STATECHANGE_END</td>
<td>NEBFLAG_NONE</td>
<td>NEBATTR_NONE</td>
<td></td>
</tr>
</tbody>
</table>

Relevant Internal Structures

Examples
Appendix B: Catalog of Global Nagios Data Structures

B.1 TBA