NAME

engine - ENGINE cryptographic module support

SYNOPSIS

```
#include <openssl/engine.h>

ENGINE *ENGINE_get_first(void);
ENGINE *ENGINE_get_last(void);
ENGINE *ENGINE_get_next(ENGINE *e);
ENGINE *ENGINE_get_prev(ENGINE *e);

int ENGINE_add(ENGINE *e);
int ENGINE_remove(ENGINE *e);

ENGINE *ENGINE_by_id(const char *id);

int ENGINE_init(ENGINE *e);
int ENGINE_finish(ENGINE *e);

void ENGINE_load_openssl(void);
void ENGINE_load_dynamic(void);
#ifndef OPENSSL_NO_STATIC_ENGINE
void ENGINE_load_4758cca(void);
void ENGINE_load_aep(void);
void ENGINE_load_atalla(void);
void ENGINE_load_chil(void);
void ENGINE_load_cswift(void);
void ENGINE_load_gmp(void);
void ENGINE_load_nuron(void);
void ENGINE_load_sureware(void);
void ENGINE_load_ubsec(void);
#endif
void ENGINE_load_cryptodev(void);
void ENGINE_load_builtin_engines(void);

void ENGINE_cleanup(void);

ENGINE *ENGINE_get_default_RSA(void);
ENGINE *ENGINE_get_default_DSA(void);
ENGINE *ENGINE_get_default_ECDH(void);
ENGINE *ENGINE_get_default_ECDSA(void);
ENGINE *ENGINE_get_default_DH(void);
ENGINE *ENGINE_get_default_RAND(void);
ENGINE *ENGINE_get_cipher_engine(int nid);
ENGINE *ENGINE_get_digest_engine(int nid);

int ENGINE_set_default_RSA(ENGINE *e);
int ENGINE_set_default_DSA(ENGINE *e);
int ENGINE_set_default_ECDH(ENGINE *e);
int ENGINE_set_default_ECDSA(ENGINE *e);
int ENGINE_set_default_DH(ENGINE *e);
int ENGINE_set_default_RAND(ENGINE *e);
int ENGINE_set_default_ciphers(ENGINE *e);
int ENGINE_set_default_digests(ENGINE *e);
int ENGINE_set_default_string(ENGINE *e, const char *list);
int ENGINE_set_default(ENGINE *e, unsigned int flags);

unsigned int ENGINE_get_table_flags(void);
void ENGINE_set_table_flags(unsigned int flags);

int ENGINE_register_RSA(ENGINE *e);
void ENGINE_unregister_RSA(ENGINE *e);
void ENGINE_register_all_RSA(void);
int ENGINE_register_DSA(ENGINE *e);
void ENGINE_unregister_DSA(ENGINE *e);
void ENGINE_register_all_DSA(void);
int ENGINE_register_ECDH(ENGINE *e);
void ENGINE_unregister_ECDH(ENGINE *e);
void ENGINE_register_all_ECDH(void);
int ENGINE_register_ECDSA(ENGINE *e);
void ENGINE_unregister_ECDSA(ENGINE *e);
void ENGINE_register_all_ECDSA(void);
int ENGINE_register_DH(ENGINE *e);
void ENGINE_unregister_DH(ENGINE *e);
void ENGINE_register_all_DH(void);
int ENGINE_register_RAND(ENGINE *e);
void ENGINE_unregister_RAND(ENGINE *e);
void ENGINE_register_all_RAND(void);
int ENGINE_register_STORE(ENGINE *e);
void ENGINE_unregister_STORE(ENGINE *e);
void ENGINE_register_all_STORE(void);
int ENGINE_register_ciphers(ENGINE *e);
void ENGINE_unregister_ciphers(ENGINE *e);
void ENGINE_register_all_ciphers(void);
int ENGINE_register_digests(ENGINE *e);
void ENGINE_unregister_digests(ENGINE *e);
void ENGINE_register_all_digests(void);
int ENGINE_register_complete(ENGINE *e);
int ENGINE_register_all_complete(void);

int ENGINE_ctrl(ENGINE *e, int cmd, long i, void *p, void (*f)(void));
int ENGINE_cmd_is_executable(ENGINE *e, int cmd);
int ENGINE_ctrl_cmd(ENGINE *e, const char *cmd_name,
long i, void *p, void (*f)(void), int cmd_optional);
int ENGINE_ctrl_cmd_string(ENGINE *e, const char *cmd_name, const char *arg,
int cmd_optional);

int ENGINE_set_ex_data(ENGINE *e, int idx, void *arg);
void *ENGINE_get_ex_data(const ENGINE *e, int idx);

int ENGINE_get_ex_new_index(long argl, void *argp, CRYPTO_EX_new *new_func,
CRYPTO_EX_dup *dup_func, CRYPTO_EX_free *free_func);

ENGINE *ENGINE_new(void);
int ENGINE_free(ENGINE *e);
int ENGINE_up_ref(ENGINE *e);

int ENGINE_set_id(ENGINE *e, const char *id);
int ENGINE_set_name(ENGINE *e, const char *name);
int ENGINE_set_RSA(ENGINE *e, const RSA_METHOD *rsa_meth);
int ENGINE_set_DSA(ENGINE *e, const DSA_METHOD *dsa_meth);
int ENGINE_set_ECDH(ENGINE *e, const ECDH_METHOD *dh_meth);
int ENGINE_set_ECDSA(ENGINE *e, const ECDSA_METHOD *dh_meth);
int ENGINE_set_DH(ENGINE *e, const DH_METHOD *dh_meth);
int ENGINE_set_RAND(ENGINE *e, const RAND_METHOD *rand_meth);
int ENGINE_set_STORE(ENGINE *e, const STORE_METHOD *rand_meth);
int ENGINE_set_destroy_function(ENGINE *e, ENGINE_GEN_INT_FUNC_PTR destroy_f);
int ENGINE_set_init_function(ENGINE *e, ENGINE_GEN_INT_FUNC_PTR init_f);
int ENGINE_set_finish_function(ENGINE *e, ENGINE_GEN_INT_FUNC_PTR finish_f);
int ENGINE_set_ctrl_function(ENGINE *e, ENGINE_CTRL_FUNC_PTR ctrl_f);
int ENGINE_set_load_privkey_function(ENGINE *e, ENGINE_LOAD_KEY_PTR loadpriv_f);
int ENGINE_set_load_pubkey_function(ENGINE *e, ENGINE_LOAD_KEY_PTR loadpub_f);
int ENGINE_set_ciphers(ENGINE *e, ENGINE_CIPHERS_PTR f);
int ENGINE_set_digests(ENGINE *e, ENGINE_DIGESTS_PTR f);
int ENGINE_set_flags(ENGINE *e, int flags);
int ENGINE_set_cmd_defns(ENGINE *e, const ENGINE_CMD_DEFN *defns);

const char *ENGINE_get_id(const ENGINE *e);
const char *ENGINE_get_name(const ENGINE *e);
const RSA_METHOD *ENGINE_get_RSA(const ENGINE *e);
const DSA_METHOD *ENGINE_get_DSA(const ENGINE *e);
const ECDH_METHOD *ENGINE_get_ECDH(const ENGINE *e);
const ECDSA_METHOD *ENGINE_get_ECDSA(const ENGINE *e);
const DH_METHOD *ENGINE_get_DH(const ENGINE *e);
const RAND_METHOD *ENGINE_get_RAND(const ENGINE *e);
const STORE_METHOD *ENGINE_get_STORE(const ENGINE *e);
ENGINE_GEN_INT_FUNC_PTR ENGINE_get_destroy_function(const ENGINE *e);
ENGINE_GEN_INT_FUNC_PTR ENGINE_get_init_function(const ENGINE *e);
ENGINE_GEN_INT_FUNC_PTR ENGINE_get_finish_function(const ENGINE *e);
ENGINE_CTRL_FUNC_PTR ENGINE_get_ctrl_function(const ENGINE *e);
ENGINE_LOAD_KEY_PTR ENGINE_get_load_privkey_function(const ENGINE *e);
ENGINE_LOAD_KEY_PTR ENGINE_get_load_pubkey_function(const ENGINE *e);
ENGINE_CIPHERS_PTR ENGINE_get_ciphers(const ENGINE *e);
ENGINE_DIGESTS_PTR ENGINE_get_digests(const ENGINE *e);
const EVP_CIPHER *ENGINE_get_cipher(ENGINE *e, int nid);
const EVP_MD *ENGINE_get_digest(ENGINE *e, int nid);
int ENGINE_get_flags(const ENGINE *e);
const ENGINE_CMD_DEFN *ENGINE_get_cmd_defns(const ENGINE *e);

EVP_PKEY *ENGINE_load_private_key(ENGINE *e, const char *key_id,
UI_METHOD *ui_method, void *callback_data);
EVP_PKEY *ENGINE_load_public_key(ENGINE *e, const char *key_id,
UI_METHOD *ui_method, void *callback_data);

void ENGINE_add_conf_module(void);
```

DESCRIPTION

These functions create, manipulate, and use cryptographic modules in the form of **ENGINE** objects. These objects act as containers for implementations of cryptographic algorithms, and support a reference-counted mechanism to allow them to be dynamically loaded in and out of the running application.

The cryptographic functionality that can be provided by an **ENGINE** implementation includes the following abstractions;

```
RSA_METHOD - for providing alternative RSA implementations
DSA_METHOD, DH_METHOD, RAND_METHOD, ECDH_METHOD, ECDSA_METHOD,
STORE_METHOD - similarly for other OpenSSL APIs
EVP_CIPHER - potentially multiple cipher algorithms (indexed by 'nid')
EVP_DIGEST - potentially multiple hash algorithms (indexed by 'nid')
key-loading - loading public and/or private EVP_PKEY keys
```

Reference counting and handles

Due to the modular nature of the ENGINE API, pointers to ENGINEs need to be treated as handles - ie. not only as pointers, but also as references to the underlying ENGINE object. Ie. one should obtain a new reference when making copies of an ENGINE pointer if the copies will be used (and released) independently.

ENGINE objects have two levels of reference-counting to match the way in which the objects are used. At the most basic level, each ENGINE pointer is inherently a **structural** reference - a structural reference is required to use the pointer value at all, as this kind of reference is a guarantee that the structure can not be deallocated until the reference is released.

However, a structural reference provides no guarantee that the ENGINE is initiliased and able to use any of its cryptographic implementations. Indeed it's quite possible that most ENGINEs will not initialise at all in typical environments, as ENGINEs are typically used to support specialised hardware. To use an ENGINE's functionality, you need a **functional** reference. This kind of reference can be considered a specialised form of structural reference, because each functional reference implicitly contains a structural reference as well - however to avoid difficult-to-find programming bugs, it is recommended to treat the two kinds of reference independently. If you have a functional reference to an ENGINE, you have a guarantee that the ENGINE has been initialised ready to perform cryptographic operations and will remain uninitialised until after you have released your reference.

Structural references

This basic type of reference is used for instantiating new ENGINEs, iterating across OpenSSL's internal linked-list of loaded ENGINEs, reading information about an ENGINE, etc. Essentially a structural reference is sufficient if you only need to query or manipulate the data of an ENGINE implementation rather than use its functionality.

The *ENGINE_new()* function returns a structural reference to a new (empty) ENGINE object. There are other ENGINE API functions that return structural references such as; *ENGINE_by_id()*, *ENGINE_get_first()*, *ENGINE_get_last()*, *ENGINE_get_next()*, *ENGINE_get_prev()*. All structural references should be released by a corresponding to call to the *ENGINE_free()* function - the ENGINE object itself will only actually be cleaned up and deallocated when the last structural reference is released.

It should also be noted that many ENGINE API function calls that accept a structural reference will internally obtain another reference - typically this happens whenever the supplied ENGINE will be needed by OpenSSL after the function has returned. Eg. the function to add a new ENGINE to OpenSSL's internal list is *ENGINE_add()* - if this function returns success, then OpenSSL will have stored a new structural reference internally so the caller is still responsible for freeing their own reference with *ENGINE_free()* when they are finished with it. In a similar way, some functions will automatically release the structural reference passed to it if part of the function's job is to do so. Eg. the *ENGINE_get_next()* and *ENGINE_get_prev()* functions are used for iterating across the internal ENGINE list - they will return a new structural reference to the next (or previous) ENGINE in the list or NULL if at the end (or beginning) of the list, but in either case the structural reference passed to the function is released on behalf of the caller.

To clarify a particular function's handling of references, one should always consult that function's documentation "man" page, or failing that the openssl/engine.h header file includes some hints.

Functional references

As mentioned, functional references exist when the cryptographic functionality of an ENGINE is required to be available. A functional reference can be obtained in one of two ways; from an existing structural reference to the required ENGINE, or by asking OpenSSL for the default operational ENGINE for a given cryptographic purpose.

To obtain a functional reference from an existing structural reference, call the *ENGINE_init()* function. This returns zero if the ENGINE was not already operational and couldn't be successfully initialised (eg. lack of system drivers, no special hardware attached, etc), otherwise it will return non-zero to indicate that the ENGINE is now operational and will have allocated a new **functional** reference to the ENGINE. All functional references are released by calling *ENGINE_finish()* (which removes the implicit structural reference as well).

The second way to get a functional reference is by asking OpenSSL for a default implementation for a given task, eg. by *ENGINE_get_default_RSA()*, *ENGINE_get_default_cipher_engine()*, etc. These are discussed in the next section, though they are not usually required by application programmers as they are used automatically when creating and using the relevant algorithm-specific types in OpenSSL, such as RSA, DSA, EVP_CIPHER_CTX, etc.

Default implementations

For each supported abstraction, the ENGINE code maintains an internal table of state to control which implementations are available for a given abstraction and which should be used by default. These implementations are registered in the tables and indexed by an 'nid' value, because abstractions like EVP_CIPHER and EVP_DIGEST support many distinct algorithms and modes, and ENGINEs can support arbitrarily many of them. In the case of other abstractions like RSA, DSA, etc, there is only one ''algorithm'' so all implementations implicitly register using the same 'nid' index.

When a default ENGINE is requested for a given abstraction/algorithm/mode, (eg. when calling RSA_new_method(NULL)), a ''get_default'' call will be made to the ENGINE subsystem to process the corresponding state table and return a functional reference to an initialised ENGINE whose implementation should be used. If no ENGINE should (or can) be used, it will return NULL and the caller will operate with a NULL ENGINE handle - this usually equates to using the conventional software implementation. In the latter case, OpenSSL will from then on behave the way it used to before the ENGINE API existed.

Each state table has a flag to note whether it has processed this ''get_default'' query since the table was last modified, because to process this question it must iterate across all the registered ENGINEs in the table trying to initialise each of them in turn, in case one of them is operational. If it returns a functional reference to an ENGINE, it will also cache another reference to speed up processing future queries (without needing to iterate across the table). Likewise, it will cache a NULL response if no ENGINE was available so that future queries won't repeat the same iteration unless the state table changes. This behaviour can also be changed; if the ENGINE_TABLE_FLAG_NOINIT flag is set (using *ENGINE_set_table_flags()*), no attempted initialisations will take place, instead the only way for the state table to return a non-NULL ENGINE to the ''get_default'' query will be if one is expressly set in the table. Eg. *ENGINE_set_default_RSA()* does the same job as *ENGINE_register_RSA()* except that it also sets the state table's cached response for the ''get_default'' query. In the case of abstractions like EVP_CIPHER, where implementations are indexed by 'nid', these flags and cached-responses are distinct for each 'nid' value.

Application requirements

This section will explain the basic things an application programmer should support to make the most useful elements of the ENGINE functionality available to the user. The first thing to consider is whether the programmer wishes to make alternative ENGINE modules available to the application and user. OpenSSL maintains an internal linked list of ''visible'' ENGINEs from which it has to operate - at start-up, this list is empty and in fact if an application does not call any ENGINE API calls and it uses static linking against openssl, then the resulting application binary will not contain any alternative ENGINE code at all. So the first consideration is whether any/all available ENGINE implementations should be made visible to OpenSSL - this is controlled by calling the various ‘‘load’’ functions, eg.

```
/* Make the "dynamic" ENGINE available */
void ENGINE_load_dynamic(void);
/* Make the CryptoSwift hardware acceleration support available */
void ENGINE_load_cswift(void);
/* Make support for nCipher's "CHIL" hardware available */
void ENGINE_load_chil(void);
...
/* Make ALL ENGINE implementations bundled with OpenSSL available */
void ENGINE_load_builtin_engines(void);
```

Having called any of these functions, ENGINE objects would have been dynamically allocated and populated with these implementations and linked into OpenSSL's internal linked list. At this point it is important to mention an important API function;

```
void ENGINE_cleanup(void);
```

If no ENGINE API functions are called at all in an application, then there are no inherent memory leaks to worry about from the ENGINE functionality, however if any ENGINEs are loaded, even if they are never registered or used, it is necessary to use the *ENGINE_cleanup()* function to correspondingly cleanup before program exit, if the caller wishes to avoid memory leaks. This mechanism uses an internal callback registration table so that any ENGINE API functionality that knows it requires cleanup can register its cleanup details to be called during *ENGINE_cleanup()*. This approach allows *ENGINE_cleanup()* to clean up after any ENGINE functionality at all that your program uses, yet doesn't automatically create linker dependencies to all possible ENGINE functionality - only the cleanup callbacks required by the functionality you do use will be required by the linker.

The fact that ENGINEs are made visible to OpenSSL (and thus are linked into the program and loaded into memory at run-time) does not mean they are ‘‘registered’’ or called into use by OpenSSL automatically - that behaviour is something for the application to control. Some applications will want to allow the user to specify exactly which ENGINE they want used if any is to be used at all. Others may prefer to load all support and have OpenSSL automatically use at run-time any ENGINE that is able to successfully initialise - ie. to assume that this corresponds to acceleration hardware attached to the machine or some such thing. There are probably numerous other ways in which applications may prefer to handle things, so we will simply illustrate the consequences as they apply to a couple of simple cases and leave developers to consider these and the source code to openssl's builtin utilities as guides.

Using a specific ENGINE implementation

Here we'll assume an application has been configured by its user or admin to want to use the ‘‘ACME’’ ENGINE if it is available in the version of OpenSSL the application was compiled with. If it is available, it should be used by default for all RSA, DSA, and symmetric cipher operation, otherwise OpenSSL should use its builtin software as per usual. The following code illustrates how to approach this;

```
ENGINE *e;
const char *engine_id = "ACME";
ENGINE_load_builtin_engines();
e = ENGINE_by_id(engine_id);
if(!e)
/* the engine isn't available */
return;
if(!ENGINE_init(e)) {
/* the engine couldn't initialise, release 'e' */
ENGINE_free(e);
return;
}
if(!ENGINE_set_default_RSA(e))
/* This should only happen when 'e' can't initialise, but the previous
* statement suggests it did. */
abort();
ENGINE_set_default_DSA(e);
ENGINE_set_default_ciphers(e);
/* Release the functional reference from ENGINE_init() */
ENGINE_finish(e);
/* Release the structural reference from ENGINE_by_id() */
ENGINE_free(e);
```

Automatically using builtin ENGINE implementations

Here we'll assume we want to load and register all ENGINE implementations bundled with OpenSSL, such that for any cryptographic algorithm required by OpenSSL - if there is an ENGINE that implements it and can be initialise, it should be used. The following code illustrates how this can work;

```
/* Load all bundled ENGINEs into memory and make them visible */
ENGINE_load_builtin_engines();
/* Register all of them for every algorithm they collectively implement */
ENGINE_register_all_complete();
```

That's all that's required. Eg. the next time OpenSSL tries to set up an RSA key, any bundled ENGINEs that implement RSA_METHOD will be passed to *ENGINE_init()* and if any of those succeed, that ENGINE will be set as the default for RSA use from then on.

Advanced configuration support

There is a mechanism supported by the ENGINE framework that allows each ENGINE implementation to define an arbitrary set of configuration "commands" and expose them to OpenSSL and any applications based on OpenSSL. This mechanism is entirely based on the use of name-value pairs and assumes ASCII input (no unicode or UTF for now!), so it is ideal if applications want to provide a transparent way for users to provide arbitrary configuration "directives" directly to such ENGINEs. It is also possible for the application to dynamically interrogate the loaded ENGINE implementations for the names, descriptions, and input flags of their available "control commands", providing a more flexible configuration scheme. However, if the user is expected to know which ENGINE device he/she is using (in the case of specialised hardware, this goes without saying) then applications may not need to concern themselves with discovering the supported control commands and simply prefer to pass settings into ENGINEs exactly as they are provided by the user.

Before illustrating how control commands work, it is worth mentioning what they are typically used for. Broadly speaking there are two uses for control commands; the first is to provide the necessary details to the implementation (which may know nothing at all specific to the host system) so that it can be initialised for use. This could include the path to any driver or config files it needs to load, required network addresses, smart-card identifiers, passwords to initialise protected devices, logging information, etc etc. This class of commands typically needs to be passed to an ENGINE **before** attempting to initialise it, ie. before calling *ENGINE_init()*. The other class of commands consist of settings or operations that tweak certain behaviour or cause certain operations to take place, and these commands may work either before or after *ENGINE_init()*, or in some cases both. ENGINE implementations should provide indications of this in the descriptions attached to builtin control commands and/or in external product documentation.

Issuing control commands to an ENGINE

Let's illustrate by example; a function for which the caller supplies the name of the ENGINE it wishes to use, a table of string-pairs for use before initialisation, and another table for use after initialisation. Note that the string-pairs used for control commands consist of a command ''name'' followed by the command ''parameter'' - the parameter could be NULL in some cases but the name can not. This function should initialise the ENGINE (issuing the ''pre'' commands beforehand and the ''post'' commands afterwards) and set it as the default for everything except RAND and then return a boolean success or failure.

```
int generic_load_engine_fn(const char *engine_id,
const char **pre_cmds, int pre_num,
const char **post_cmds, int post_num)
{
ENGINE *e = ENGINE_by_id(engine_id);
if(!e) return 0;
while(pre_num--) {
if(!ENGINE_ctrl_cmd_string(e, pre_cmds[0], pre_cmds[1], 0)) {
fprintf(stderr, "Failed command (%s - %s:%s)\n", engine_id,
pre_cmds[0], pre_cmds[1] ? pre_cmds[1] : "(NULL)");
ENGINE_free(e);
return 0;
}
pre_cmds += 2;
}
if(!ENGINE_init(e)) {
fprintf(stderr, "Failed initialisation\n");
ENGINE_free(e);
return 0;
}
/* ENGINE_init() returned a functional reference, so free the structural
* reference from ENGINE_by_id(). */
ENGINE_free(e);
while(post_num--) {
if(!ENGINE_ctrl_cmd_string(e, post_cmds[0], post_cmds[1], 0)) {
fprintf(stderr, "Failed command (%s - %s:%s)\n", engine_id,
post_cmds[0], post_cmds[1] ? post_cmds[1] : "(NULL)");
ENGINE_finish(e);
return 0;
}
post_cmds += 2;
}
ENGINE_set_default(e, ENGINE_METHOD_ALL & ENGINE_METHOD_RAND);
/* Success */
return 1;
}
```

Note that *ENGINE_ctrl_cmd_string()* accepts a boolean argument that can relax the semantics of the function - if set non-zero it will only return failure if the ENGINE supported the given command name but failed while executing it, if the ENGINE doesn't support the command name it will simply return success without doing anything. In this case we assume the user is only supplying commands specific to the given ENGINE so we set this to FALSE.

Discovering supported control commands

It is possible to discover at run-time the names, numerical-ids, descriptions and input parameters of the control commands supported by an ENGINE using a structural reference. Note that some control commands are defined by OpenSSL itself and it will intercept and handle these control commands on behalf of the ENGINE, ie. the ENGINE's *ctrl()* handler is not used for the control command. openssl/engine.h defines an index, ENGINE_CMD_BASE, that all control commands implemented by ENGINEs should be numbered from. Any command value lower than this symbol is considered a ''generic'' command is handled directly by the OpenSSL core routines.

It is using these ''core'' control commands that one can discover the the control commands implemented by a given ENGINE, specifically the commands;

```
#define ENGINE_HAS_CTRL_FUNCTION 10
#define ENGINE_CTRL_GET_FIRST_CMD_TYPE 11
#define ENGINE_CTRL_GET_NEXT_CMD_TYPE 12
#define ENGINE_CTRL_GET_CMD_FROM_NAME 13
#define ENGINE_CTRL_GET_NAME_LEN_FROM_CMD 14
#define ENGINE_CTRL_GET_NAME_FROM_CMD 15
#define ENGINE_CTRL_GET_DESC_LEN_FROM_CMD 16
#define ENGINE_CTRL_GET_DESC_FROM_CMD 17
#define ENGINE_CTRL_GET_CMD_FLAGS 18
```

Whilst these commands are automatically processed by the OpenSSL framework code, they use various properties exposed by each ENGINE to process these queries. An ENGINE has 3 properties it exposes that can affect how this behaves; it can supply a *ctrl()* handler, it can specify ENGINE_FLAGS_MANUAL_CMD_CTRL in the ENGINE's flags, and it can expose an array of control command descriptions. If an ENGINE specifies the ENGINE_FLAGS_MANUAL_CMD_CTRL flag, then it will simply pass all these ''core'' control commands directly to the ENGINE's *ctrl()* handler (and thus, it must have supplied one), so it is up to the ENGINE to reply to these ''discovery'' commands itself. If that flag is not set, then the OpenSSL framework code will work with the following rules;

```
if no ctrl() handler supplied;
ENGINE_HAS_CTRL_FUNCTION returns FALSE (zero),
all other commands fail.
if a ctrl() handler was supplied but no array of control commands;
ENGINE_HAS_CTRL_FUNCTION returns TRUE,
all other commands fail.
if a ctrl() handler and array of control commands was supplied;
ENGINE_HAS_CTRL_FUNCTION returns TRUE,
all other commands proceed processing ...
```

If the ENGINE's array of control commands is empty then all other commands will fail, otherwise; ENGINE_CTRL_GET_FIRST_CMD_TYPE returns the identifier of the first command supported by the ENGINE, ENGINE_GET_NEXT_CMD_TYPE takes the identifier of a command supported by the ENGINE and returns the next command identifier or fails if there are no more, ENGINE_CMD_FROM_NAME takes a string name for a command and returns the corresponding identifier or fails if no such command name exists, and the remaining commands take a command identifier and return properties of the corresponding commands. All except ENGINE_CTRL_GET_FLAGS return the string length of a command name or description, or populate a supplied character buffer with a copy of the command name or description. ENGINE_CTRL_GET_FLAGS returns a bitwise-OR'd mask of the following possible values;

```
#define ENGINE_CMD_FLAG_NUMERIC (unsigned int)0x0001
#define ENGINE_CMD_FLAG_STRING (unsigned int)0x0002
#define ENGINE_CMD_FLAG_NO_INPUT (unsigned int)0x0004
#define ENGINE_CMD_FLAG_INTERNAL (unsigned int)0x0008
```

If the ENGINE_CMD_FLAG_INTERNAL flag is set, then any other flags are purely informational to the caller - this flag will prevent the command being usable for any higher-level ENGINE functions such as *ENGINE_ctrl_cmd_string()*. "INTERNAL" commands are not intended to be exposed to text-based configuration by applications, administrations, users, etc. These can support arbitrary operations via *ENGINE_ctrl()*, including passing to and/or from the control commands data of any arbitrary type. These commands are supported in the discovery mechanisms simply to allow applications determinie if an ENGINE supports certain specific commands it might want to use (eg. application "foo" might query various ENGINEs to see if they implement "FOO_GET_VENDOR_LOGO_GIF" - and ENGINE could therefore decide whether or not to support this "foo"-specific extension).

Future developments

The ENGINE API and internal architecture is currently being reviewed. Slated for possible release in 0.9.8 is support for transparent loading of "dynamic" ENGINEs (built as self-contained shared-libraries). This would allow ENGINE implementations to be provided independently of OpenSSL libraries and/or OpenSSL-based applications, and would also remove any requirement for applications to explicitly use the "dynamic" ENGINE to bind to shared-library implementations.

SEE ALSO

rsa(3), *dsa(3)*, *dh(3)*, *rand(3)*