



Full credit is given to the above companies including the OS that this PDF file was generated!

Rocky Enterprise Linux 9.2 Manual Pages on command 'ASYNC_STATUS_OK.3ossl'

\$ man ASYNC_STATUS_OK.3ossl

ASYNC_WAIT_CTX_NEW(3ossl) OpenSSL ASYNC_WAIT_CTX_NEW(3ossl)

NAME

ASYNC_WAIT_CTX_new, ASYNC_WAIT_CTX_free, ASYNC_WAIT_CTX_set_wait_fd,
ASYNC_WAIT_CTX_get_fd, ASYNC_WAIT_CTX_get_all_fds,
ASYNC_WAIT_CTX_get_changed_fds, ASYNC_WAIT_CTX_clear_fd,
ASYNC_WAIT_CTX_set_callback, ASYNC_WAIT_CTX_get_callback,
ASYNC_WAIT_CTX_set_status, ASYNC_WAIT_CTX_get_status,
ASYNC_callback_fn, ASYNC_STATUS_UNSUPPORTED, ASYNC_STATUS_ERR,
ASYNC_STATUS_OK, ASYNC_STATUS_EAGAIN - functions to manage waiting for
asynchronous jobs to complete

SYNOPSIS

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

#define ASYNC_STATUS_UNSUPPORTED  0

#define ASYNC_STATUS_ERR          1

#define ASYNC_STATUS_OK           2

#define ASYNC_STATUS_EAGAIN       3

typedef int (*ASYNC_callback_fn)(void *arg);

ASYNC_WAIT_CTX *ASYNC_WAIT_CTX_new(void);
```

```

void ASYNC_WAIT_CTX_free(ASYNC_WAIT_CTX *ctx);

int ASYNC_WAIT_CTX_set_wait_fd(ASYNC_WAIT_CTX *ctx, const void *key,
                               OSSL_ASYNC_FD fd,
                               void *custom_data,
                               void (*cleanup)(ASYNC_WAIT_CTX *, const void *,
                                                OSSL_ASYNC_FD, void *));

int ASYNC_WAIT_CTX_get_fd(ASYNC_WAIT_CTX *ctx, const void *key,
                          OSSL_ASYNC_FD *fd, void **custom_data);

int ASYNC_WAIT_CTX_get_all_fds(ASYNC_WAIT_CTX *ctx, OSSL_ASYNC_FD *fd,
                               size_t *numfds);

int ASYNC_WAIT_CTX_get_changed_fds(ASYNC_WAIT_CTX *ctx, OSSL_ASYNC_FD *addfd,
                                   size_t *numaddfds, OSSL_ASYNC_FD *delfd,
                                   size_t *numdelfds);

int ASYNC_WAIT_CTX_clear_fd(ASYNC_WAIT_CTX *ctx, const void *key);

int ASYNC_WAIT_CTX_set_callback(ASYNC_WAIT_CTX *ctx,
                               ASYNC_callback_fn callback,
                               void *callback_arg);

int ASYNC_WAIT_CTX_get_callback(ASYNC_WAIT_CTX *ctx,
                                ASYNC_callback_fn *callback,
                                void **callback_arg);

int ASYNC_WAIT_CTX_set_status(ASYNC_WAIT_CTX *ctx, int status);

int ASYNC_WAIT_CTX_get_status(ASYNC_WAIT_CTX *ctx);

```

DESCRIPTION

For an overview of how asynchronous operations are implemented in OpenSSL see `ASYNC_start_job(3)`. An `ASYNC_WAIT_CTX` object represents an asynchronous "session", i.e. a related set of crypto operations. For example in SSL terms this would have a one-to-one correspondence with an SSL connection.

Application code must create an `ASYNC_WAIT_CTX` using the `ASYNC_WAIT_CTX_new()` function prior to calling `ASYNC_start_job()` (see `ASYNC_start_job(3)`). When the job is started it is associated with the `ASYNC_WAIT_CTX` for the duration of that job. An `ASYNC_WAIT_CTX` should only be used for one `ASYNC_JOB` at any one time, but can be reused after

an ASYNC_JOB has finished for a subsequent ASYNC_JOB. When the session is complete (e.g. the SSL connection is closed), application code cleans up with ASYNC_WAIT_CTX_free().

ASYNC_WAIT_CTXs can have "wait" file descriptors associated with them.

Calling ASYNC_WAIT_CTX_get_all_fds() and passing in a pointer to an

ASYNC_WAIT_CTX in the ctx parameter will return the wait file

descriptors associated with that job in *fd. The number of file

descriptors returned will be stored in *numfds. It is the caller's

responsibility to ensure that sufficient memory has been allocated in

*fd to receive all the file descriptors. Calling

ASYNC_WAIT_CTX_get_all_fds() with a NULL fd value will return no file

descriptors but will still populate *numfds. Therefore, application

code is typically expected to call this function twice: once to get the

number of fds, and then again when sufficient memory has been

allocated. If only one asynchronous engine is being used then normally

this call will only ever return one fd. If multiple asynchronous

engines are being used then more could be returned.

The function ASYNC_WAIT_CTX_get_changed_fds() can be used to detect if

any fds have changed since the last call time ASYNC_start_job()

returned ASYNC_PAUSE (or since the ASYNC_WAIT_CTX was created if no

ASYNC_PAUSE result has been received). The numaddfds and numdelfds

parameters will be populated with the number of fds added or deleted

respectively. *addfd and *delfd will be populated with the list of

added and deleted fds respectively. Similarly to

ASYNC_WAIT_CTX_get_all_fds() either of these can be NULL, but if they

are not NULL then the caller is responsible for ensuring sufficient

memory is allocated.

Implementors of async aware code (e.g. engines) are encouraged to

return a stable fd for the lifetime of the ASYNC_WAIT_CTX in order to

reduce the "churn" of regularly changing fds - although no guarantees

of this are provided to applications.

Applications can wait for the file descriptor to be ready for "read"

using a system function call such as select or poll (being ready for

"read" indicates that the job should be resumed). If no file descriptor is made available then an application will have to periodically "poll" the job by attempting to restart it to see if it is ready to continue.

Async aware code (e.g. engines) can get the current ASYNC_WAIT_CTX from the job via ASYNC_get_wait_ctx(3) and provide a file descriptor to use for waiting on by calling ASYNC_WAIT_CTX_set_wait_fd(). Typically this would be done by an engine immediately prior to calling ASYNC_pause_job() and not by end user code. An existing association with a file descriptor can be obtained using ASYNC_WAIT_CTX_get_fd() and cleared using ASYNC_WAIT_CTX_clear_fd(). Both of these functions requires a key value which is unique to the async aware code. This could be any unique value but a good candidate might be the ENGINE * for the engine. The custom_data parameter can be any value, and will be returned in a subsequent call to ASYNC_WAIT_CTX_get_fd(). The ASYNC_WAIT_CTX_set_wait_fd() function also expects a pointer to a "cleanup" routine. This can be NULL but if provided will automatically get called when the ASYNC_WAIT_CTX is freed, and gives the engine the opportunity to close the fd or any other resources. Note: The "cleanup" routine does not get called if the fd is cleared directly via a call to ASYNC_WAIT_CTX_clear_fd().

An example of typical usage might be an async capable engine. User code would initiate cryptographic operations. The engine would initiate those operations asynchronously and then call ASYNC_WAIT_CTX_set_wait_fd() followed by ASYNC_pause_job() to return control to the user code. The user code can then perform other tasks or wait for the job to be ready by calling "select" or other similar function on the wait file descriptor. The engine can signal to the user code that the job should be resumed by making the wait file descriptor "readable". Once resumed the engine should clear the wake signal on the wait file descriptor.

As well as a file descriptor, user code may also be notified via a callback. The callback and data pointers are stored within the ASYNC_WAIT_CTX along with an additional status field that can be used

for the notification of retries from an engine. This additional method can be used when the user thinks that a file descriptor is too costly in terms of CPU cycles or in some context where a file descriptor is not appropriate.

`ASYNC_WAIT_CTX_set_callback()` sets the callback and the callback argument. The callback will be called to notify user code when an engine completes a cryptography operation. It is a requirement that the callback function is small and nonblocking as it will be run in the context of a polling mechanism or an interrupt.

`ASYNC_WAIT_CTX_get_callback()` returns the callback set in the `ASYNC_WAIT_CTX` structure.

`ASYNC_WAIT_CTX_set_status()` allows an engine to set the current engine status. The possible status values are the following:

`ASYNC_STATUS_UNSUPPORTED`

The engine does not support the callback mechanism. This is the default value. The engine must call `ASYNC_WAIT_CTX_set_status()` to set the status to some value other than `ASYNC_STATUS_UNSUPPORTED` if it intends to enable the callback mechanism.

`ASYNC_STATUS_ERR`

The engine has a fatal problem with this request. The user code should clean up this session.

`ASYNC_STATUS_OK`

The request has been successfully submitted.

`ASYNC_STATUS_EAGAIN`

The engine has some problem which will be recovered soon, such as a buffer is full, so user code should resume the job.

`ASYNC_WAIT_CTX_get_status()` allows user code to obtain the current status value. If the status is any value other than `ASYNC_STATUS_OK` then the user code should not expect to receive a callback from the engine even if one has been set.

An example of the usage of the callback method might be the following.

User code would initiate cryptographic operations, and the engine code would dispatch this operation to hardware, and if the dispatch is

successful, then the engine code would call `ASYNC_pause_job()` to return control to the user code. After that, user code can perform other tasks. When the hardware completes the operation, normally it is detected by a polling function or an interrupt, as the user code set a callback by calling `ASYNC_WAIT_CTX_set_callback()` previously, then the registered callback will be called.

RETURN VALUES

`ASYNC_WAIT_CTX_new()` returns a pointer to the newly allocated `ASYNC_WAIT_CTX` or `NULL` on error.

`ASYNC_WAIT_CTX_set_wait_fd`, `ASYNC_WAIT_CTX_get_fd`, `ASYNC_WAIT_CTX_get_all_fds`, `ASYNC_WAIT_CTX_get_changed_fds`, `ASYNC_WAIT_CTX_clear_fd`, `ASYNC_WAIT_CTX_set_callback`, `ASYNC_WAIT_CTX_get_callback` and `ASYNC_WAIT_CTX_set_status` all return 1 on success or 0 on error. `ASYNC_WAIT_CTX_get_status()` returns the engine status.

NOTES

On Windows platforms the `<openssl/async.h>` header is dependent on some of the types customarily made available by including `<windows.h>`. The application developer is likely to require control over when the latter is included, commonly as one of the first included headers. Therefore, it is defined as an application developer's responsibility to include `<windows.h>` prior to `<openssl/async.h>`.

SEE ALSO

`crypto(7)`, `ASYNC_start_job(3)`

HISTORY

`ASYNC_WAIT_CTX_new()`, `ASYNC_WAIT_CTX_free()`, `ASYNC_WAIT_CTX_set_wait_fd()`, `ASYNC_WAIT_CTX_get_fd()`, `ASYNC_WAIT_CTX_get_all_fds()`, `ASYNC_WAIT_CTX_get_changed_fds()` and `ASYNC_WAIT_CTX_clear_fd()` were added in OpenSSL 1.1.0. `ASYNC_WAIT_CTX_set_callback()`, `ASYNC_WAIT_CTX_get_callback()`, `ASYNC_WAIT_CTX_set_status()`, and `ASYNC_WAIT_CTX_get_status()` were added in OpenSSL 3.0.

Copyright 2016-2021 The OpenSSL Project Authors. All Rights Reserved.
Licensed under the Apache License 2.0 (the "License"). You may not use
this file except in compliance with the License. You can obtain a copy
in the file LICENSE in the source distribution or at
<<https://www.openssl.org/source/license.html>>.

3.0.7 2023-07-13 ASYNC_WAIT_CTX_NEW(3ossl)