Disclaimer to Recruiters

Please be aware of the following:

1. The following samples were intended to be viewed as part of a larger online help system. As such, there will be minor, non-content related differences between these samples and their final published versions (i.e. dead links, branding, etc.)
2. The following samples will showcase my ability to write for a technical audience. Please inquire about samples meant for a non-technical audience.

Measuring Timer Latencies with SRTM / Measuring HAL-Level Timer Latencies with KSRTM - Technical Concepts

Supported Visual Studio Breakpoint Features - Technical Features

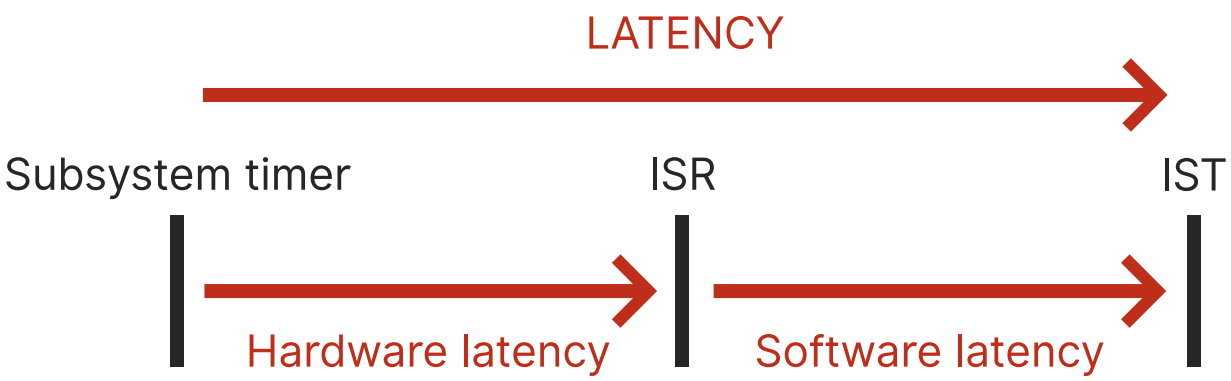
Attaching to a Remote Running RTSS Process - Technical Process

RtQuerySystemPerfCaps - API Documentation

1

Measuring Timer Latencies with SRTM

System Timer Response Latency is measured as the total time from when a Subsystem timer interrupt sends a signal to when that signal is recognized by software running in an Interrupt Service Thread (IST). As shown in the diagram below, Latency is made up of Hardware latency, the time it takes for a signal to be recognized by an Interrupt Service Routine (ISR), and Software latency, the time it takes from ISR to the routine running in IST.



The RTX64 Runtime provides a utility called **System Response Time Measurement (SRTM)**, a real-time API timer latency measurement tool that measures timer latency observed by the application. Each of the two supplied versions of SRTM, Windows and RTSS, measures its own environment. An SRTM histogram often shows a characteristic double peak profile: the first peak is a near-best case; the second is the dirty-cache case. The diagram occasionally shows additional smaller spikes from processor-level interrupt masking. With the introduction of RTSS timer tick compensation in RTX64 4.1, the diagram may show a longer tail and larger maximum latency on systems with bus contentions between Windows cores and RTSS cores.

By default, this tool generates a 15-second tone and prints a histogram of timer response latencies. You can run SRTM in the RTSS environment to measure RTSS latencies.

To measure timer latency, type the following at a command prompt:

```
RTSSrun "c:\program files\IntervalZero\RTX64\bin\srtm.rtss" [-h][-s][-1][f]
seconds_to_sample
```

RELATED TOPICS:

- Measuring HAL-Level Timer Latencies
- KSRTM versus SRTM
- Using SRTM
- Using KSRTM
- RTX64 Latency View

Using SRTM

SRTM (System Response Time Measurement) is an RTAPI timer latency measurement tool that measures timer latency observed by an application. There are two supplied versions: one for the Windows environment (srtm.exe), the other for an RTSS environment (srtm.rtss).

Usage

```
srtm [/?] [/h] [/s] [/1] [/f] [/n num] seconds_to_sample
```

Parameters

/h

Display histogram (in addition to summary)

> **NOTE:** The histogram in the Windows version (srtm.exe) may not match the maximum latency value. This is due to Windows being non-deterministic.

/s

Turn on sound (square wave driven by timer)

/1

Use a 10 MS timer period (default is 1 MS)

/f

Use fastest available timer (1MS or better)

/n num

Multiple SRTM instances aware, where *num* is the total number of instances

seconds_to_sample

Duration in seconds to sample timer response latencies.

/?

Help on usage

If no parameters are given, the default is `srtm /h /s /f 15`

Remarks

SRTM is also provided as sample code.

mSRTM is another sample that shows how to measure latency across multiple cores.

The RTSS timer latency observed by an application is made up of hardware and software latency:

- **Hardware latency** is the time it takes for a signal to be recognized by the HAL Interrupt Service Routine (ISR).
- **Software latency** is the time it takes from an ISR to the routine running in an IST.

System Management Interrupt (SMI) and bus contention between Windows cores and RTSS cores is a major source of hardware latency. To mitigate the side effects of hardware latency on RTSS timer latency and RTSS time, RTX64 uses a timer tick compensation algorithm. This algorithm uses the two Time Stamp Count (TSC) readings between previous and current ISRs to calculate the number of ticks. ISR then uses the calculated number of ticks (instead of 1 tick) to check user timer expiration and to increment RTSS time. If SMI or bus contention occurs at the early stage of user timer period, its handling routine will be called on time. If this

contention occurs at the later stage, the current call is later, but the subsequent call will occur on time because of the reduced number of ticks to expire.

SRTM calculates the timer latency by subtracting the expected time from the time obtained by calling **RtGetClockTime**. The expected time is always incremented by the user timer period, instead of the previous time added to the user timer period. Without timer tick compensation, the time obtained by **RtGetClockTime** may be slower when compared with universal time. However, because RTSS time always increments by 1 tick, such time drift is not reflected in the difference of the expected time from the time obtained by **RtGetClockTime**. Without timer tick compensation, SRTM results may not fully reflect the side effect of SMI or bus contention. With timer tick compensation, the time obtained by **RtGetClockTime** is much closer to the universal time. Therefore, SRTM will fully reflect the side effect of SMI or bus contention.

Beginning with RTX64 4.1, timer tick compensation is the default configuration. Timer tick compensation will decrease occurrences of user timer latency jitters, but it will not reduce the absolute value of jitters (that is, when SMI or bus contention occurs at the last tick of the user timer period). Therefore, SRTM may show a longer tail in its diagram and larger maximum latency on systems with bus contentions between Windows cores and RTSS cores.

If you already have a workaround for timer latency jitters, see the TechNote *Real-Time Subsystem Timer Tick Compensation* for instructions on how to disable timer tick compensation.

RELATED TOPICS:

- Measuring Timer Latencies
- Measuring HAL-Level Timer Latencies
- KSRTM versus SRTM
- Using KSRTM
- RTX64 Latency View

2

Measuring HAL-Level Timer Latencies

Kernel System Response Time Measurement (KSRTM) is a driver and a Windows utility that measures HAL-level timer latency. Short code paths make it less sensitive to cache jitter than SRTM. It can determine which Windows OS component or device driver is causing the greatest latency event for a real-time application.

KSRTM measures timer response latencies and obtains reports and histograms of the results.

This tool exercises the real-time HAL extension, and measures and reports timer response latencies against a synchronized clock. It can present a histogram of such latencies and report which kernel component was running when the longest latency event occurred. It can also provide a list of drivers. KSRTM optionally measures the latencies associated with kernel (DPC-level) and multimedia (user-level) timers (single core).

It also allows the timer to drive the sound speaker (a 500-microsecond timer will generate a 1000 Hz square wave). Any timing jitter will produce an audible frequency wobble. This is only practical when RTX is configured to use one real time core.

RELATED TOPICS:

- Measuring Timer Latencies
- KSRTM versus SRTM
- Using SRTM
- Using KSRTM
- RTX64 Latency View

Using KSRTM

Kernel System Response Time Measurement (KSRTM) is a Windows driver and utility that measures HAL-level timer latencies on the RTSS cores and provides reports and histograms of the results.

To measure HAL-level timer latency, run command prompt *as Administrator* and type the following:

```
"c:\program files\IntervalZero\rtx64\bin\ksrtm.exe" [/r|/k|/m][/h][/n][/s][/d]
[/l][/u minutes] seconds_to_sample
```

> **IMPORTANT!** Do not attempt to stop the RTX64 subsystem while KSRTM is running. The subsystem may become unstable and crash.

Parameters

/r

Real-time HAL extension timer (default). Exercises the KSRTM test using the RTX64 hardware abstraction layer timer. Runs on as many cores as RTX64 is configured to use for real time.

/k

Kernel (DPC-level) timer. Exercises the KSRTM test using the Windows kernel timer. Only runs on one CPU.

/m

Multimedia (user-level) timer. Exercises the KSRTM test using the multimedia timer. Only runs on one CPU.

/h

Displays histogram of latencies (in addition to summary).

/n

Do not use real-time process and thread priorities. Avoids setting highest real-time priority (relevant only for the multimedia user level timer).

/s

Sound output (square wave driven by timer). Only practical when RTX64 is configured to use one real time core. With two or more cores driving the speaker, there are collisions leading to “dirty” sound.

/d

Display loaded driver information. Causes the program to display information about currently loaded drivers, such as the name, start address, and end address.

/l

Longest latency event information. Only supported for the RTX64 HAL timer test [/r].

/u minutes

Minutes between display updates. For tests that are set to last longer than a minute (seconds_to_ sample>60), this option forces the program to display the current data after the specified number of minutes.

seconds_to_sample

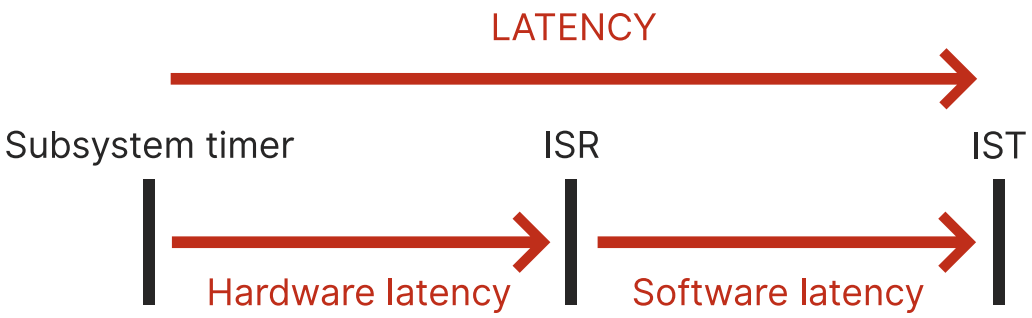
Duration in seconds to sample timer response latencies.

RELATED TOPICS:

- Measuring Timer Latencies
- Measuring HAL-Level Timer Latencies
- KSRTM versus SRTM
- Using SRTM
- RTX64 Latency View

KSRTM versus SRTM

Both KSRTM (Kernel System Response Time Measurement) and SRTM (System Response Time Measurement) measure timer latencies. While KSRTM provides detailed information about the source of the worst-case latency, it measures only the time to the beginning of the Interrupt Service Routine (ISR). SRTM measures the total time to an RTSS program's timer handler (IST) (i.e., a full thread context); it is the more realistic measurement of latencies encountered by RTSS applications.



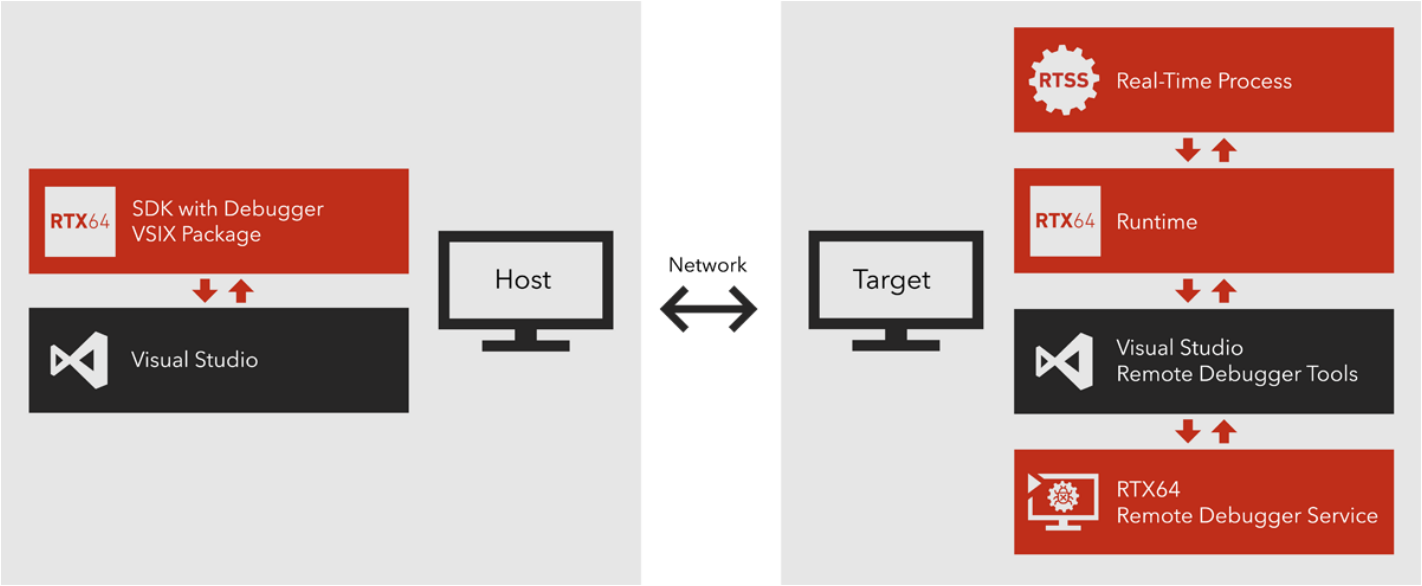
The worst-case SRTM for the RTX64 environment (srtm.rtss) reported latencies range from 2 to 20 microseconds on most Pentium-based systems. The worst-case SRTM for the Windows environment (srtm.exe) reported latencies range from 30 to 100 microseconds on most Pentium-based systems.

RELATED TOPICS:

- Measuring Timer Latencies
- Measuring HAL-Level Timer Latencies
- Using SRTM
- Using KSRTM
- RTX64 Latency View

Attaching to a Remote Running RTSS Process

You can attach the Visual Studio debugger to an RTSS process running on a remote system outside of the Visual Studio IDE. This requires communication between the host system, which runs the Visual Studio IDE, and the target system where the process resides.



This topic provides instructions for configuring the Host-Target connection. For information on attaching to an RTSS process on a local system, see Attaching to a Local Running RTSS Process.

NOTE: You cannot use Task Manager or RtssKill to terminate a RTSS process that is being debugged.

NOTE: You cannot stop the Subsystem on the target system while RTSS processes are being debugged.

SECTIONS IN THIS TOPIC:

- Setting up the Target System
- Attaching to a Process Running on the Target System

- Firewall Configuration
- Troubleshooting

Setting up the Target System

On the target system, install and configure the Visual Studio remote tools so the host system can connect over the network using Visual Studio. Then, configure remote debugging through the RTX64 Control Panel. You must also make sure the RTX64 Subsystem is running.

STEP 1: INSTALL THE VISUAL STUDIO REMOTE TOOLS

You must install the version of the Visual Studio remote tools on the target machine that is compatible with the version of Visual Studio installed on the host machine. View the Microsoft documentation to access the proper remote tools for your copy of Visual Studio:

1. View the Microsoft documentation to determine the version of Visual Studio remote tools that you need:

   https://docs.microsoft.com/en-us/visualstudio/debugger/remote-debugging

2. Download and install the remote tools on the target system.

STEP 2: CONFIGURE REMOTE DEBUGGING

Configure the remote debugging setup through the Configure Remote Debugging page in the RTX64 Control Panel.

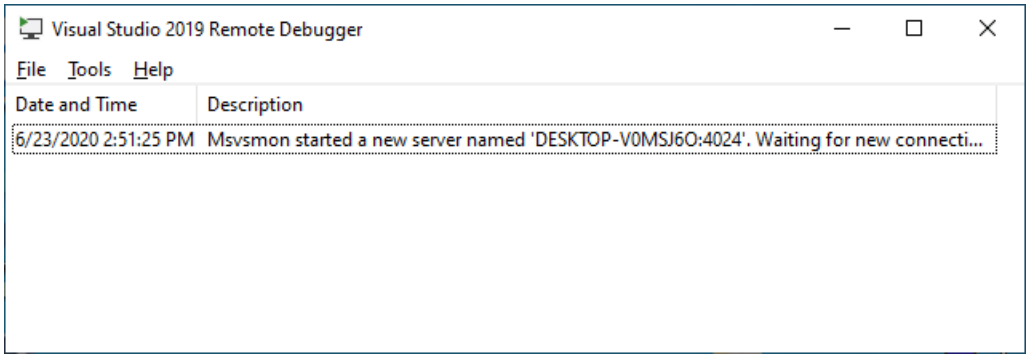
STEP 3: START THE SUBSYSTEM

Start the RTX64 Subsystem through the Control Panel, if it is not already running.

Attaching to a Process Running on the Target System

ON THE TARGET SYSTEM:

1. Launch the **Visual Studio Remote Debugger** from **Start > All Programs > Visual Studio 2022/2019/2017/2015**.



   - Optionally, change the default **TCP/IP port** by clicking **Tools > Options**.

2. Launch the **RTX64 Control Panel**

   - Under **Remote Debug**, click **Change** to open the **Configure remote debugging** window.
   - Click **Allow remote debugging connections on this computer**.
   - Verify the IP address of the target physical network appears as **<IP address:port msvsmon>**.

     Optionally, change to the default port 31094.

     > **NOTE:** The **msvsmon** port and the **RTSS remote debugging** port must be different.

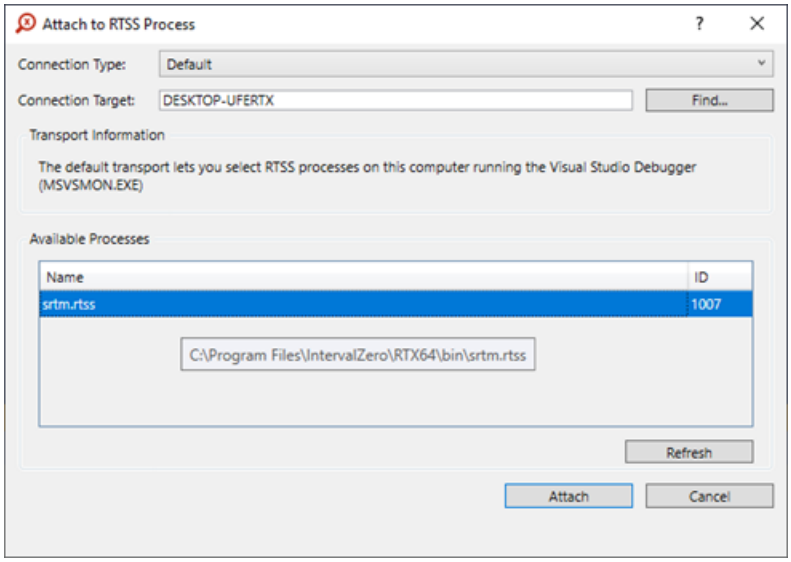
   - Restart the **RTX64 Control Panel**.

ON THE HOST SYSTEM:

Launch the version of Visual Studio that is compatible with the Visual Studio Remote Debugger on the target system.

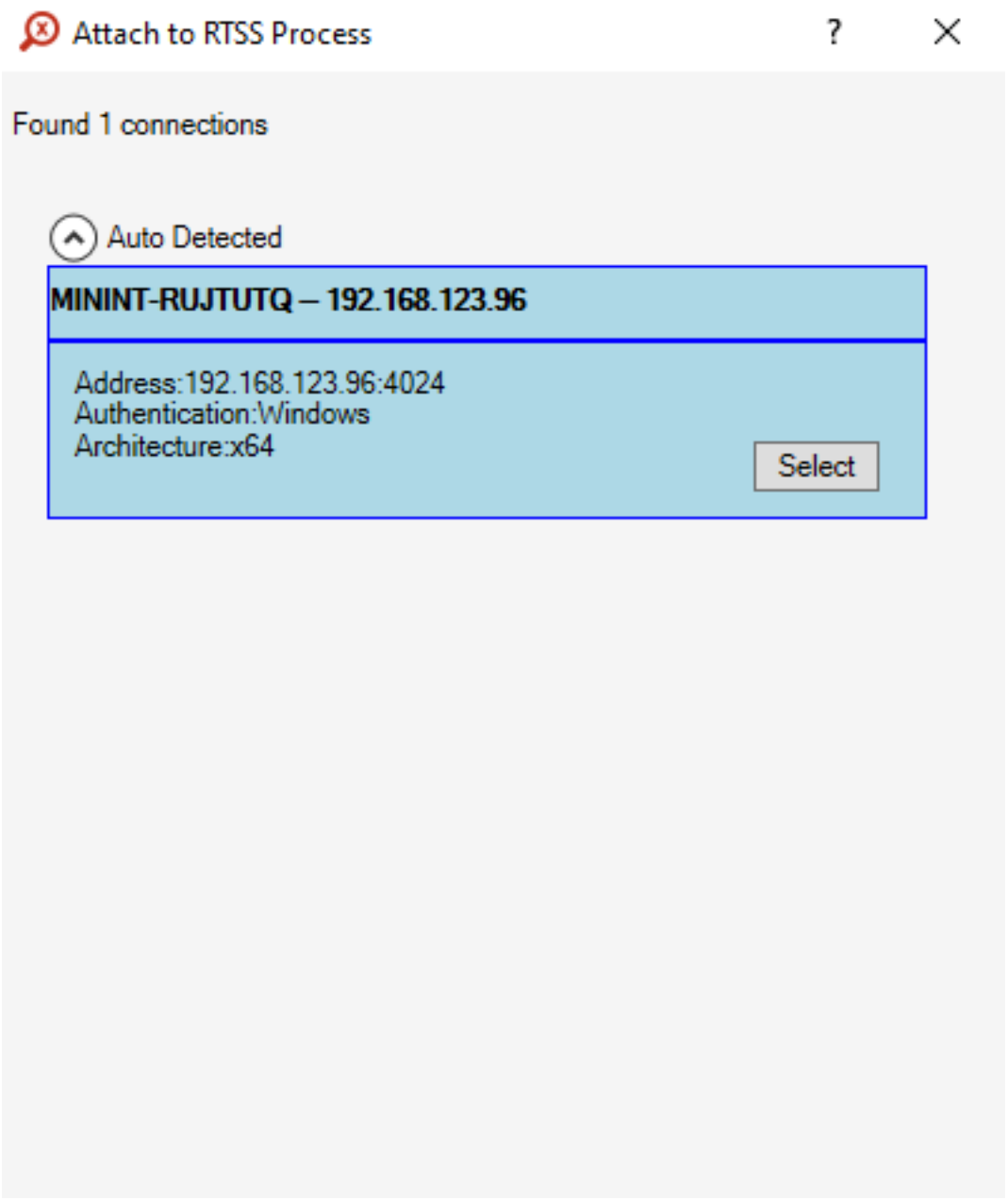
1. Under **Debug**, select **Attach to RTSS Process**.

   The *Attach to RTSS Process* dialog appears. This dialog displays the name and process ID for each running RTSS process. You can hover the mouse over the process name to display its path in a ToolTip. All processes to which you are currently attached, or that are currently being debugged, are grayed out.



2. For **Connection Target**, specify the remote computer using one of the following methods:

   - Enter the name of the remote computer in the text box and press Enter.
   - Enter the IP address of the remote computer in the text box and press Enter.
   - Click **Find**. This opens the *Remote Connections* dialog, which lists all the devices that are either on your local subnet or directly attached to your computer.



Select the computer or device you want to attach to, and then click **Select**. Use the Filter search box to search auto-detected connections by name or IP address.

NOTE: You may need to open UDP port 3702 on the target system to discover remote devices.

NOTE: If you can't connect to the remote computer name, you may need to open port 31094 for the RTX64 Remote debugger and Visual Studio Remote Debugger ports. Follow the instructions available from Microsoft to determine which ports you need to open:

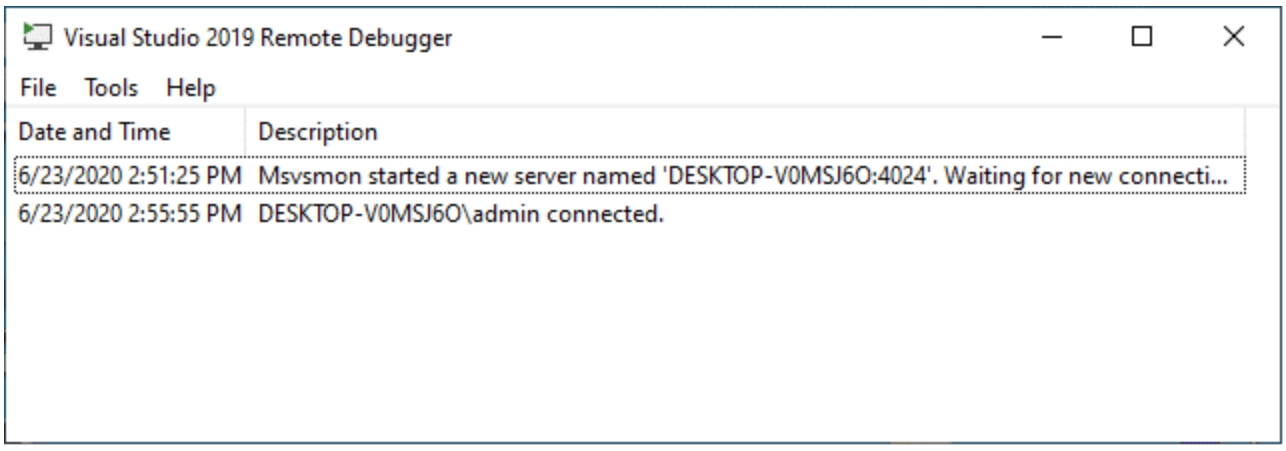
https://docs.microsoft.com/en-us/visualstudio/debugger/remote-debugger-port-assignments?view=vs-2019

3. In the *Attach to RTSS Process* dialog under **Available Processes**, select the process you want to attach to, and then click **Attach**.

NOTE: Debugging multiple processes in one instance of Visual Studio is not recommended.

NOTE: You cannot attach to a process that is either frozen by Watchdog timeout or already under debugger control. A process that is both frozen and under debugger control will appear as debugged. Processes in either scenario will appear grayed-out in the *Attach to RTSS Process* dialog. You can hover over a grayed-out process to view a tooltip with information on why the process cannot be attached.

You should now see the Host-Target connection in the *Visual Studio Remote Debugger* dialog on the target system.



Firewall Configuration

If a firewall is in use on the target system, you may need to configure it to allow certain ports for remote debugging. See the instructions from Microsoft here:

https://docs.microsoft.com/en-us/visualstudio/debugger/configure-the-windows-firewall-for-remote-debugging?view=vs-2022

Use command line New-NetFirewallRule (powershell) or netsh advfirewall firewall or manually add firewall rules using **Windows Defender Firewall with Advanced Security**.

For remote debugging, the following ports must be open on the remote computer:

Ports	Incoming/Outgoing	Protocol	Description
msvsmon	Incoming	TCP	For more information, see Visual Studio remote debugger port assignments.
RTSS remote debugging (31094)	Incoming/Outgoing	TCP	Required for RTX64 remote debugger discovery.
3702	Outgoing	UDP	(Optional) Required for remote debugger discovery.

Allow and configure the remote debugger through Windows Firewall or another firewall application:

1. **Launch Windows Defender Firewall** or another firewall application.
2. Select **Allow an app through Windows Firewall** or another firewall application.
3. If **RTX64 Remote Debugger** or **Visual Studio Remote Debugger** doesn't appear under **Allowed apps and features**, select **Change settings**, and then select **Allow another app**.

Troubleshooting

If you encounter issues connecting the host and target systems, try the following:

- Verify the RTX64 Subsystem is running on the target system.
- Verify *Allow remote debugging connections* is enabled in the RTX64 Control Panel and that it's listening on the correct IP address on the target system.
- Verify Visual Studio Remote Debugging Monitor (MSVSMON) is running on the target system.

- Make sure the target system name or the IP address in the *RTX64 Host-Target Connection* window matches the Computer Name found in the **Configuration Properties > Debugging** dialog on the host system.
- Verify the application you selected from *Attach to RTSS Process* dialog on the host system is running on the target system.
- Antivirus programs may cause problems with remote debugging. If you are unable to create a successful remote debugging session, check to see if an Antivirus program is installed and active on the target system.

RELATED TOPICS:

- Debugging with Visual Studio

Supported Visual Studio Breakpoint Features

RTX64 supports the following breakpoint features in Microsoft Visual Studio:

- Dependent Breakpoints
- Temporary Breakpoints
- Force Run To Cursor
- Drag and Drop Breakpoints
- TracePoints
- Hard-Coded Breakpoints
- Conditional Breakpoints

Dependent Breakpoints

A dependent breakpoint only executes if another breakpoint is first hit.

Available in Visual Studio 2022

TO CREATE A DEPENDENT BREAKPOINT:

1. Click on the left margin of the line you want the first breakpoint to occur.
2. Click on the left margin of the line you want the dependent breakpoint to occur.
3. Hover over the dependent breakpoint and click the **Settings** icon.
4. Select the **Only enable when the following breakpoint is hit** check box.
5. Click on the drop down menu and select the first breakpoint.

Temporary Breakpoints

A temporary breakpoint is executed once and deletes itself upon execution.

Available in Visual Studio 2022

TO CREATE A TEMPORARY BREAKPOINT:

1. Click on the left margin of the line where you want the breakpoint to occur.
2. Hover over the breakpoint and click the **Settings** icon.
3. Select the **Remove breakpoint once hit** check box.

Force Run To Cursor

Force Run To Cursor disables all breakpoints until the program hits the line specified as *Force Run To Cursor*.

Available in Visual Studio 2022

TO ENABLE *FORCE RUN TO CURSOR*:

1. Right-click the line of code you want to specify as *Force Run To Cursor*.
2. Select **Force Run To Cursor**.

Drag and Drop Breakpoints

Breakpoints can be dragged into new locations in the code and while retaining their current settings.

Available in Visual Studio 2022

TracePoints

TracePoints allow you to log information to the Output window under configurable conditions without modifying or stopping your code. This feature does not create a breakpoint in the code.

Available in Visual Studio 2015/2017/2019/2022

TO CREATE A TRACEPOINT:

1. Create a breakpoint by clicking on the left margin of the line you want the tracepoint to occur.
2. Hover over the breakpoint and click the **Settings** icon.
3. Select the **Actions** check box.
4. Type the desired message into the **Show a message in the Output Window** text box.

   The breakpoint icon will appear as a diamond to indicate it is now a tracepoint.

NOTE: For more information on the features above, see the following link: https://docs.microsoft.com/en-us/visualstudio/debugger/using-breakpoints?view=vs-2022

Hard-coded Breakpoints

A hard-coded breakpoint is written into program code. It cannot be changed during runtime.

Available in Visual Studio 2015/2017/2019/2022

Conditional Breakpoints

A conditional breakpoint executes when a pre-configured condition is occurs..

Available in Visual Studio 2015/2017/2019/2022

TO CREATE A CONDITIONAL BREAKPOINT:

1. Create a breakpoint by clicking on the left margin of the line where you want the breakpoint to occur.
2. Hover over the breakpoint and click the **Settings** icon.
3. Select the **Conditions** check box.
4. Configure the conditions you would like to set.

RELATED TOPICS:

- Supported Visual Studio Versions
- Debugging RTSS Applications
- Debugging RTDLLs
- Setting up Visual Studio for RTX64 Wizards
- Debugging Multiple Processes Simultaneously

RtQuerySystemPerfCaps

RtQuerySystemPerfCaps queries the performance capability of each logical processor for the specified ClassID.

Syntax

```
BOOL RtQuerySystemPerfCaps(
    ULONG ClassID,
    ULONG  SystemPerfCaps[]
);
```

Parameters

ClassID

Specifies the Class ID of the performance capabilities to be queried. Class ID (0, 1, 2, …) represents a software thread's characteristic; i.e., the instruction set the thread executes on the logical processor.

SystemPerfCaps[]

A ULONG array to store current performance capabilities of Class ID on each active logical processor, including Windows logical processors and RTSS logical processors.

Return Value

Returns TRUE if the function succeeds. Returns FALSE if the function fails. **GetLastError** returns *ERROR_NOT_SUPPORTED* if Intel Thread Director is not supported, or *ERROR_INVALID_PARAMETER* if there are invalid parameters.

Remarks

Thread Class IDs (0, 1, 2, ...) represent a software thread's characteristics; i.e., the assembly instruction set (e.g., AVX2) the thread executes. With Intel Thread Director Technology (IDT), the logical processor feedbacks the Class ID of the thread running on it with enough history.

RTX64 does not dynamically migrate RTSS threads during thread scheduling. It is recommended that RTSS processes call **RtQuerySystemPerfCaps** for each ClassID to identify the performance ideal processor/processors. Since **SetThreadIdealProcessor** or **SetThreadAffinityMask** cannot migrate Interrupt Service Thread (IST), Timer Service Thread (TST), and NPX used thread at run-time, you may need to affinity those threads to their performance ideal processors at create time if you know the instruction set those threads use.

On performance hybrid architectures, there are performance cores (P-Core) and efficiency cores (E-Core). To optimize RTX64 performance on a hybrid architecture, you may want to run the HybridInfo utility, available for download from the Support Site, before you assign processors to Windows and RTSS using the RTX64 Activation and Configuration utility. The HybridInfo utility displays each core type, performance capability, and thread class ID for different sets of assembly instructions. The Class IDs for instruction sets are as follows:

Instruction Set	Class ID
Default	0
Integer Registers	1
AVX/AVX2	2
Pause	3

Requirements

Minimum Supported Version	RTX64 4.3
Header	RtssApi.h
Library	Rtx_Rtss.lib (RTSS)

SEE ALSO:

SetThreadIdealProcessor

SetThreadAffinityMask