



RtCreateProcess(...)

RtCreateProcess creates and starts a new RTSS process. The new RTSS process runs the specified RTSS executable file. Function is supported **only** in the Win32 environment.

Prototype

BOOL

RtCreateProcess(

```
    LPCTSTR lpApplicationName,  
    LPTSTR lpCommandLine,  
    LPSECURITY_ATTRIBUTES lpProcessAttributes,  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    BOOL bInheritHandles,  
    DWORD dwCreationFlags,  
    LPVOID lpEnvironment,  
    LPCTSTR lpCurrentDirectory,  
    LPSTARTUPINFO lpStartupInfo,  
    LPPROCESS_INFORMATION lpProcessInformation  
);
```



RtCreateProcess(...)

Parameters

lpApplicationName

pointer to a null-terminated string that specifies the module to execute. The string must specify the full path and file name of the module to execute.

The *lpApplicationName* parameter can be NULL. In that case, the module name must be the first white space-delimited token in the *lpCommandLine* string.

lpCommandLine

pointer to a null-terminated string that specifies the command line to execute. The system adds a null character to the command line.

The *lpCommandLine* parameter can be NULL. In that case, the function uses the string pointed to by *lpApplicationName* as the command line.



RtCreateProcess(...)

Parameters

<i>lpProcessAttributes</i>	ignored in RTSS environment
<i>lpThreadAttributes</i>	ignored in RTSS environment
<i>bInheritHandles</i>	ignored in RTSS environment
<i>dwCreationFlags</i>	ignored in RTSS environment
<i>lpEnvironment</i>	ignored in RTSS environment
<i>lpCurrentDirectory</i>	ignored in RTSS environment
<i>lpStartupInfo</i>	ignored in RTSS environment
<i>lpProcessInformation</i>	Pointer to a PROCESS_INFORMATION structure that receives identification information about the new process



RtCreateProcess(...)

Return value

If the function succeeds, the return value is nonzero.

If the function **fails**, the return value is **zero**. To get extended error information, call `GetLastError()`.

`RtCreateProcess` does not create a handle for the primary thread of the new RTSS process. The returning value of *hThread* is NULL and *dwThreadId* is 0.

The preferred way to shut down an RTSS process is by using `ExitProcess(...)`, because this function sends notification of approaching termination to all RTDLLs attached to the RTSS process. Other means of shutting down an RTSS process do not notify the attached RTDLLs.

The created RTSS process remains in the system until all threads within the RTSS process have terminated and all handles to the RTSS process and any of its threads have been closed through calls to `RtCloseHandle()`.



PROCESS_INFORMATION

```
typedef struct _PROCESS_INFORMATION {  
HANDLE hProcess;  
HANDLE hThread;  
DWORD dwProcessId;  
DWORD dwThreadId;  
} PROCESS_INFORMATION;
```

hProcess Handle to the newly created process. The handle is used to specify the process in all functions that perform operations on the process object.

hThread Handle to the primary thread of the newly created process. The handle is used to specify the thread in all functions that perform operations on the thread object.

dwProcessId Value that can be used to identify a process. The value is valid from the time the process is created until the time the process is terminated.

dwThreadId Value that can be used to identify a thread. The value is valid from the time the thread is created until the time the thread is terminated.



ExitProcess(...)

ExitProcess ends a process and all its threads.

Prototype

```
VOID  
    ExitProcess(  
        UINT uExitCode  
    );
```

Parameters

uExitCode ignored in RTSS

Return Values

This function does not return a value.



CreateThread(...)

CreateThread creates a thread to execute within the address space of the calling process.

Prototype

HANDLE

CreateThread(

LPSECURITY_ATTRIBUTES

DWORD

LPTHREAD_START_ROUTINE

LPVOID

DWORD

LPDWORD

);

lpThreadAttributes,

StackSize,

lpStartAddress,

lpParameter,

dwCreationFlags,

lpThreadId



CreateThread(...)

Parameters

lpThreadAttributes

ignored in RTSS

StackSize

the size, in bytes, of the stack for the new thread. If 0 is specified, the stack size defaults to 8192 bytes. In the RTSS environment, the stack **cannot** grow. The number of bytes specified by *StackSize* must be available from non-paged memory in the kernel.

lpStartAddress

A pointer to the application-supplied function to be executed by the thread and represents the starting address of the thread. The function accepts a single 32-bit argument and returns a 32-bit exit value.

lpParameter

A single 32-bit parameter value passed to the thread.



CreateThread(...)

Parameters

`dwCreationFlags`

Additional flags that control the creation of the thread. If the `CREATE_SUSPENDED` flag is specified, the thread is created in a suspended state and will not run until `ResumeThread` is called. If this value is zero, the thread runs immediately after creation.

`lpThreadId`

A pointer to a 32-bit variable that receives the thread identifier.

Return Values

If the function succeeds, the return value is a handle to the new thread.

If the function fails, the return value is `NULL`. To get extended error information, call `GetLastError`.



SuspendThread(...)

SuspendThread suspends the specified thread.

Prototype

DWORD

```
SuspendThread(  
    HANDLE hThread  
) ;
```

Parameters

hThread The thread to suspend.

Return Values

If the function succeeds, the return value is the thread's previous suspend count; otherwise, it is 0xFFFFFFFF. To get extended error information, use `GetLastError`.

Each thread has a suspend count (with a maximum value of `MAXIMUM_SUSPEND_COUNT`). If the suspend count is greater than zero, the thread is suspended; otherwise, the thread is not suspended and is eligible for execution. Calling `SuspendThread` causes the target thread's suspend count to be raised by one. Attempting to increment past the maximum suspend count causes an error without incrementing the count.



ResumeThread(...)

ResumeThread subtracts one from a thread's suspend count. When the suspend count is reduced to zero, the execution of the thread is resumed.

Prototype

DWORD

```
ResumeThread(  
    HANDLE hThread  
);
```

Parameters

hThread A handle for the thread to be restarted.

Return Values

If the function succeeds, the return value is the thread's previous suspend count.

If the function fails, the return value is 0xFFFFFFFF.

To get extended error information, call `GetLastError`.



RtSetThreadPriority(...)

RtSetThreadPriority sets the priority value for the specified thread.

Prototype

BOOL

```
RtSetThreadPriority(  
    HANDLE      hThread,  
    int  nPriority  
);
```

Parameters

<i>hThread</i>	the thread whose priority value is to be set.
<i>nPriority</i>	a priority level from 0 to 127, where 127 identifies the highest priority thread

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE.

To get extended error information, call `GetLastError()`.



RtGetThreadPriority(...)

RtGetThreadPriority sets the priority value for the specified thread.

Prototype

```
int  
RtGetThreadPriority(  
    HANDLE          hThread,  
) ;
```

Parameters

hThread the thread whose priority value is to be set.

Return Values

If the function succeeds, the return value is the thread's priority level.

If the function fails, the return value is `THREAD_PRIORITY_ERROR_RETURN`.
To get extended error information, call `GetLastError()`.



ExitThread(...)

ExitThread ends a thread.

Prototype

```
VOID  
ExitThread(  
    DWORD ExitCode  
);
```

Parameters

ExitCode The exit code for the calling thread. Use `GetExitCodeThread` to retrieve a thread's exit code.

Return Values

This function does not return a value.

`ExitThread` is the preferred method of exiting a thread. When this function is called (either explicitly or by returning from a thread procedure), the current thread's stack is de-allocated and the thread terminates.

If the thread is the last thread in the process when this function is called, the thread's process is also terminated.

Terminating a thread does not necessarily remove the thread object from the operating system. A thread object is deleted when the last handle to the thread is closed.



GetExitCodeThread(...)

ExitThread retrieves the termination status of the specified thread.

Prototype

BOOL

```
GetExitCodeThread(  
    HANDLE          hThread,  
    LPDWORD lpExitCode  
);
```

Parameters

hThread

the thread identifier.

lpExitCode

a pointer to a 32-bit variable to receive the thread termination status.

If the specified thread has not terminated, the termination status returned is STILL_ACTIVE.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE.

To get extended error information, call GetLastError().



MRTS – RTX6.5



```

/*****
* CreateComputeThreads(...)
*
* Description:
* Creates compute threads for particular streams
*****/
BOOL CreateComputeThreads(void) {
int parameter;

    parameter = 0;
    ComputeThreadsTerminate[0] = FALSE;
    hComputeThreads[0] = NULL;
    hComputeThreads[0] = RtCreateThread( 0, 0, ComputeThread_0, &parameter, CREATE_SUSPENDED, 0);
    if ( hComputeThreads[0] == NULL ) {
        LOG_ERROR_MESSAGE();
        return FALSE;
    }
    sprintf(message, "OK\t\tThread 'hComputeThreads[0]' was created. HANDLER = 0x%X.", hComputeThreads[0]);
    LOG_MESSAGE();
    if (RtSetThreadPriority(hComputeThreads[0], RT_PRIORITY_MAX - 5) ) {
        sprintf(message, "OK\t\t'hComputeThreads[0]' priority was set to: RT_PRIORITY_MAX - 5");
        LOG_MESSAGE();
    }
    else {
        LOG_ERROR_MESSAGE();
        return FALSE;
    }
    if (RtResumeThread(hComputeThreads[0]) != 0xFFFFFFFF){
        sprintf(message, "OK\t\t'hComputeThreads[0]' was resumed.");
        LOG_MESSAGE();
    }
    else {
        LOG_ERROR_MESSAGE();
        return FALSE;
    }
}
return TRUE;
}

```



MRTS – RTX6.5



```

//*****
// ComputeThread_0(...)
//
// Description:
// Compute thread for stream 0
//*****
ULONG RTFCNDCL ComputeThread_0( void *nContext ) {
int stream;

    stream = *((int *) nContext);
    while (!ComputeThreadsTerminate[stream]) {
        if (RtWaitForSingleObject(hComputeEvents[stream], 100) == WAIT_OBJECT_0) {
            RtResetEvent(hComputeEvents[stream]);
            StreamProcessing(stream);
        }
        else continue;
    }
    ComputeThreadsTerminated[stream] = TRUE;
return 0;
}

```