NAME

ASYNC_get_wait_ctx, ASYNC_init_thread, ASYNC_cleanup_thread, ASYNC_start_job, ASYNC_pause_job, ASYNC_get_current_job, ASYNC_block_pause, ASYNC_unblock_pause, ASYNC_is_capable - asynchronous job management functions

SYNOPSIS

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

int ASYNC_init_thread(size_t max_size, size_t init_size);
void ASYNC_cleanup_thread(void);

int ASYNC_start_job(ASYNC_JOB **job, ASYNC_WAIT_CTX *ctx, int *ret,
                    int (*func)(void *), void *args, size_t size);
int ASYNC_pause_job(void);

ASYNC_JOB *ASYNC_get_current_job(void);
ASYNC_WAIT_CTX *ASYNC_get_wait_ctx(ASYNC_JOB *job);
void ASYNC_block_pause(void);
void ASYNC_unblock_pause(void);

int ASYNC_is_capable(void);
```

DESCRIPTION

OpenSSL implements asynchronous capabilities through an **ASYNC_JOB**. This represents code that can be started and executes until some event occurs. At that point the code can be paused and control returns to user code until some subsequent event indicates that the job can be resumed.

The creation of an **ASYNC_JOB** is a relatively expensive operation. Therefore, for efficiency reasons, jobs can be created up front and reused many times. They are held in a pool until they are needed, at which point they are removed from the pool, used, and then returned to the pool when the job completes. If the user application is multi-threaded, then **ASYNC_init_thread()** may be called for each thread that will initiate asynchronous jobs. Before user code exits per-thread resources need to be cleaned up. This will normally occur automatically (see **OPENSSL_init_crypto**(3)) but may be explicitly initiated by using **ASYNC_cleanup_thread()**. No asynchronous jobs must be outstanding for the thread when **ASYNC_cleanup_thread()** is called. Failing to ensure this will result in memory leaks.

The *max_size* argument limits the number of **ASYNC_JOB**s that will be held in the pool. If *max_size* is set to 0 then no upper limit is set. When an **ASYNC_JOB** is needed but there are none available in the pool already then one will be automatically created, as long as the total of **ASYNC_JOB**s managed by the pool does not exceed *max_size*. When the pool is first initialised *init_size* **ASYNC_JOB**s will be created immediately. If **ASYNC_init_thread()** is not called before the pool is first used then it will be called automatically with a *max_size* of 0 (no upper limit) and an *init_size* of 0 (no **ASYNC_JOB**s created up front).

An asynchronous job is started by calling the **ASYNC_start_job()** function. Initially **job* should be NULL. *ctx* should point to an **ASYNC_WAIT_CTX** object created through the **ASYNC_WAIT_CTX_new**(3) function. *ret* should point to a location where the return value of the asynchronous function should be stored on completion of the job. *func* represents the function that should be started asynchronously. The data pointed to by *args* and of size *size* will be copied and then passed as an argument to *func* when the job starts. ASYNC_start_job will return one of the following values:

ASYNC_ERR
An error occurred trying to start the job. Check the OpenSSL error queue (e.g. see **ERR_print_errors**(3)) for more details.

ASYNC_NO_JOBS
There are no jobs currently available in the pool. This call can be retried again at a later time.

ASYNC_PAUSE
The job was successfully started but was "paused" before it completed (see **ASYNC_pause_job()** below). A handle to the job is placed in **job*. Other work can be performed (if desired) and the job restarted at a later time. To restart a job call **ASYNC_start_job()** again passing the job handle in **job*. The *func*, *args* and *size* parameters will be ignored when restarting a job. When restarting a job **ASYNC_start_job() must** be called from the same thread that the job was originally started from.

ASYNC_FINISH
The job completed. **job* will be NULL and the return value from *func* will be placed in **ret*.

At any one time there can be a maximum of one job actively running per thread (you can have many that are paused). **ASYNC_get_current_job()** can be used to get a pointer to the currently executing **ASYNC_JOB**. If no job is currently executing then this will return NULL.

If executing within the context of a job (i.e. having been called directly or indirectly by the function "func" passed as an argument to **ASYNC_start_job()**) then **ASYNC_pause_job()** will immediately return control to the calling application with **ASYNC_PAUSE** returned from the **ASYNC_start_job()** call. A subsequent call to ASYNC_start_job passing in the relevant **ASYNC_JOB** in the **job* parameter will resume execution from the **ASYNC_pause_job()** call. If **ASYNC_pause_job()** is called whilst not within the context of a job then no action is taken and **ASYNC_pause_job()** returns immediately.

ASYNC_get_wait_ctx() can be used to get a pointer to the **ASYNC_WAIT_CTX** for the *job*. **ASYNC_WAIT_CTX**s contain two different ways to notify applications that a job is ready to be resumed. One is a "wait" file descriptor, and the other is a "callback" mechanism.

The "wait" file descriptor associated with **ASYNC_WAIT_CTX** is used for applications to 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_WAIT_CTXs also have a "callback" mechanism to notify applications. The callback is set by an application, and it will be automatically called when an engine completes a cryptography operation, so that the application can resume the paused work flow without polling. An engine could be written to look whether the callback has been set. If it has then it would use the callback mechanism in preference to the file descriptor notifications. If a callback is not set then the engine may use file descriptor based notifications. Please note that not all engines may support the callback mechanism, so the callback may not be used even if it has been set. See **ASYNC_WAIT_CTX_new()** for more details.

The **ASYNC_block_pause()** function will prevent the currently active job from pausing. The block will remain in place until a subsequent call to **ASYNC_unblock_pause()**. These functions can be nested, e.g. if you call **ASYNC_block_pause()** twice then you must call **ASYNC_unblock_pause()** twice in order to re-enable pausing. If these functions are called while there is no currently active job then they have no effect. This functionality can be useful to avoid deadlock scenarios. For example during the execution of an **ASYNC_JOB** an application acquires a lock. It then calls some cryptographic function which invokes **ASYNC_pause_job()**. This returns control back to the code that created the **ASYNC_JOB**. If that code then attempts to acquire the same lock before resuming the original job then a deadlock can occur. By calling **ASYNC_block_pause()** immediately after acquiring the lock and **ASYNC_unblock_pause()** immediately before releasing it then this situation cannot occur.

Some platforms cannot support async operations. The **ASYNC_is_capable()** function can be used to detect whether the current platform is async capable or not.

RETURN VALUES

ASYNC_init_thread returns 1 on success or 0 otherwise.

ASYNC_start_job returns one of **ASYNC_ERR**, **ASYNC_NO_JOBS**, **ASYNC_PAUSE** or **ASYNC_FINISH** as described above.

ASYNC_pause_job returns 0 if an error occurred or 1 on success. If called when not within the context of an **ASYNC_JOB** then this is counted as success so 1 is returned.

ASYNC_get_current_job returns a pointer to the currently executing **ASYNC_JOB** or NULL if not within the context of a job.

ASYNC_get_wait_ctx() returns a pointer to the **ASYNC_WAIT_CTX** for the job.

ASYNC_is_capable() returns 1 if the current platform is async capable or 0 otherwise.

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>*.

EXAMPLES

The following example demonstrates how to use most of the core async APIs:

```
#ifdef _WIN32
# include <windows.h>
#endif
#include <stdio.h>
#include <unistd.h>
#include <openssl/async.h>
#include <openssl/crypto.h>

int unique = 0;

void cleanup(ASYNC_WAIT_CTX *ctx, const void *key, OSSL_ASYNC_FD r, void *vw)
{
    OSSL_ASYNC_FD *w = (OSSL_ASYNC_FD *)vw;

    close(r);
    close(*w);
    OPENSSL_free(w);
}

int jobfunc(void *arg)
{
```

```
ASYNC_JOB *currjob;
unsigned char *msg;
int pipefds[2] = {0, 0};
OSSL_ASYNC_FD *wptr;
char buf = 'X';

currjob = ASYNC_get_current_job();
if (currjob != NULL) {
    printf("Executing within a job\n");
} else {
    printf("Not executing within a job - should not happen\n");
    return 0;
}

msg = (unsigned char *)arg;
printf("Passed in message is: %s\n", msg);

if (pipe(pipefds) != 0) {
    printf("Failed to create pipe\n");
    return 0;
}
wptr = OPENSSL_malloc(sizeof(OSSL_ASYNC_FD));
if (wptr == NULL) {
    printf("Failed to malloc\n");
    return 0;
}
*wptr = pipefds[1];
ASYNC_WAIT_CTX_set_wait_fd(ASYNC_get_wait_ctx(currjob), &unique,
                           pipefds[0], wptr, cleanup);

/*
 * Normally some external event would cause this to happen at some
 * later point - but we do it here for demo purposes, i.e.
 * immediately signalling that the job is ready to be woken up after
 * we return to main via ASYNC_pause_job().
 */
write(pipefds[1], &buf, 1);

/* Return control back to main */
ASYNC_pause_job();
```

```
    /* Clear the wake signal */
    read(pipefds[0], &buf, 1);

    printf ("Resumed the job after a pause\n");

    return 1;
}

int main(void)
{
    ASYNC_JOB *job = NULL;
    ASYNC_WAIT_CTX *ctx = NULL;
    int ret;
    OSSL_ASYNC_FD waitfd;
    fd_set waitfdset;
    size_t numfds;
    unsigned char msg[13] = "Hello world!";

    printf("Starting...\n");

    ctx = ASYNC_WAIT_CTX_new();
    if (ctx == NULL) {
        printf("Failed to create ASYNC_WAIT_CTX\n");
        abort();
    }

    for (;;) {
        switch (ASYNC_start_job(&job, ctx, &ret, jobfunc, msg, sizeof(msg))) {
        case ASYNC_ERR:
        case ASYNC_NO_JOBS:
            printf("An error occurred\n");
            goto end;
        case ASYNC_PAUSE:
            printf("Job was paused\n");
            break;
        case ASYNC_FINISH:
            printf("Job finished with return value %d\n", ret);
            goto end;
        }
        /* Wait for the job to be woken */
        printf("Waiting for the job to be woken up\n");

        if (!ASYNC_WAIT_CTX_get_all_fds(ctx, NULL, &numfds)
                || numfds > 1) {
            printf("Unexpected number of fds\n");
            abort();
        }
        ASYNC_WAIT_CTX_get_all_fds(ctx, &waitfd, &numfds);
        FD_ZERO(&waitfdset);
        FD_SET(waitfd, &waitfdset);
        select(waitfd + 1, &waitfdset, NULL, NULL, NULL);
    }

end:
    ASYNC_WAIT_CTX_free(ctx);
    printf("Finishing\n");

    return 0;
}
```

The expected output from executing the above example program is:

```
Starting...
Executing within a job
Passed in message is: Hello world!
Job was paused
Waiting for the job to be woken up
Resumed the job after a pause
Job finished with return value 1
Finishing
```

SEE ALSO

crypto(7), **ERR_print_errors**(3)

HISTORY

ASYNC_init_thread, ASYNC_cleanup_thread, ASYNC_start_job, ASYNC_pause_job, ASYNC_get_current_job, **ASYNC_get_wait_ctx()**, **ASYNC_block_pause()**, **ASYNC_unblock_pause()** and **ASYNC_is_capable()** were first added in OpenSSL 1.1.0.

COPYRIGHT