LAUTERBACH
DEVELOPMENT TOOLS

Application Note for FDX

Application Note for FDX

Application Note for FDX

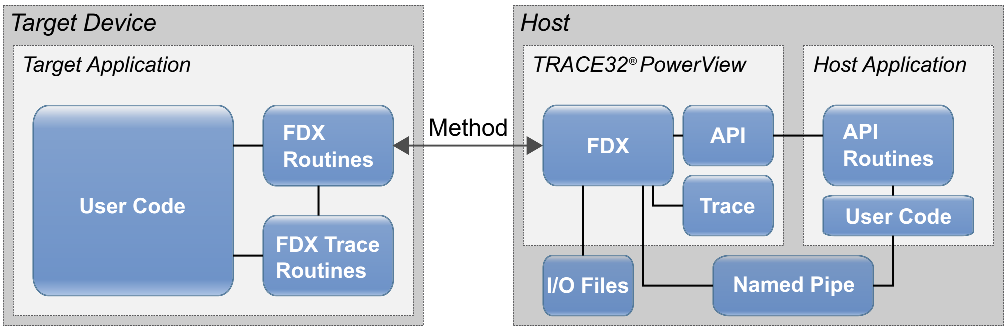
Version 06-Jun-2024

General Function

The Fast Data Exchange (FDX) enables transferring universal data between the target and the host. The protocol implementation on target side is included in the target application. The source code (C) is provided by LAUTERBACH. On the host side the transmitted data can be processed by a user application communicating with the TRACE32 Application Interface or through Named Pipes. In a non-interactive mode TRACE32 can also read or write normal files as a general data source and sink.

A special application of the FDX is the FDX software trace. Through the trace application interface the target application can send trace information to the host. TRACE32 is capable to interpret the FDX data stream and handle it as ordinary trace information device. This document covers only the target application interface and FDX specific configuration. For further information about the trace functions see 'Trace' in 'General Commands Reference Guide T'.

The basic packet transport method differs dependent on the target. TRACE32 supports memory mapped buffered transfer through dual port memory access or normal access at breakpoints or spot breakpoints. Some target devices support a Debug Communication Channel (DCC), which can be used to transfer FDX data in real-time.



Restrictions

FDX requires instrumentation of the target application in order to reserve a memory output buffer and to send/receive FDX data. The transmission speed depends on the used channel (dual port memory access, DCC, normal memory access at breakpoints). When using breakpoints, the real-time behavior of the target application is influenced.

Related Tutorials

For a video tutorial about FDX trace, visit:
support.lauterbach.com/kb/articles/trace32-fdx-trace

Contacting Support

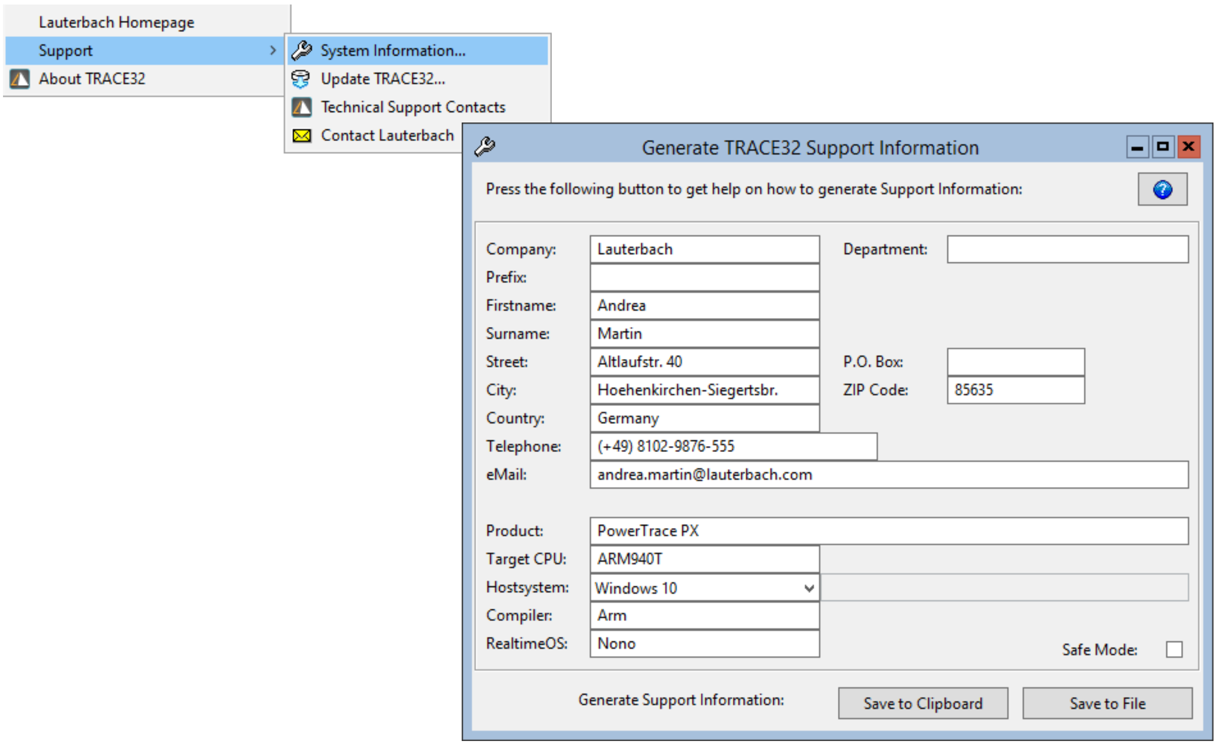
Use the Lauterbach Support Center: https://support.lauterbach.com

- To contact your local TRACE32 support team directly.
- To register and submit a support ticket to the TRACE32 global center.
- To log in and manage your support tickets.
- To benefit from the TRACE32 knowledgebase (FAQs, technical articles, tutorial videos) and our tips & tricks around debugging.

Or send an email in the traditional way to support@lauterbach.com.

Be sure to include detailed system information about your TRACE32 configuration.

1. To generate a system information report, choose **TRACE32** > **Help** > **Support** > **Systeminfo**.



Press the following button to get help on how to generate Support Information:

Company: Lauterbach
Department:
Prefix:
Firstname: Andrea
Surname: Martin
Street: Altlaufstr. 40
P.O. Box:
City: Hoehenkirchen-Siegertsbr.
ZIP Code: 85635
Country: Germany
Telephone: (+49) 8102-9876-555
eMail: andrea.martin@lauterbach.com

Product: PowerTrace PX
Target CPU: ARM940T
Hostsystem: Windows 10
Compiler: Arm
RealtimeOS: Nono
Safe Mode: ☐

Generate Support Information:

NOTE:	Please help to speed up processing of your support request. By filling out the system information form completely and with correct data, you minimize the number of additional questions and clarification request e-mails we need to resolve your problem.

2. Preferred: click **Save to File**, and send the system information as an attachment to your e-mail.

3. Click **Save to Clipboard**, and then paste the system information into your e-mail.

Target Application Interface for General Data Transfer

The target application FDX interface is provided in the files *t32fdx.h* and *t32fdx.c* which can usually be found under the ~~/demo/<processor_family>/fdx directory. The application can open multiple FDX channels. Every channel must have it's own data type which can be derived by using the macro *T32_Fdx_DefineChannel*. One channel can transfer data only in one direction. Usually two channels are defined and created. One for output and the other for input direction. Every channel defines a fifo buffer which must be more than two times larger than the block size of the maximum transferred data blocks.

```
#define FDXTEST_BUFSIZE 0x1000
T32_Fdx_DefineChannel(FdxTestSendBuffer,FDXTEST_BUFSIZE);
T32_Fdx_DefineChannel(FdxTestReceiveBuffer,FDXTEST_BUFSIZE);
```

When the channel data types are defined, they can be used and initialized by the macro *T32_Fdx_InitChannel*.

```
T32_Fdx_InitChannel(FdxTestSendBuffer);
T32_Fdx_InitChannel(FdxTestReceiveBuffer);
```

To enable a channel for transferring data use the macro *T32_Fdx_EnableChannel*.

```
T32_Fdx_EnableChannel(FdxTestSendBuffer);
T32_Fdx_EnableChannel(FdxTestReceiveBuffer);
```

With *T32_Fdx_Send* data can be copied into the transmit buffer. Depending to the selected method the data can be transmitted to the host in background. When the transmit buffer is full, the routine will be blocking to send the data stored in the buffer to the host until the buffer can hold the new data. The macro *T32_FDX_DATATYPE* defines the data type of one item in buf. When the target only allows word accesses, *T32_FDX_DATATYPE* should be set to *unsigned short*. The host program *fdxdatatype* should be the same type. The result of *T32_Fdx_Send* is size, if the operation finished successfully. A result of -1 indicates that an error has occurred.

```
Declaration
int T32_Fdx_Send(void * channel, void * buf, int size)
Example
unsigned char* buf= “Message”;
int len = strlen(buf)
T32_Fdx_Send(&FdxTestSendBuffer, buf, len+1);
```

There is a non blocking variant of *T32_Fdx_Send* named *T32_Fdx_SendPoll*. If the target and host ring buffer is full the routine returns zero and the T32_Fdx_SendPoll has to be called again later.

```
Declaration
int T32_Fdx_SendPoll(void * channel, void * buf, int size)
Example
int result;
result = T32_Fdx_SendPoll(&FdxTestSendBuffer, buf, len+1);
```

To flush all pending send operations use the *T32_Fdx_Poll* routine combined with checking by macro *T32_Fdx_Pending*.

```
Declaration
void T32_Fdx_Poll(void)
Example
while (T32_Fdx_Pending(&FdxTestSendBuffer))
   T32_Fdx_Poll();
```

To receive data from the host use *T32_Fdx_Receive*. The routine copies the next entry in the receive ring buffer. If there is no data left in the buffer, the routine will wait for receiving data from the host.

```
Declaration
int T32_Fdx_Receive(void * channel, void * buf, int size)
Example
len = T32_Fdx_Receive(&FdxTestReceiveBuffer, buf, sizeof(buf));
```

T32_Fdx_ReceivePoll can be used to copy data from the receive fifo buffer without blocking. If no data is available the communication port is polled one time at least.

```
Declaration
int T32_Fdx_ReceivePoll(void * channel, void * buf, int size)
Example
while (!(len = T32_Fdx_ReceivePoll(&FdxTestReceiveBuffer, buf,
sizeof(buf))));
```

When *len* becomes lower or equal than zero a communication error has occurred.

```
if (len <= 0)

     printf(“Communication Error”);

else

     printf(“Received Message %s”,buf);
```

Host Application Interface for General Data Transfer

To handle FDX data a host application is required. This application can communicate either over the TRACE32 Remote API or directly over Named Pipes. The use of Named Pipes is explained in section **“Configuration of TRACE32 for General Data Transfer”**, page 10. This section covers the adaptations of the standard T32 API to FDX communication. An example in *fdxhost.c* can be found in the TRACE32 demo directory. To build a host application hlinknet.c, hremote.c and t32.h are needed. These files can be used to create executables for all operating systems on which TRACE32 is available. A general description to use the TRACE32 Remote API can be found in **“API for Remote Control and JTAG Access in C”** (api_remote.pdf).

After initialization of the API the handles of the communication channels are created by a call of *T32_Fdx_Open*. The first parameter must be equal to the symbol name of the channel structure in the target application. The second parameter indicates read or write direction and must comply with the direction of the channels on the target. The result of *T32_Fdx_Open* is a handle to the FDX channel. If the handle is -1 the function has failed.

```
Declaration
int T32_Fdx_Open(char * name, char * mode)
Example
int fdxin, fdxout;
fdxin = T32_Fdx_Open("FdxTestSendBuffer","r");
fdxout = T32_Fdx_Open("FdxTestReceiveBuffer","w")
```

To receive data from the target channel *T32_Fdx_Receive* can be used. *fdxdatatype* should be the same type as *T32_FDX_DATATYPE* in the target program. The third and fourth parameters are the size of *fdxdatatype* and the amount of data in pieces of *fdxdatatype* that the destination buffer can hold. The function result is the length of the received block. This length is equal to the length parameter of the *T32_Fdx_Send* routine in the target program. When the result is lower or equal to zero an error has occurred.

```
Declaration
int T32_Fdx_Receive(int channel, void * data, int width, int maxsize)
Example
fdxdatatype     buffer[4096];
len = T32_Fdx_Receive(fdxin, buffer, sizeof(buffer[0]), sizeof(buffer) /
sizeof(buffer[0]));
```

T32_Fdx_ReceivePoll is the non blocking equivalent to *T32_Fdx_Receive*. A function result of zero indicates that there is no data available. *T32_Fdx_Receive* is increasing the CPU load of the host dramatically, because it is polling in a loop. If you still want to use TRACE32 in parallel, use the non blocking routine combined with wait instructions. The example below lets TRACE32 be usable, but decreases the channel latency with 50 ms in worst case.

```
Declaration
int T32_Fdx_ReceivePoll(int channel, void * data, int width, int maxsize)
Example
while (!(len = T32_Fdx_ReceivePoll(fdxin, buffer, sizeof(buffer[0]),
sizeof(buffer) / sizeof(buffer[0])))
 sleep(50);
```

With T32_Fdx_Send data can be sent to the target program. The parameters are equal to the *T32_Fdx_Receive* routine, except the fourth parameter indicates the amount of data that has to be transmitted. If the routine does not fail, the result will be *size*.

```
Declaration
int T32_Fdx_Send(int channel, void * data, int width, int size)
Example
int result;
const char* strMessage = “Message”;
fdxdatatype* buffer = (fdxdatatype*) strMessage;
int len = strlen(strMessage)+1;
result = T32_Fdx_Send(fdxout, buffer, sizeof(buffer[0]), len
/sizeof(buffer[0]));
```

When the host ring buffer is full, *T32_Fdx_Send* may increase the CPU load similar to *T32_Fdx_Receive*. Please use *T32_Fdx_SendPoll* combined with *sleep* statements to reduce this effect. The result of T32_Fdx_SendPoll is zero, in case the packet has not been sent and *T32_Fdx_SendPoll* has to be called again.

Configuration of TRACE32 for General Data Transfer

Enable Remote API.

If the TRACE32 Remote API is used to communicate between the host application and TRACE32, the configuration file has to be modified to enable the network interface. Please add therefore the lines to the *config.t32* file. If the port is in use, the port number can be set to another value. *PACKLEN* must not be higher than the system value otherwise the communication fails.

```
RCL=NETASSIST
PACKLEN=1024
PORT=20000
```

Initialize FDX Communication

Use a PRACTICE script to initialize FDX communication. The startup script depends on the basic packet transport method of FDX. For DCC based communication the example below can be used.

```
FDX.DISable
FDX.METHOD DCC8
FDX.OutChannel
FDX.InChannel
```

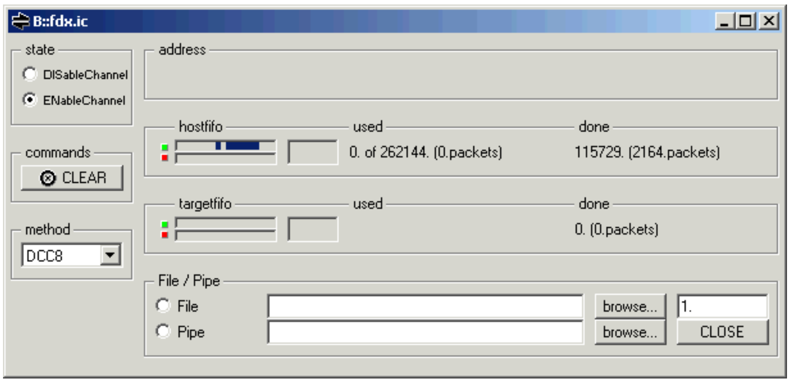
The line *FDX.METHOD DCC8* specifies the basic packet transport method. Please read the command reference for **FDX.METHOD** command for more details. All DCC based communications don't need any additional parameter for InChannel and OutChannel.

Memory access based FDX uses a different configuration.

```
FDX.DISable
FDX.METHOD BUFFERE
FDX.OutChannel FdxSendBuffer
FDX.InChannel FdxReceiveBuffer
```

All memory access based FDX methods need the symbolic name of the target program channels as parameter for OutChannel and InChannel. **Before FDX InChannel and OutChannel are set the target should have initialized the passed structures, otherwise the communication fails.**

InChannel and OutChannel open a dialog, where statistical data is displayed and channel specific settings can be done. The figure below shows opened the window for an InChannel. Due to window settings for OutChannels are a subset, it will not be explained here.



B::fdx.ic

state
DISableChannel
[✓ ENableChannel]

address

commands
CLEAR

method
DCC8

hostfifo — used: 0. of 262144. (0.packets) — done: 115729. (2164.packets)

targetfifo — used — done: 0. (0.packets)

File / Pipe
File — browse... — 1.
Pipe — browse... — CLOSE

DisableChannel, **EnableChannel**, **Clear**, **Method** and **Close** match to the *FDX.* command line functions described in the Command Reference Manual.

The two panels in the middle show statistical data about the communication channel. There exists a FIFO buffer for each channel on the host side and on the target side in the target application. The green labeled fields indicate activity on the channel in the last seconds. The red labled fields show stalls during the last time. In the DCC based communication mode there is no information available about the target fifo, because TRACE32 has no random access to the target buffer structure.

In the *File/Pipe* panel a file or Named Pipe can be selected as general data input and output stream. Because the FDX protocol is packet oriented, for packet size has to be defined for these ordinary streams. This can be done in the field right to the browse button of the *File* text field.

The *address* panel displays the target memory address of the target buffer structure.

FDX Software Trace

The FDX software trace uses the same interface files as needed for general data transfer, because the trace interface routines use the FDX target interface routines.

Target Application Interface for FDX Trace

To initialize the trace, the target application calls *T32_Fdx_TraceInit*.

```
Declaration
void T32_Fdx_TraceInit(void);
Example
T32_Fdx_TraceInit();
```

A call to *T32_Fdx_TraceData* copies the trace information of a single trace record into the output buffer. The declaration of this function is:

```
Declaration
void T32_Fdx_TraceData(int cycletype, void* address, unsigned long data)
```

A trace record consists of the *cycletype*, data address and the data itself. The value for *cycletype* includes the information of the data size in bytes and the bus access type (read, write, fetch).

Valid values for *cycletype* are the result of the expression (data byte width | (access type << 4)), with 0 <= data byte width <= 15 and accesstype may be 0x1 for fetch, 0x2 for read and 0x3 for write accesses, e.g. a 32 Bit write access has a *cycletype* of 0x4 | (0x3 << 4) which equaled 0x34.

Examples:

```
Example
//Traces a fetch of function func8 (datasize == 0 bytes)
T32_Fdx_TraceData(0x10, &func8, 0);

//Traces a write access to a 32Bit variable mcount
T32_Fdx_TraceData(0x34, &mcount, mcount);
```

If the output buffer of the trace channel is full, *T32_Fdx_TraceData* will send data from the buffer to the host. To send all trace data to the host at a certain point of execution use the *T32_Fdx_Poll* macro.

```
Example
while (T32_Fdx_Pending(&FdxTraceSendBuffer))
   T32_Fdx_Poll();
```

The trace transmit routines support two different modes. In the compress mode, the trace records are compressed on the target side and decompressed by the host. The uncompressed mode leaves the data as it is. The defines T32_FDX_TRACE_COMPRESSED and T32_FDX_TRACE_UNCOMPRESSED can switch these modes. The compressed mode has a small communication band width but needs more CPU cycles to transform the trace data.

Every time *T32_Fdx_TraceData* is called from the application the timestamp is determined by a call of *T32_Fdx_GetTimebase*. The declaration of *T32_Fdx_GetTimebase* is:

```
Declaration
unsigned long T32_Fdx_GetTimebase(void)
```

The function returns the timestamp. The implementation of it depends on the target and the resources. Many processors have hardware counter that can be used to form a result of the routine. Another possibility is to create a dedicated counter by using interrupts. All hardware based methods depend on free resources on the target. LAUTERBACH provides an FDX Trace example for many devices, but this routine may be adapted to use the right hardware resources according to the application.

DCC-Based Packet Transport Method

See below an example script of a DCC based communication.

```
FDX.RESet

//set transport method to 4-byte DCC
FDX.METHOD DCC

//set trace buffer size
FDX.SIZE 100000.

//set trace channel (no address because DCC here)
FDX.TraceChannel FdxTraceSendBuffer
FDX.OFF

//set compress option according target program settings
FDX.Mode.Compress OFF

//use stack mode for trace buffer
FDX.Mode Stack

//set timestamp rate (timestamp is retrieved from T32_Fdx_GetTimebase)
FDX.TimeStamp.Rate 300000000.

//counter is decreasing
FDX.TimeStamp.Down

//arm trace
FDX.Arm

//begin execution
Go
```

The memory based access uses nearly the same configuration script.

```
//avoid any access to the target before the CPU has executed
// T32_Fdx_InitChannel
FDX.RESet

SYStem.Up //start session
Data.Load.ELF <file> //load elf-file

//before the access to the symbol FdxSendChannel is done, it needs to be
initialized by T32_Fdx_InitChannel and T32_Fdx_EnableChannel on the target.
Go FdxChannelInitDone
WAIT!STATE.RUN()

//set transport method to dual port
FDX.METHOD BUFFERE

//set trace buffer size
FDX.SIZE 100000.

//set trace channel
FDX.TraceChannel FdxSendChannel
FDX.OFF

//set compress option according target program settings
FDX.Mode.Compress OFF

//use stack mode for trace buffer
FDX.Mode Stack

//set timestamp rate (timestamp is retrieved from T32_Fdx_GetTimebase)
FDX.TimeStamp.Rate 300000000.

//counter is decreasing
FDX.TimeStamo.Down

//arm trace
FDX.Arm

//begin execution
Go
```