Getting Started Guide for the Target Access Plugin

All information is subject to change without notice. IAR Systems assumes no responsibility for errors and shall not be liable for any damage or expenses.

Content

Description

The Target Access plugin is a generic server/client solution which allows third-party applications to read from and write to memory locations on the target device via IAR Embedded Workbench through a UDP connection.

In addition, for IAR Embedded Workbench for Arm, it also provides functions for monitoring ITM (Instrumentation Trace Macrocell) events.

Start the server part by activating Target Access Server from the C-SPY plugin setup in IAR Embedded Workbench or from `cspybat`. The plugin starts a UDP server that listens for commands.

The client is composed of the Target Access Client SDK (`TargetAccessClientSDK.dll`), which is a library that exports the target commands through a set of C functions. An external application can communicate with the server through this library, which is described in detail in this guide.

Directory structure

The plugin is organized in the following directories:

Directory	Description
`common\plugins\TargetAccessServer\`	The Target Access Server C-SPY plugin
`common\bin\TargetAccessClientSDK.dll`	The Target Access Client SDK (.dll)
`[`*`target`*`]\src\TargetAccessPlugin\lib\`	The Target Access Client SDK (.lib)
`[`*`target`*`]\src\TargetAccessPlugin\inc\`	The Target Access Client SDK (.h)

Enabling the Target Access Server plugin

The server plugin is included in the IAR Embedded Workbench distribution. To enable the plugin, choose:

Project>Options>Debugger>Plugins>Target Access Server

Using the Target Access Client SDK

To communicate with the server, a C language client SDK library is provided. This library contains functions for connecting to the server plugin and for reading from and writing to memory on the target device.

The library consists of the following files:

- TargetAccessClientSDK.dll
- TargetAccessClientSDK.h
- TargetAccessClientErrorCodes.h
- TargetAccessClientSDK.lib

To use the SDK in your application:

1. Include TargetAccessClientSDK.h (which in turn includes TargetAccessClientErrorCodes.h)
2. Link your application with TargetAccessClientSDK.lib
3. At runtime, make sure that TargetAccessClientSDK.dll is located in the same directory as the executable file (alternatively, in the system path described by the Path environment variable)

Types

Generic types

```
typedef int64_t TargetAccessAddr;
```

This typedef is used when an address is required as parameter to a function.

Types specific to IAR Embedded Workbench for Arm

```
struct ITMEvent
{
 uint8_t portId;
 uint64_t timeStamp;
 uint32_t dataLength;
 uint8_t data[4];
};
```

When an ITM event has been recorded, this struct is populated with the port ID and the applicable data payload. The portId member specifies the ITM port with the range [0…31]. Note that ITM ports 0–19 are reserved by IAR Embedded Workbench, which can result in undefined behavior if those ports are used by client applications.

The data can be 1, 2, or 4 bytes long. The actual length is stored in the dataLength member. The timeStamp member shows the timestamp for the event in nanoseconds, as reported by the debug probe.

```
struct ITMListenerStatus
{
  uint8_t listening;
  uint32_t channels;
  uint32_t bufferCapacity;
  uint32_t bufferSize;
};
```

The ITMListenerStatus struct stores the status information collected when calling the TargetAccessGetItmListenerStatus() function. The listening member is set to

1 when ITM listening is active, otherwise to 0. The channels and bufferCapacity members are the bit mask and buffer capacity, respectively, set when the TargetAccessConfigureItmListener() function is called. If that function has not yet been called, the default values of those members are 0. The bufferSize member shows the current buffer size. When bufferSize == bufferCapacity, it can be assumed that ITM events are being dropped.

Functions

Generic functions

Most of the following C functions all return 0 if successful, otherwise an error code. For more information about the latest error, call the TargetAccessGetLastErrorMsg().

```
int TargetAccessInitialize(const char *serverName)
```

This function initializes the SDK and must be called before any of the other functions are used. serverName is the computer to connect to, for example, localhost or 127.0.0.1 to connect to the local computer.

```
int TargetAccessShutdown()
```

This function must be called when you no longer need to access the Target Access interface.

```
int TargetAccessGetClientProtocolVersion()
```

Returns the client version.

```
int TargetAccessGetServerProtocolVersion()
```

Returns the server version.

```
int TargetAccessReadTargetMemoryZone(const char *zoneName,
TargetAccessAddr address, unsigned char *buffer, uint16_t numBytes)
```

Reads numBytes bytes of memory at address in the memory zone zoneName to buffer.

```
int TargetAccessReadTargetMemory(TargetAccessAddr address, unsigned char
*buffer, uint16_t numBytes)
```

Reads numBytes bytes of memory at address in the default memory zone to buffer.

```
int TargetAccessWriteTargetMemoryZone(const char *zoneName,
TargetAccessAddr address, const unsigned char *buffer, uint16_t numBytes)
```

Writes numBytes bytes of memory at address in the memory zone zoneName using data from buffer.

```
int TargetAccessWriteTargetMemory(TargetAccessAddr address, const unsigned
char *buffer, uint16_t numBytes)
```

Writes numBytes bytes of memory at address in the default memory zone using data from buffer.

```
size_t TargetAccessGetLastErrorMsg(char *errorMsg, size_t bufSize)
```

Writes a null-terminated text string with a description of the last error to `errorMsg`, which is a `char` buffer with the size `bufSize`. The function returns `0` on success. If the provided buffer size was too small to hold the error message, the required buffer size is returned.

Functions specific to IAR Embedded Workbench for Arm

```
int TargetAccessConfigureItmListener(uint32_t channels, uint32_t
bufferCapacity)
```

Configures the ITM event monitoring. The `channels` parameter is a 32-bit mask that can be used for filtering which ITM events to be kept. This is in addition to the **ITM Stimulus Ports** settings found in the **SWO Configuration** dialog box in C-SPY, which means that to receive the ITM event at a specific port, it is important that you selected it both in the **SWO Configuration** dialog box and as a set bit in the `channels` parameter.

The `bufferCapacity` parameter specifies how many events will be saved before they are dropped.

The default values of `channels` and `bufferCapacity` are `0`.

```
int TargetAccessGetItmListenerStatus(ITMListenerStatus *status)
```

Fills the `status` parameter with information described for the `ITMListenerStatus struct` in the section *Types*.

```
int TargetAccessStartItmListener()
```

This function starts the ITM event monitoring.

```
int TargetAccessStopItmListener()
```

This function stops the ITM event monitoring.

```
int TargetAccessReadItmEvent(ITMEvent *itmEvent, bool *isValid)
```

Once the ITM event monitoring has been started, use this function to receive events. If no event has been collected, the `isValid` parameter is set to `false`. The event is stored in `itmEvent`, described in the section *Types*.

Example applications

Accessing Target Memory

This code example demonstrates how to read two bytes of memory at address `0x10` on the target device.

```
char errorMsg[256] = {0};
int res = 0;

// Initialize client
res = TargetAccessInitialize("127.0.0.1");
if (res != TARGET_ACCESS_STATUS_OK)
{
  TargetAccessGetLastErrorMsg(errorMsg, sizeof(errorMsg));
  fprintf(stderr, "Error: %s\n", errorMsg);
  TargetAccessShutdown();
  return 1;
}

// Read bytes from default memory zone
TargetAccessAddr addr = 0x10;
unsigned char buffer[2];

res = TargetAccessReadTargetMemory(addr, buffer, sizeof(buffer));
if (res != TARGET_ACCESS_STATUS_OK)
{
  TargetAccessGetLastErrorMsg(errorMsg, sizeof(errorMsg));
  fprintf(stderr, "Error: %s\n", errorMsg);

  TargetAccessShutdown();
  return 1;
}

// Shut down client
res = TargetAccessShutdown();
if (res != TARGET_ACCESS_STATUS_OK)
{
  TargetAccessGetLastErrorMsg(errorMsg, sizeof(errorMsg));
  fprintf(stderr, "Error: %s\n", errorMsg);
}
```

Implementing ITM Listening (only for IAR Embedded Workbench for Arm)

This example shows a complete application (written in C++) that sets up a console-based logger of ITM events. The next section shows an example of how the application can be used.

First, prepare the application with some includes and define an exception class that will be used by the application. The `ConsoleCloseHandler()` function is a handler required by Windows to allow for a clean shutdown of the network connection when the user closes the console.

```
#include "TargetAccessClientSDK.h"

#include <iostream>
#include <iomanip>
#include <string>
#include <thread>
#include <chrono>

#ifdef _WIN32
#include <Windows.h>
BOOL WINAPI ConsoleCloseHandler(DWORD dwCtrlType)
{
  if (dwCtrlType == CTRL_CLOSE_EVENT)
  {
    std::cout << "TargetAccessClientITMdemo is shutting down" <<
      std::setfill(' ') << std::setw(50) << "" << std::endl;
    ::TargetAccessShutdown();
  }
  return TRUE;
}
#endif

class DemoException : public std::exception
{
public:
  DemoException()
  {
  }
};
```

The next step demonstrates the use of the `TargetAccessGetItmListenerStatus()` function:

```
void ReportITMStatus()
{
  ITMListenerStatus status;
  status.listening = 0;
  status.channels = 0;
  status.bufferCapacity = 0;
  status.bufferSize = 0;
  ::TargetAccessGetItmListenerStatus(&status);
  std::cout << "ITM Listener: " << (status.listening != 0 ? "on" : "off")
    << " ";
  std::cout << "Channels: 0x" << std::hex << std::setfill('0') <<
    std::setw(8) << status.channels << std::dec << " ";
  std::cout << "Buffer capacity: " << status.bufferCapacity << " ";
  std::cout << "Buffer size: " << status.bufferSize << "\r";
}
```

This demonstrates how errors can be handled:

```
void ReportError()
{
  std::string errorMsg;
  errorMsg.resize(512);
  ::TargetAccessGetLastErrorMsg(const_cast<char *>(errorMsg.data()),
    errorMsg.size());
  std::cout << "Error: " << errorMsg << std::endl;
  std::cout << "Client protocol version: " <<
    ::TargetAccessGetClientProtocolVersion() << std::endl;
  std::cout << "Server protocol version: " <<
    ::TargetAccessGetServerProtocolVersion() << std::endl;
}
```

Next, implement the main event loop. It starts by setting the ITM channels bit mask to `0xffffffff` and the buffer capacity to 1000 events. Note that the `duration` variable switches between 0 and 1 seconds depending on whether the ITM buffer is empty or not. If the received `ITMEvent` block is invalid (the `isValid` argument is `false`), the wait time for the next poll is set to 1 second.

```
int RunItmLogging()
{
  if (::TargetAccessConfigureItmListener(0xffffffff, 1000) != 0)
    throw DemoException();

  if (::TargetAccessStartItmListener() != 0)
    throw DemoException();

  auto defaultDuration = std::chrono::seconds(1);
  auto duration = defaultDuration;
  bool isValid = false;

  while (1)
  {
    ITMEvent itmEvent;
    if (::TargetAccessReadItmEvent(&itmEvent, &isValid) !=
        TARGET_ACCESS_STATUS_OK)
      throw DemoException();
    if (isValid)
    {
      std::cout << "[" << std::hex << itmEvent.timeStamp << std::dec <<
        "] ITM port " << (int)itmEvent.portId << ": ";
      uint32_t value = 0;
      switch (itmEvent.dataLength)
      {
      case 1:
        value = itmEvent.data[0];
        break;
      case 2:
        value = *reinterpret_cast<uint16_t *>(itmEvent.data);
        break;
      case 4:
        value = *reinterpret_cast<uint32_t *>(itmEvent.data);
        break;
      }
      std::cout << std::hex << std::setfill('0') << std::setw(8) <<
        value << std::dec;
      std::cout << " (" << itmEvent.dataLength << " byte" <<
        (itmEvent.dataLength > 1 ? "s" : "") << " length)";
      std::cout << std::setw(50) << std::setfill(' ') << "" << std::endl;
      duration = std::chrono::seconds(0); // Keep reading events until the
                                          // buffer is empty
    }
    else
    {
      duration = defaultDuration; // Pause with 1 second delays when no
                                  // data is recorded
      ::ReportITMStatus();
    }

    std::this_thread::sleep_for(duration);
  }
  return 0;
}
```

Finally, the entry function for the program sets up communication to the server and handles errors. It also performs a clean shutdown before exiting:

```
int main(int argc, const char **argv)
{
#ifdef _WIN32
  ::SetConsoleCtrlHandler(ConsoleCloseHandler, TRUE);
#endif

  char *portStr = ::getenv("TARGET_ACCESS_PORT");
  if (portStr == nullptr)
    portStr = "9931";
  int port = std::stoi(portStr);

  try
  {
    if (argc > 1)
      port = std::stoi(argv[1]);

    // Initialize client
    if (::TargetAccessInitializeWithPort("localhost", port) !=
        TARGET_ACCESS_STATUS_OK)
      throw DemoException();
    ::RunItmLogging();
  }
  catch (const DemoException &)
  {
    ::ReportError();
  }

  ::TargetAccessShutdown();
  system("pause");
  return 0;
}
```

Target application example (only for IAR Embedded Workbench for Arm)

The `arm_itm.h` header file (located in `arm\inc\c`) contains predefined preprocessor macros for setting ITM events. The following code shows a simple example of how these macros can be used in combination with the `TargetAccessClientITMdemo` application introduced in the previous section.

Although only two `ITM_EVENT` macros are called, there are actually three ITM events set in the `while` loop below. The reason is that the `ITM_EVENT32_WITH_PC` macro also sets the current value of the `PC` register at ITM port 5. The `arm_itm.h` header file contains macros for 8, 16, and 32 bits; `ITM_EVENT8`, `ITM_EVENT16`, and `ITM_EVENT32`, respectively. Each of these has a corresponding macro that sets the current `PC` value (`ITM_EVENT8_WITH_PC`, `ITM_EVENT16_WITH_PC` and `ITM_EVENT32_WITH_PC`).

```
#include <arm_itm.h>
#include <stdint.h>

void main()
{
  ITM_EVENT8(20, 0);
  ITM_EVENT32(21, 0);
  uint8_t value = 0;
  uint32_t sum = 0;
  while (1)
  {
    if (value++ % 10 == 0)
    {
      sum += value;
      ITM_EVENT8(20, value);
      ITM_EVENT32_WITH_PC(21, sum);
      for (int i = 10000; i > 0; --i); /* Delay */
    }
  }
}
```