Free MSP430 Debugger Link Library v0.2a

Manual

Aurélien VALADE <wolvilataniere@users.sourceforge.net>

This manual is for Free MSP430 Debugger Link Library, version 0.2a.

Published by Aurélien VALADE

Table of Contents

1 Introduction

Free MSP430 Debugger Link Library v0.2a is a free software initially designed to allow GNU/Linux user to use their OLIMEX MSP430 JTAG debugger without running a virtual machine under Windows to execute the *msp430-gdbproxy* using OLIMEX DLL.

This library is still under development and may have some problems. If you find one, I would appreciate you give me a feedback or a patch to optimize the whole library.

I hope it will grow up and get more functions than only OLIMEX debugger handling. I wrote it to be flexible, so you can "easily" add TI FET support as an example. I do not have one so I can't do it myself.

For more informations, send me a mail to `wolvilataniere@users.sourceforge.net`

The current project page is at `http://freemsp430debug.sourceforge.net/fmdll`

2 Supported Devices

This library currently supports the debuggers :

- OLIMEX MSP430 JTAG ISO (Tested and working),
- OLIMEX MSP430 JTAG TINY (not tested).

The known devices in the database are :

- MSP430F1121,
- MSP430F1132,
- MSP430F123,
- MSP430F1232,
- MSP430F149,
- MSP430F169,
- MSP430F1610,
- MSP430F2013,
- MSP430F1231,
- MSP430F2274,
- MSP430F2370,
- MSP430F249,
- MSP430F2619,
- MSP430F413,
- MSP430FW427,
- MSP430F437,
- MSP430FG439,
- MSP430F449,
- MSP430FG4619,
- MSP430F5435.

The only ones I tested are :

- MSP430F2013,
- MSP430FG4619,
- MSP430F5435.

To add support for other devices, you have to add it to the '`src/msp430-device.c`' file, in the array respecting the structure.

Note : Most of the untested devices are probably not correctly configured in the array, I just completed some data from an example from *TI Replicator (slau265e)* on which I added some fields to get the needed data.

Attention, when adding data to the array, wrong values can put the device to an unpredictable state.

Example of a new device :

```
{
0xF46F, /* The device Identifier */
FALSE, /* Test pin presence */
TRUE, /* CpuX device */
TRUE, /* DataQuick support */
TRUE, /* FastFlash support */
TRUE, /* Enhanced Memory verification */
TRUE, /* JTAG Presence */
FALSE, /* Spy-Bi-Wire support */
0x1100, /* Ram start address */
0x20FF, /* Ram end address */
0x0000, /* Ram2 start address (for future products I suppose) */
0x0000, /* Ram2 end address */
0x2100, /* Main flash start address */
0x200FF, /* Main flash end address */
0x1000, /* Informations data start address */
0x10FF, /* Informations data end address */
0x0008, /* Breakpoints count */
0x0003, /* Emulation count (I don't know where to find it) */
0x0002, /* Clock control count /*
1800, /* Minimum operating voltage (in mV) */
3600, /* Maximum operating voltage */
"MSP430FG4619" /* Device name */
},
```

3 Installation

This Library has been designed to work on a multiple choice of operating systems, also the installation steps are different on each one.

3.1 `configure` options

By default, the program will build using *libftdi* for debugger access. You can force it to build using *FTDI FTD2XX* library (for example on Windows where *libftdi* doesn't work) by using the switch `--disable-libftdi` in the `configure` options.

You can also turn the debugging output on by using `--enable-debug` switch.

3.2 GNU/Linux

Building the library will require you to have installed on your system the libraries :

- *libftdi* >= 0.14
- *glib-2.0*

Once it is present on your system, just type
`./configure`
Then, when the configuration is correctly done, launch `make` to build the library. The installation can be done by typing `make install` as we are used to.

If you want to use it with the *msp430-gdbproxy* program, you have to create a symbolic link on the file '`libMSP430.so`' to point to '`/usr/lib/libfmdll.so`'.
Note : If you are running a 64bits system, you will have problems when running msp430-gdbproxy which is a 32bits version with your 64bits shared library, you should build FMDLL with a 32bits compiler and linker.

3.3 Windows

Building on *Windows* is a little bit more tricky since it is a portage.

To build *libfmdll* on windows, you will need mingw32 and msys to generate the code.

Extract the program into *msys* home directory and create a folder tree like this :

- '`lib`'
- '`lib/windows`'
- '`lib/windows/ftd2xx`'
- '`lib/windows/glib-2.0`'

In '`lib/windows/ftd2xx`', put '`ftd2xx.h`' and '`libftd2xx.a`' (you can get it from '`ftd2xx.lib`' by using the `reimp` utility) from *FTDI D2XX* package.
In '`lib/windows/glib-2.0`', extract '`include`' and '`lib`' directory from *Glib2.0 dev package* (can be found here `http://www.gtk.org/download-windows.html`).

Then, in *msys*, use the commands :
`./configure --disable-libftdi`

```
make
```

You will get a '`libfmdll.dll`' into '`src/.libs`' or in '`/local/lib`' if you type `make install`.

3.4 MacOSX

Building on MacOSX works the same way as under Linux. You just have to ensure that *libglib-2.0-dev* and *libftdi* are installed (you can do it using macport). Once installed, just go into the *libfmdll* directory and type :

```
./configure
make
make install
```

To get the library on your system.

If you get some problems, try to type `./bootstrap` and restart from `./configure`.

Note: msp430-gdbproxy is not yet present on this platform. A free implementation which would work with the FMDLL is under development.

3.5 FreeBSD

Library building works the same way as under GNU/Linux, you just might have to launch the `./bootstrap` script before the configuration in order to regenerate the *m4* scripts and the '`libtool`' link.

Note: msp430-gdbproxy is not yet present on this platform. A free implementation which would work with the FMDLL is under development.

4 Hacking into the code

This chapter will attempt to describe the program structure in the aim to help programmers to understand the way it was designed, to add modules, functionalities as needed.

4.1 Program structure

'`crc.c`' This file contains the CRC calculation procedure for the protocol layer.

'`protocole_ftdi.c`'
: This module content the hardware access layer using *libftdi*, it is the lowest layer in this program. Here is handled the only *libftdi* accesses. All data transit to the debugger pass through this layer.

'`protocole_ftd2xx.c`'
: This module content the hardware access layer using *ftd2xx*, it is the lowest layer in this program. Here is handled the only *ftd2xx* accesses. All data transit to the debugger pass through this layer.

'`olimex_mji.c`'
: Here is the *OLIMEX MSP430 JTAG ISO* device handling. This layer handles the debugger state, opening, initialization and closing. The commands frames are generated to match the debugger protocol and are sent using the '`protocole`' module.

'`olimex_target.c`'
: This modules contains all the target relative functions used with the *OLIMEX MSP430 JTAG ISO* debugger. It works the same way as '`olimex_mji.c`' module.

'`msp430_device.c`'
: This file contains the supported device definitions, including the needed data for the debugger to run. A searching procedure is available to get a device from it's ID.

'`libmsp430_interface.c`'
: This is the upper layer on this library. It's an common interface to generate a shared object compliant with *libmsp430* library from *Texas Instrument*. All the functions are not currently implemented, the *FMDLL* library may not work with some programs.

4.2 Coding conventions

As you will notice in the next section, the coding conventions used in this project are pretty simples:

- Each file is name from the module it contains (ex: '`protocole_ftdi.c`' contains the protocol relative functions),
- All the functions names begins with the module name (ex: `protocole_init()` function is used to initialize the protocol module),

- Nearly all functions return a status telling the exit conditions,
- As soon as possible, use a variable handling the return value and an only return point (it is generally easier to debug by putting an only breakpoint),
- When possible, use a prefix for variables indicating their type (ex: `guiToto` is a `guint`),
- Respect the layers containment, it make the program easier to update and port to other platform and devices.

4.3 Detailed files description

This section exposes the modules functions and the relative exported functions, for more details, see the source code which I tried to comment as clearly as possible.

4.3.1 'crc.c'

This file is from the *fetproxy* software (see Chapter 6 [Bibliography], page 12). I used it without any modifications.
It is used to generate the sent frame CRC and to check the received ones.

4.3.2 'protocole_ftdi.c'

This module allow the program to communicate with the debugger only focusing on the frame content. It handles the whole low level aspect, from opening the USB device communication (using *libftdi*, see Chapter 6 [Bibliography], page 12) to managing the data encapsulation.

The exported functions are :

```
PROTOCOLE_STATUS protocole_init(void);
PROTOCOLE_STATUS protocole_open_device(guint16 vid, guint16 pid);
PROTOCOLE_STATUS protocole_close_device(void);
PROTOCOLE_STATUS protocole_configure_device(PROTOCOLE_CONFIG *pcConfig);
PROTOCOLE_STATUS protocole_send_data(guint8* data, size_t len, gint *sent);
PROTOCOLE_STATUS protocole_send_frame_raw(guint8* data, size_t len, gint
*sent);
PROTOCOLE_STATUS protocole_read_data(guint8* buffer, size_t buffer_size,
gint* read);
PROTOCOLE_STATUS protocole_read_frame_raw(guint8* buffer, size_t buffer_
size, gint* read);
PROTOCOLE_STATUS protocole_flush_buffers(void);
```

All these functions are declared as `DLL_LOCAL`, that means in this program that they are not exported when the shared object is generated; You can only access them from the library.

4.3.3 'protocole_ftd2xx.c'

This modules is a replacement for '`protocole_ftdi.c`' when you want to use *FTDI D2XX* library instead of *libftdi* (this is mainly used on Windows systems). It uses the same interface and can't be compiled in the same time as '`protocole_ftdi.c`'.

4.3.4 'olimex_mji.c'

This module contains the *OLIMEX MSP430 JTAG ISO* debugger specific instructions. It handles the debugger state (opened, initialized, closed) and the general commands (Debugger initialization, VCC setting...).

The exported functions are :

```
OLIMEX_STATUS olimex_initialize(guint32* version);
OLIMEX_STATUS olimex_close(guint8 close_type);
OLIMEX_STATUS olimex_vcc(gfloat fVoltage);
OLIMEX_STATUS olimex_configure(guint mode, guint val);
```

All these functions are declared as `DLL_LOCAL`, that means in this program that they are not exported when the shared object is generated; You can only access them from the library.

4.3.5 'olimex_target.c'

This module works with the same device as the precedent, it handles the target device relative functions (Identification, memory access, context reading/writing, execution control...). It works once the debugger is initialized, configured and the target is powered.

The exported functions are :

```
OLIMEX_STATUS olimex_target_identify(guint *deviceId);
OLIMEX_STATUS olimex_target_reset(guint8 param);
OLIMEX_STATUS olimex_target_state(guint32 *state, guint count, guint32*
read);
OLIMEX_STATUS olimex_target_get_context(guint32 *registers, guint regs_
count, guint* read_registers);
OLIMEX_STATUS olimex_target_set_context(guint32 *registers, guint regs_
count);
OLIMEX_STATUS olimex_target_run(guint32 mode, guint32 param);
OLIMEX_STATUS olimex_target_set_breakpoint(guint32 code, guint32 address);
OLIMEX_STATUS olimex_target_memory(guint32 mem_address, guint8 *data, guint16
data_len, OT_MEMORY_FUNCTION function);
OLIMEX_STATUS olimex_target_erase(guint8 function, guint32 address, guint16
length);
```

All these functions are declared as `DLL_LOCAL`, that means in this program that they are not exported when the shared object is generated; You can only access them from the library.

4.3.6 'msp430_device.c'

This module contains the target device relative informations; an array of structure is used to store device informations needed to debug it (CpuX core, Memory location, Flash location, spy-bi-wire support...).

The exported function is: `MSP430_DEVICE msp430_get_device(guint16 devId);`

It returns the structure containing the needed data from the device ID.

4.3.7 'libmsp430_interface.c'

This part of the library is not really a module, it makes an interface between the other modules and an external program made to work with *TI libMSP430* library. All it's functions are exported to be accessed from an external software and provides a common interface to the internal functions.

The exported functions are:
```
int MSP430_Initialize(CHAR const * port, LONG* version);
int MSP430_Close(LONG vccOff);
int MSP430_Configure(LONG mode, LONG value);
int MSP430_VCC(LONG voltage);
int MSP430_Reset(LONG method, LONG execute, LONG releaseJTAG);
int MSP430_Erase(LONG type, LONG address, LONG length);
int MSP430_Memory(LONG address, CHAR* buffer, LONG count, LONG rw);
int MSP430_Breakpoint(LONG bpNumber, LONG address);
int MSP430_Registers(guint* Regs, LONG Value, guint32 rw);
int MSP430_Run(guint32 mode, guint32 param);
int MSP430_State(guint32* state, guint32 count, guint32* read);
int MSP430_VerifyMem(LONG StartAddr, LONG Length, CHAR *DataArray);
int MSP430_EraseCheck(LONG StartAddr, LONG Length);
LONG MSP430_Error_Number(void);
const CHAR* MSP430_Error_String(LONG errorNumber);
guint32 MSP430_Identify(gint8 *data, gint size, guint32 setId);
```

4.4 Submit your patch

If you have been working for this project, it would be great to give us feed back by submitting a *patch* file that would be integreated in the distribution.

Note: Don't forget to update the 'ChangeLog' file before doing the `diff` *in order to give us a description of your changes.*

4.4.1 Modifications from the tarball

To do so is quite easy, just extract the original project beside your modified one (for example, get in a directory '`libfmdll-vX.X`' and '`libfmdll-vX.X-by-yourself`'), and use the `diff` tool to make a patch file :
```
diff -Nru libfmdll-vX.X libfmdll-vX.X-by-yourself > libfmdll-vX.X-patch-by-yourself.patch
```

You then just have to send the generated '`libfmdll-vX.X-patch-by-yourself.patch`' file by email.

4.4.2 Modifications from the GIT repository

Submitting a modification from the GIT repository is very easy.

If you did n't added your modification with `git add ''Your files''`, just type :
```
git diff > libfmdll-vX.X-patch-by-yourself-git.patch
```

If you added your modification to your local git (without committing), type :
```
git diff master > libfmdll-vX.X-patch-by-yourself-git.patch
```

Then just send an email with the generated patch.
Note: Adding modification without committing gives a better result for the patch file.

5 License

6 Bibliography

For the initial release of this project, some data and sites were helpful :

- Robert Spanton and his project *fetproxy* which inspired me and helped on some protocol specificities.
  http://xgoat.com/wp/2009/03/25/fetproxy-an-open-source-replacement-for-msp430-gdbproxy/
- Intra2net and it's library *libftdi* which support the whole communication with the debugger.
  http://www.intra2net.com/en/developer/libftdi/
- Future Technology and their *D2XX* driver to replace *libftdi* under Windows.
  http://www.ftdichip.com/
- The GTK+ project for lib GLIB-2.0 as a multi-plateforme coding standard.
  http://www.gtk.org/