python-msp430-tools Documentation

Release 0.7 beta

Chris Liechti

Sep 30, 2017

Contents

This page's online home is at: http://python-msp430-tools.readthedocs.io

Package home (PyPI): http://pypi.python.org/pypi/python-msp430-tools

Developement / Project page: https://github.com/zsquareplusc/python-msp430-tools

Contents:

CHAPTER 1

Overview

This is about the python-msp430-tools, that provide a number of tools related to the MSP430 microcontroller.

Python 2.6 or newer should be used. The Python package "msp430" can be installed with `python setup.py install`. These modules can be used as standalone applications or as library for other programs.

NEWS

Compared to the python-mspgcc-tools:

- new "target" base implementation that all upload/download tools share
- >64k address space support for most tools
- new download tool: `msp430.gdb.target`. It communicates with a GDB server for the MSP430 such as `msp430-gdbproxy` or `mspdebug`.
- new BSL implementation (F1x, F2x, F4x): `msp430.bsl.target`
- new BSL implementation (F5x, F6x): `msp430.bsl5.hid` and `msp430.bsl5.uart`
- JTAG tool renamed to: `msp430.jtag.target`
  - renamed command line options `-l/--lpt` to `-p/--port`
  - new command line option `-l/--library-path`
- all target tools:
  - renamed command line options `-P`, `-V` to upper case
  - new command line option `-U/--upload-by-file`
  - new command line option `-b/--erase-by-file`
  - multiple files on the command line are merged before downloading (supporting overlapping areas - last one counts). Useful e.g. if a boot loader part should be merged with an application part.
  - specifying input format is now one option: `-i/--input-format`

  - specifying output format is now one option: `-f/--output-format`
  - new file formats: `hex`, `bin`
- new modules:
  - `msp430.listing` (read IAR and mspgcc listing files)
  - `msp430.gdb` (GDB client code for use with GDB servers)
  - `msp430.shell.command` (busybox alike shell commands: `mv`, `cp`, `rm` and more)
  - `msp430.bsl5` (F5xx/F6xx BSL support)
    * `msp430.bsl5.hid` (USB HID frontend)
    * `msp430.bsl5.uart` (serial frontend)
- new tools:
  - `msp430.memory.convert` (convert hex file formats)
  - `msp430.memory.generate` (create hex files with fill pattern)
  - `msp430.memory.compare` (compare hex files)
  - `msp430.memory.hexdump` (show contents of hex files)
- new license: Simplified BSD License instead of Python License.

There is no longer a separate line frontend for each tool. However tools can be used as follows:

```
python -m <module name> [options] [arguments]
```

e.g.:

```
python -m msp430.bsl.target -p /dev/ttyUSB0 -e somefile.elf
```

CHAPTER 2

Commandline Tools

The following sections show the README files of the different command line tools.

Programming tools:

- msp430-bsl: F1x, F2x, F4x BSL
- msp430-bsl5: F5x, F6x BSL
- msp430-jtag: JTAG interface

Other Utilities:

- msp430-dco: clock calibration tool
- msp430-downloader: JTAG download wrapper (GUI)

msp430-bsl

MSP430 Boot Strap Loader software for F1xx, F2xx, F4xx.

Features

- Understands ELF, TI-Text and Intel-hex object files.
- Download to Flash and/or RAM, erase, verify, ...
- Reset and wait for key press (to run a device directly from the port power).
- Load address into R0/PC and run.
- Password file can be any data file, e.g. the one used to program the device in an earlier session.
- Upload a memory block MSP->PC (output as binary data or hex dump).
- Written in Python, runs on Win32, Linux, BSD (and others).
- Use on command line, or in a Python script.

- Downloadable BSL for larger devices (integrated).
- Baud rate change for newer MSP430-BSLs.
- Test and reset lines can be inverted or exchanged for non standard BSL hardware. Test singal on TX line is also possible.

Requirements

- Linux, BSD, Un*x or Windows PC
- Python 2.5 or newer
- pySerial (2.4 or newer recommended)
- BSL hardware with an MSP430 device connected to a serial port

Short introduction

First the MSP430 BSL hardware is needed. An example schematics can be found in the application note "slaa96b" from TI (see references). Then this program can be used to communicate between the PC and the MSP430 device.

The program can be started by typing "msp430-bsl" in a console. To run it in the source directory, use "python msp430-bsl"

Usage: msp430.bsl.target [OPTIONS] [FILE [FILE...]]

Options:

-h, --help	show this help message and exit
-d, --debug	print debug messages and tracebacks (development mode)
-v, --verbose	show more messages (can be given multiple times)
-q, --quiet	suppress all messages
--time	measure time
-S, --progress	show progress while programming

Data input: File format is auto detected, unless –input-format is used. Preferred file extensions are ".txt" for TI-Text format, ".a43" or ".hex" for Intel HEX. ELF files can also be loaded.

Multiple files can be given on the command line, all are merged before the download starts. "-" reads from stdin.

-i TYPE, --input-format=TYPE input format name (titext, ihex, bin, hex, elf)

Flash erase: Multiple –erase options are allowed. It is also possible to use address ranges such as 0xf000-0xf0ff or 0xf000/4k.

NOTE: SegmentA on F2xx is NOT erased with –mass-erase, that must be done separately with –erase=0x10c0 or –info-erase".

-e, --mass-erase	mass erase (clear all flash memory)
-m, --main-erase	erase main flash memory only
--info-erase	erase info flash memory only (0x1000-0x10ff)
-b, --erase-by-file	erase only Flash segments where new data is downloaded
--erase=ADDRESS	selectively erase segment at the specified address or address range

Program flow specifiers: All these options work against the file(s) provided on the command line. Program flow specifiers default to "-P" if a file is given.

"-P" usually verifies the programmed data, "-V" adds an additional verification through uploading the written data for a 1:1 compare.

No default action is taken if "-P", "-V" or "-E" is given, say specifying only "-V" does a "check by file" of a programmed device without programming.

Don't forget to erase ("-e", "-b" or "-m") before programming flash!

-E, --erase-check erase check by file

-P, --program program file

-V, --verify verify by file

-U, --upload-by-file upload the memory that is present in the given file(s)

Data upload: This can be used to read out the device memory. It is possible to use address ranges such as 0xf000-0xf0ff or 0xf000/256, 0xfc00/1k.

Multiple –upload options are allowed.

-u ADDRESS, --upload=ADDRESS upload a data block, can be passed multiple times

-o DESTINATION, --output=DESTINATION write uploaded data to given file

-f TYPE, --output-format=TYPE output format name (titext, ihex, bin, hex), default:hex

Do before exit:

-x ADDRESS, --execute=ADDRESS start program execution at specified address, might only be useful in conjunction with –wait

-r, --reset perform a normal device reset that will start the program that is specified in the reset interrupt vector

-w, --wait wait for <ENTER> before closing the port

--no-close do not close port on exit

Communication settings:

-p PORT, --port=PORT Use com-port

--invert-test invert RTS line

--invert-reset invert DTR line

--swap-reset-test exchenage RST and TEST signals (DTR/RTS)

--test-on-tx TEST/TCK signal is muxed on TX line

BSL settings:

--no-start no not use ROM-BSL start pattern on RST+TEST/TCK

-s SPEED, --speed=SPEED change baud rate (default 9600)

--password=FILE transmit password before doing anything else, password is given in given (TI-Text/ihex/etc) file

--ignore-answer do not wait for answer to BSL commands

--control-delay=CONTROL_DELAY set delay in seconds (float) for BSL start pattern

--replace-bsl download replacement BSL (V1.50) for F1x and F4x devices with 2k RAM

--erase-cycles=EXTRA_ERASE_CYCLES configure extra erase cycles (e.g. very old F149 chips require this for –main-erase)

If it says `command failed (DATA_NAK)` it's probably because no or a wrong password was specified, while a `ERROR:BSL:Sync failed, aborting...` is typical when the BSL could not be started at all.

Examples

`led.txt` in the following examples is a place holder for some sort of binary for the MSP430. A `led.txt` that contains an example in TI-Text format can be built from the code in `examples/asm/led`.

`msp430-bsl -e` Only erase flash.

`msp430-bsl -eErw led.txt` Erase flash, erase check, download an executable, run it (reset) and wait.

Old F149 devices need additional erase cycles! Use the `--erase-cycles` option in this case (`--erase-cycles 20` will be OK is most cases)

`msp430-bsl led.txt` Download of an executable to en empty (new or erased) device. (Note that in new devices, some of the first bytes in the information memory are random data. If data should be downloaded there, specify -e.)

`msp430-bsl --upload 0x0c00/1024 --password led.txt` Get a memory dump in HEX, from the bootstrap loader (on a device that was previously programmed with led.txt and therefore needs a specific password):

`msp430-bsl -rw` Just start the user program (with a reset) and wait.

`cat led.txt|msp430-bsl -e -` Pipe the data from "cat" to the BSL to erase and program the flash. (un*x example, don't forget the dash at the end of the line)

`msp430-bsl --replace-bsl -e -s 38400 led.txt` First download the internal replacement BSL and then use it to program at 38400 baud. Only works with targets with more than 1kB of RAM. Newer devices with already know this command, in that case omit the `--replace-bsl`

History

V1.4 uses improved serial library, support for BSL download to MSP, support for higher baudrates (up to 38400)

V1.5 ELF file support, replacement BSLs are now internal

V2.0 New implementation. Some command line options have been renamed or replaced.

References

- Python: http://www.python.org
- pySerial: Serial port extension for Python http://pypi.python.org/pypi/pyserial
- slaa89.pdf: "Features of the MSP430 Bootstrap Loader in the MSP430F1121", TI, http://www.ti.com/msp430
- slaa96b.pdf: "Application of Bootstrap Loader in MSP430 With Flash Hardware and Software Proposal", TI
- Texas Instruments MSP430 Homepage, links to data sheets and application notes: http://www.ti.com/msp430

msp430-bsl5

MSP430 Boot Strap Loader software for F5xx, F6xx.

Features

- Understands ELF, TI-Text and Intel-hex object files.
- Download to Flash and/or RAM, erase, verify, ...
- Reset and wait for key press (to run a device directly from the port power).
- Load address into R0/PC and run.
- Password file can be any data file, e.g. the one used to program the device in an earlier session.
- Upload a memory block MSP->PC (output as binary data or hex dump).
- Written in Python, runs on Win32, Linux, BSD (and others).
- Use on command line, or in a Python script.
- USB-HID BSL version:
  - Automatic detection of HID device.
- UART BSL version:
  - Baud rate change
  - Test and reset lines can be inverted and/or exchanged for non standard BSL hardware. Test singal on TX line is also possible.

Requirements

- Linux, BSD, Un*x or Windows PC
- Python 2.6 or newer
- USB support requires:
  - "pywinusb" library on Windows
  - "rawhid" kernel driver on Linux
  - other platforms are currently not supported
- pySerial (2.4 or newer recommended)
- MSP430 F5x / F6x with UART BSL connected to a serial port or a USB capable device connected to USB.

Short introduction

There are separate command line fontends for the USB and UART version:

- `python -m msp430.bsl5.uart` - UART version
- `python -m msp430.bsl5.hid` - USB version

Usage: hid.py [OPTIONS] [FILE [FILE...]]

Options:

-h, --help	show this help message and exit
--debug	print debug messages and tracebacks (development mode)
-v, --verbose	show more messages (can be given multiple times)
-q, --quiet	suppress all messages
--time	measure time
-S, --progress	show progress while programming

Data input: File format is auto detected, unless –input-format is used. Preferred file extensions are ".txt" for TI-Text format, ".a43" or ".hex" for Intel HEX. ELF files can also be loaded.

Multiple files can be given on the command line, all are merged before the download starts. "-" reads from stdin.

-i TYPE, --input-format=TYPE input format name (titext, ihex, bin, hex, elf)

Flash erase: Multiple –erase options are allowed. It is also possible to use address ranges such as 0xf000-0xf0ff or 0xf000/4k.

NOTE: SegmentA on F2xx is NOT erased with –mass-erase, that must be done separately with –erase=0x10c0 or –info-erase".

-e, --mass-erase	mass erase (clear all flash memory)
-m, --main-erase	erase main flash memory only
--info-erase	erase info flash memory only (0x1000-0x10ff)
-b, --erase-by-file	erase only Flash segments where new data is downloaded
--erase=ADDRESS	selectively erase segment at the specified address or address range

Program flow specifiers: All these options work against the file(s) provided on the command line. Program flow specifiers default to "-P" if a file is given.

"-P" usually verifies the programmed data, "-V" adds an additional verification through uploading the written data for a 1:1 compare.

No default action is taken if "-P", "-V" or "-E" is given, say specifying only "-V" does a "check by file" of a programmed device without programming.

Don't forget to erase ("-e", "-b" or "-m") before programming flash!

-E, --erase-check	erase check by file
-P, --program	program file
-V, --verify	verify by file
-U, --upload-by-file	upload the memory that is present in the given file(s)

Data upload: This can be used to read out the device memory. It is possible to use address ranges such as 0xf000-0xf0ff or 0xf000/256, 0xfc00/1k.

Multiple –upload options are allowed.

-u ADDRESS, --upload=ADDRESS upload a data block, can be passed multiple times

-o DESTINATION, --output=DESTINATION write uploaded data to given file

-f TYPE, --output-format=TYPE output format name (titext, ihex, bin, hex), default:hex

Do before exit:

-x ADDRESS, --execute=ADDRESS start program execution at specified address, might only be useful in conjunction with –wait

-r, --reset perform a normal device reset that will start the program that is specified in the reset interrupt vector

-w, --wait wait for <ENTER> before closing the port

--no-close do not close port on exit

Communication settings:

-d DEVICE, --device=DEVICE device name (default: auto detection)

BSL settings:

--password=FILE transmit password before doing anything else, password is given in given (TI-Text/ihex/etc) file

The UART version only differs in the options controlling the "Communication" and "BSL" settings:

Communication settings:

-p PORT, --port=PORT Use com-port

--invert-test invert RTS line

--invert-reset invert DTR line

--swap-reset-test exchenage RST and TEST signals (DTR/RTS)

--test-on-tx TEST/TCK signal is muxed on TX line

BSL settings:

--no-start no not use ROM-BSL start pattern on RST+TEST/TCK

-s SPEED, --speed=SPEED change baud rate (default 9600)

--password=FILE transmit password before doing anything else, password is given in given (TI-Text/ihex/etc) file

--ignore-answer do not wait for answer to BSL commands

--control-delay=CONTROL_DELAY set delay in seconds (float) for BSL start pattern

Examples

`led.txt` in the following examples is a place holder for some sort of binary for the MSP430. A `led.txt` that contains an example in TI-Text format can be built from the code in `examples/asm/led5x`.

`python -m msp430.bsl5.hid -e` Only erase flash.

`python -m msp430.bsl5.uart -eErw led.txt` Erase flash, erase check, download an executable, run it (reset) and wait.

`python -m msp430.bsl5.hid led.txt` Download of an executable to en empty (new or erased) device. (Note that in new devices, some of the first bytes in the information memory are random data. If data should be downloaded there, specify -e.)

`python -m msp430.bsl5.hid --upload 0xf000/1024 --password led.txt` Get a memory dump in HEX, from a part of the memory (on a device that was previously programmed with led.txt and therefore needs a specific password):

python -m msp430.bsl5.uart -rw‘ Just start the user program (with a reset) and wait.

cat led.txt|python -m msp430.bsl5.uart -e - Pipe the data from "cat" to the BSL to erase and program the flash. (un*x example, don't forget the dash at the end of the line)

python -m msp430.bsl5.uart -e -s 38400 led.txt Change to faster baud rate for download.

Tips & Tricks

USB-HID Linux permissions The USB HID device simply works when plugged in under Linux and the tool can use the device when the "rawhid" kernel module is present. It will create `/dev/rawhid*` devices. However, those devices are usually only writeable by root. To automatically change the permissions of the device, the following udev rule can be applied.

Create a file, e.g. `/etc/udev/rules.d/20-msp430-hid.rules` with the following contents:

```
SUBSYSTEM=="hidraw", ATTRS{idVendor}=="2047", ATTRS{idProduct}=="0200" , MODE=
↪"0666"
```

History

V1.0 New tool.

References

- Python: http://www.python.org
- pySerial: Serial port extension for Python http://pypi.python.org/pypi/pyserial
- pywinusb: USB HID library http://pypi.python.org/pypi/pywinusb/
- slau319a.pdf: "MSP430 Programming Via the Bootstrap Loader" http://www.ti.com/msp430
- Texas Instruments MSP430 Homepage, links to data sheets and application notes: http://www.ti.com/msp430

msp430-jtag

Software to talk to the parallel port and USB JTAG adapters for the MSP430.

Features

- understands ELF, TI-Text and Intel-hex object files
- download to Flash and/or RAM, erase flash, verify
- reset device
- upload a memory block MSP->PC (output as binary data or hex dump, ihex)
- written in Python, runs on Win32, Linux, BSD, ...
- use on command line, or in a Python script
- reset and wait for key press (to run a device directly from the port power)
- TI/3rd party library support for USB JTAG adaptors

Requirements

- Linux, BSD, Un*x or Windows PC
- Python 2.5 or newer
- Parallel JTAG hardware with an MSP430 device connected
- or USB adapter with a corresponding [3rd party] MSP430 library

Short introduction

This software uses the JTAG hardware that comes with the FET kits. It is connected to the parallel port. Using 3rd party backends it is also possible to use USB programmers.

The program can be started by typing `msp430-jtag` when installed correctly If it's used from the source directory use "python -m msp430.jtag.target".

Usage: msp430.jtag.target [OPTIONS] [FILE [FILE...]]

Options:

-h, --help	show this help message and exit
-d, --debug	print debug messages and tracebacks (development mode)
-v, --verbose	show more messages (can be given multiple times)
-q, --quiet	suppress all messages
--time	measure time
-S, --progress	show progress while programming
--help-backend	show help about the different backends
-l LIBRARY_PATH, --library-path=LIBRARY_PATH	search for libMSP430.so or libMSP430mspgcc.so in this place first

Data input: File format is auto detected, unless –input-format is used. Preferred file extensions are ".txt" for TI-Text format, ".a43" or ".hex" for Intel HEX. ELF files can also be loaded.

Multiple files can be given on the command line, all are merged before the download starts. "-" reads from stdin.

-i TYPE, --input-format=TYPE	input format name (titext, ihex, bin, hex, elf)

Flash erase: Multiple –erase options are allowed. It is also possible to use address ranges such as 0xf000-0xf0ff or 0xf000/4k.

NOTE: SegmentA on F2xx is NOT erased with –mass-erase, that must be done separately with –erase=0x10c0 or –info-erase".

-e, --mass-erase	mass erase (clear all flash memory)
-m, --main-erase	erase main flash memory only
--info-erase	erase info flash memory only (0x1000-0x10ff)
-b, --erase-by-file	erase only Flash segments where new data is downloaded
--erase=ADDRESS	selectively erase segment at the specified address or address range

Program flow specifiers: All these options work against the file(s) provided on the command line. Program flow specifiers default to "-P" if a file is given.

"-P" usually verifies the programmed data, "-V" adds an additional verification through uploading the written data for a 1:1 compare.

No default action is taken if "-P", "-V" or "-E" is given, say specifying only "-V" does a "check by file" of a programmed device without programming.

Don't forget to erase ("-e", "-b" or "-m") before programming flash!

-E, --erase-check erase check by file

-P, --program program file

-V, --verify verify by file

-U, --upload-by-file upload the memory that is present in the given file(s)

Data upload: This can be used to read out the device memory. It is possible to use address ranges such as 0xf000-0xf0ff or 0xf000/256, 0xfc00/1k.

Multiple –upload options are allowed.

-u ADDRESS, --upload=ADDRESS upload a data block, can be passed multiple times

-o DESTINATION, --output=DESTINATION write uploaded data to given file

-f TYPE, --output-format=TYPE output format name (titext, ihex, bin, hex), default:hex

Do before exit:

-x ADDRESS, --execute=ADDRESS start program execution at specified address, might only be useful in conjunction with –wait

-r, --reset perform a normal device reset that will start the program that is specified in the reset interrupt vector

-w, --wait wait for <ENTER> before closing the port

--no-close do not close port on exit

Connection: NOTE: On Windows, use "USB", "TIUSB" or "COM5" etc if using MSP430.dll from TI. On other platforms, e.g. Linux, use "/dev/ttyUSB0" etc. if using libMSP430.so. If a libMSP430.so is found, it is preferred, otherwise libMSP430mspgcc.so is used.

NOTE: –slowdown > 50 can result in failures for the RAM size auto detection (use –ramsize option to fix this). Use the –verbose option and watch the outputs. The DCO clock adjustment and thus the Flash timing may be inaccurate for large values.

--backend=BACKEND select an alternate backend. See –help-backend for more information

-p PORT, --port=PORT specify an other parallel port or serial port for the USBFET (the later requires libMSP430.so instead of libMSP430mspgcc.so). (defaults to "LPT1" ("/dev/parport0" on Linux))

--spy-bi-wire-jtag interface is 4 wire on a spy-bi-wire capable device

--spy-bi-wire interface is 2 wire on a spy-bi-wire capable device

--slowdown=MICROSECONDS artificially slow down the communication. Can help with long lines, try values between 1 and 50 (parallel port interface with mspgcc's HIL library only). (experts only)

-R BYTES, --ramsize=BYTES specify the amount of RAM to be used to program flash (default: auto detected)

--unlock-bsl unlock Flash BSL (e.g. F5x)

JTAG fuse: WARNING: This is not reversible, use with care! Note: Not supported with the simple parallel port adapter (7V source required).",

--secure blow JTAG security fuse

Examples: Mass erase and program from file: "/home/lch/python-mspgcc- tools/msp430/jtag/target.py -e firmware.elf" Dump information memory: "/home/lch/python-mspgcc-tools/msp430/jtag/target.py –upload=0x1000-0x10ff"

Note: Some versions of the Texas Instruments MSP430 Development Tool require that you give the '–no-close' option to msp430-jtag. This is because the Texas Instruments tool is powered via the JTAG adapter; the '–no-close' option prevents msp430-jtag from powering the adapter off. You may also need to restart the program with msp430-jtag (using the '–no-close' and '-r' options is sufficient) after rebooting your machine.

Other development kits that rely on the parallel port for their power source may also need the '–no-close' option. It is preferable to try programming the device *without* the '–no-close' option first, and introduce this option only if the uploaded code fails to start.

Alternatively, it is possible run `msp430-jtag -w` to power the eval board from the JTAG interface.

Note: –secure on F1x,F2x,F4x requires support by the JTAG interface for higher voltage.

Warning: –secure is irreversible and disabled JTAG access

Warning: –unlock-bsl enables writes to the BSL memory area on F5x/F6x devices. The user is responsible to erase and program appropriately, otherwise the device can be made unusable (JTAG disabled, BSL disabled, no user code running),

Backends

msp430-jtag can use different libraries to connect to the target. The backend can be chosen with the –backend command line option.

"mspgcc" Using MSP430mspgcc.dll, the open source implementation from the mspgcc project.

"ti" (default) Using MSP430.dll, the proprietary library from TI or a compatible one from a 3rd party supplier.

"parjtag" Old way of using MSP430mspgcc.dll. Use "mspgcc" instead.

Compatibility of backends:

Feature	mspgcc	ti
4 Wire JTAG	yes	yes
4 Wire JTAG on devices with spy-bi-wire	yes(1)	no
using –spy-bi-wire option	no	yes
support for USB JTAG adapters	no	yes
unsing –funclet option	yes	no

Notes:

1. Timing critical, may not work on all machines or at every try.

Examples

msp430-jtag -e Only erase flash.

msp430-jtag -eErw led.txt Erase flash, erase check, download an executable, run it (reset) and wait, the keep it powered (from the parallel port).

msp430-jtag led.txt Download of an executable to en empty (new or erased) device. (Note that in new devices some of the first bytes in the information memory are random data. If data should be downloaded there, specify -eE.)

msp430-jtag --go=0x220 ramtest.a43 Download a program into RAM and run it, may not work with all devices.

msp430-jtag -u 0x0c00/1k Get a memory dump in HEX, from the bootstrap loader. Or save the binary in a file:

```
msp430-jtag -u 0x0c00 -s 1024 -f bin >dump.bin
```

or as an intel-hex file:

```
msp430-jtag -u 0x0c00 -s 1024 -f ihex >dump.a43
```

msp430-jtag Just start the user program (with a reset).

cat led.txt|msp430-jtag -e - Pipe the data from "cat" to msp430-jtag to erase and program the flash. (un*x example, don't forget the dash at the end of the line)

msp430-jtag --unlock-bsl --erase 0x1000/2k newbsl.a43 Write a replacement BSL to a F5x or F6x device. This is a dangerous option, if it goes wrong, the device may become inaccessible (JTAG disabled, BSL disabled, no user code running). Never write new code without –erase as the BSL would end up being an AND combination of the previous Flash content and the new code, which is certainly not executing properly. Unfortunately does the BSL signature that is checked by the boot code of the CPU not guard against this mistake. However, it is allowed to erase but write no new BSL.

NOTE: Only do this if you know how to write new BSL code. The BSL signature are in the binary must contain the correct data, otherwise the MCU may be rendered unusable.

NOTE: Works only with MSP430.dll as backend.

USB JTAG adapters

This section only applies to Windows. On Linux replace MSP430.dll with libMSP430.so etc.

USB JTAG adapters are supported through the MSP430.dlls from the adaptor vendor. To enable its use, copy MSP430.dll to the `bin\lib` folder, where `shared.zip` is located. Optionally copy `HIL.dll` to the `bin` folder.

For example for MSP-FET430UIF from TI:

- download a the MSP430.dll binary from the downloads section in http://mspgcc.sf.net
- copy MSP430.dll to `c:\mspgcc\bin` (substitute the source and destination folders according to you own setup)

The windows installer already includes this library.

To use the first available MSP-FET430UIF:

```
msp430-jtag -p TIUSB --upload=0x0ff0
```

The MSP-FET430UIF is registered as serial port. If more than one MSP-FET430UIF is connected, find out which COM port the desired adapter is using with the Device Manager. Then for example run:

```
msp430-jtag -p COM5 --upload=0x0ff0
```

Linux users have to specify the serial port differently:

```
msp430-jtag -p /dev/ttyUSB0 --upload=0x0ff0
```

History

V1.0 Public release.

V1.1 Fix of verify error.

V1.2 Use the verification during programming.

V1.3 Mainerase, progress options, ihex output.

V2.0 Updated implementation, new ctypes backend.

V2.1 F2xx support, improved options for funclets.

V2.2 Added –quiet and –secure. Try to use 3rd party MSP430 libraries so that USB adapters can be used. Allow multiple –upload with address ranges.

V2.3 Added support for F2xx and MSP430X architectures. Improved 3rd party library support for Linux and Windows.

V3.0 Rewrite command line frontend. Changed file type options, program flow specifiers.

References

- Python: http://www.python.org
- ctypes: http://starship.python.net/crew/theller/ctypes This module is included in the standard distribution since Python 2.5: http://docs.python.org/lib/module-ctypes.html
- Texas Instruments MSP430 homepage, links to data sheets and application notes: http://www.ti.com/msp430

msp430-dco

MSP430 clock calibration utility.

Features

- can handle F1xx, F2xx and F4xx devices, with or without external Rsel resistor
- measure calibration values for a given frequency
- restore calibration values of F2xx devices

- selectable clock tolerance
- can write measured values to the target flash, output C code or #defines

Requirements

- Linux, BSD, Un*x or Windows PC
- Python 2.5 or newer
- Parallel JTAG hardware with an MSP430 device connected (currently only the parallel port adapter with the MSP430mspgcc library is supported)

Short introduction

This software uses the JTAG hardware that comes with the FET kits. It is connected to the parallel port.

The program can be started by typing `msp430-dco` when installed correctly If it's used from the source directory use `python -m msp430.jtag.dco`.

Usage: msp430.jtag.dco [options] frequency

MSP430 clock calibration utility V1.1

This tool can measure the internal oscillator of F1xx, F2xx and F4xx devices, display the supported frequencies, or run a software FLL to find the settings for a specified frequency.

The target device has to be connected to the JTAG interface.

Examples:

See min and max clock speeds: dco.py –measure

Get clock settings for 2.0MHz +/-1%: dco.py –tolerance=0.01 2.0e6

Write clock calibration for 1.5MHz to the information memory at 0x1000: dco.py 1.5e6 BCSCTL1@0x1000 DCOCTL@0x1000

Use it at your own risk. No guarantee that the values are correct.

Options:

-h, --help	show this help message and exit
-o FILE, --output=FILE	write result to given file
--dcor	use external resistor
-d, --debug	print debug messages
-l LPT, --lpt=LPT	set the parallel port
-m, --measure	measure min and max clock settings and exit
-c, --calibrate	Restore calibration values on F2xx devices
-t TOLERANCE, --tolerance=TOLERANCE	set the clock tolerance as factor. e.g. 0.01 means 1% (default=0.005)
--define	output #defines instead of assignments
--erase=ERASE	erase flash page at given address. Use with care!

Variables

Arguments in the form `variable@address` are used to write the corresponding values to the target device. Variable names are case insensitive, addresses can be specified in decimal, octal or hexadecimal format.

The available variables depend on the target type and executed operation. All variables that are written all caps in the table below are in `unsigned char` format, others in `unsigned short` format. The later should be written to even addresses only, as the code reading these values could have problems otherwise.

Frequencies are in kHz.

Operation	*MCU*	*Variables*
frequency	F1xx	BCSCTL1 BCSCTL2 DCOCTL freq
	F2xx	BCSCTL1 BCSCTL2 DCOCTL freq
	F4xx	SCFI0 SCFI1 SCFQCTL FLL_CTL0 FLL_CTL1 freq
–measure	F1xx	fmax fmin rsel0_fmax rsel0_fmin rsel1_fmax rsel1_fmin rsel2_fmax rsel2_fmin rsel3_fmax rsel3_fmin rsel4_fmax rsel4_fmin rsel5_fmax rsel5_fmin rsel6_fmax rsel6_fmin rsel7_fmax rsel7_fmin
	F2xx	fmax fmin rsel0_fmax rsel0_fmin rsel1_fmax rsel1_fmin rsel2_fmax rsel2_fmin rsel3_fmax rsel3_fmin rsel4_fmax rsel4_fmin rsel5_fmax rsel5_fmin rsel6_fmax rsel6_fmin rsel7_fmax rsel7_fmin rsel8_fmax rsel8_fmin rsel9_fmax rsel9_fmin rsel10_fmax rsel10_fmin rsel11_fmax rsel11_fmin rsel12_fmax rsel12_fmin rsel13_fmax rsel13_fmin rsel14_fmax rsel14_fmin rsel15_fmax rsel15_fmin
	F4xx	fmax fmin
–calibrate	F1xx	*not supported*
	F2xx	f16MHz_dcoctl f16MHz_bcsctl1 f12MHz_dcoctl f12MHz_bcsctl1 f8MHz_dcoctl f8MHz_bcsctl1 f1MHz_dcoctl f1MHz_bcsctl1
	F4xx	*not supported*

When the `msp430-dco` tool is run with the `--debug` option it provides an output with all the possible variables and their values.

Examples

`msp430-dco 2.5e6` Print the calibration values for 2.5MHz

`msp430-dco 2.5e6 --define` Same as above, but format the output as defines usable for C include files.

`msp430-dco 1e6 --erase 0x1000 BCSCTL1@0x1000 DCOCTL@0x1001` Measure calibration values for 1MHz, then erase the information memory flash page at 0x1000. These values are then written to the flash at 0x1000 and 0x1001.

This can be useful in combination with firmware downloads. For example make a mass erase, write firmware, then write clock calibration for this device:

```
msp430-jtag -e my_firmware.elf
msp430-dco 1e6 BCSCTL1@0x1000 DCOCTL@0x1001
```

The firmware can then read the values from the flash and configure the Basic Clock System using these values.

`msp430-dco --measure` Print frequency ranges of all DCO settings as well as minimal and maximal values. (Note: restricted functionality on F4xx devices)

`msp430-dco --calibrate` Recalculate the calibration values for 16MHz, 12MHz, 8MHz and 1MHz that are available in the information memory at 0x10f8-0x10ff. This is only possible for F2xx devices.

Known Issues

The algorithm does not search for the best match, it stops when the frequency is within the window. Therefore it's not unlikely that the frequency is at the border of the tolerance window and not in the center.

History

V1.0 Public release.

V1.1 Can write values to target flash

References

- Python: http://www.python.org
- Texas Instruments MSP430 homepage, links to data sheets and application notes: http://www.ti.com/msp430

msp430-downloader

Software to talk to the parallel port and USB JTAG adapters as seen with the FET kits.

Features

- understands ELF, TI-Text and Intel-hex object files
- download to Flash and/or RAM, erase flash, verify
- reset device
- upload a memory block MSP->PC (output as binary data or hex dump, ihex)
- written in Python, runs on Win32, Linux, BSD, ...
- use on command line, or in a Python script
- reset and wait for keypress (to run a device directly from the port power)
- TI/3rd party library support for USB JTAG adaptors (Windows only)

Short introduction

The tool is intended to be assigned to .a43 and .elf files.

Without configuration file a dialog box is shown, first to ask for the programmer type, USB or parallel, and then the erase mode. These settings and some additional options can be preconfigured in a configuration file.

The configuration file together with the binary can be bundled into a single ZIP archive (extension must be renamed to .z43). The name of the configuration file is irrelevant as the first one with the ending `.m43` is loaded. The binary is referenced in the configuration file, its name must match.

Example configuration file `downloader-demo.m43`:

```
########################################################################
# This is a configuration file for msp430-download
# It shows and describes all available options.
#
# When used as separate file:
#   - copy a binary to the destination folder
#   - copy this file to the destination folder
#   - edit configuration for your needs
#
# When used with a ZIP file:
#   - as above copy binary and configuration file, edit config
#   - add all files to a zip file and rename it with to a .z43 ending
#
########################################################################

[modes]
########################################################################
## Erase modes:
##   "all" or "mass"     erase all memory
##   "main"              leave information memory
##   "ask"               ask the user
erase_mode = mass

########################################################################
## Interface selection:
##   "ask"               ask the user
##   "1" or "parallel"   parallel port. hint: numbers: LPT1, LPT2 etc
##   "TIUSB" or "COMn"   USB interface
interface = parallel

########################################################################
## Program in a loop, so that several targets can easily be programmed
## Single run and exit if not set.
#loop = Yes

########################################################################
## Ask again before programming.
## Recommended if no ther questions before programming are enabled, so
## that the user has a chance to abort. It is forced on if "loop"
## programming is on.
#ask_start = Yes

########################################################################
## Fake the progress bar and increment depending on state, not depending
## on data. Automatically set if the USB JTAG is used.
fake_progess = No

########################################################################
## For developers only. Remove key or set it to "no" for releases.
## When enabled, some diagnostic messages are printed to stdout.
#debug = Yes

########################################################################
## Backend selection:
##   "mspgcc"            use MSP430mspgcc.dll
##   "parjtag"           use _parjtag + MSP430mspgcc.dll (not recommended)
##   "ti"                use MSP430.dll from TI ord 3rd party
## Autodetect if key is not given.
```

```
#backend = mspgcc

[data]
###############################################################################
## A filename can be predefined.
## File open dialog will not be shown in this case.
filename = leds.a43

###############################################################################
## If defined, a question is displayed, asking the user if he wants to
## see the readme.
#readme = readme.txt

###############################################################################
## Select the viewer for the readme. Possible values are:
##   "browser"           the default web browser or text editor, depending
##                       on file ending
##   "internal"          use a message box (only for very short texts)
viewer = browser
```

CHAPTER 3

Target Tools

msp430.bsl.target

python -m msp430.bsl.target -h [OPTIONS] [FILE [FILE...]]:

```
Options:
  -h, --help            show this help message and exit
  --debug               print debug messages and tracebacks (development mode)
  -v, --verbose         show more messages (can be given multiple times)
  -q, --quiet           suppress all messages
  --time                measure time
  -S, --progress        show progress while programming

  Data input:
    File format is auto detected, unless --input-format is used. Preferred
    file extensions are ".txt" for TI-Text format, ".a43" or ".hex" for
    Intel HEX. ELF files can also be loaded.

    Multiple files can be given on the command line, all are merged before
    the download starts. "-" reads from stdin.

    -i TYPE, --input-format=TYPE
                        input format name (titext, ihex, bin, hex, elf)

  Flash erase:
    Multiple --erase options are allowed. It is also possible to use
    address ranges such as 0xf000-0xf0ff or 0xf000/4k.

    NOTE: SegmentA on F2xx is NOT erased with --mass-erase, that must be
    done separately with --erase=0x10c0 or --info-erase".

    -e, --mass-erase    mass erase (clear all flash memory)
    -m, --main-erase    erase main flash memory only
    --info-erase        erase info flash memory only (0x1000-0x10ff)
    -b, --erase-by-file
```

```
                        erase only Flash segments where new data is downloaded
  --erase=ADDRESS       selectively erase segment at the specified address or
                        address range

Program flow specifiers:
  All these options work against the file(s) provided on the command
  line. Program flow specifiers default to "-P" if a file is given.

  "-P" usually verifies the programmed data, "-V" adds an additional
  verification through uploading the written data for a 1:1 compare.

  No default action is taken if "-P", "-V" or "-E" is given, say
  specifying only "-V" does a "check by file" of a programmed device
  without programming.

  Don't forget to erase ("-e", "-b" or "-m") before programming flash!

  -E, --erase-check     erase check by file
  -P, --program         program file
  -V, --verify          verify by file
  -U, --upload-by-file
                        upload the memory that is present in the given file(s)

Data upload:
  This can be used to read out the device memory. It is possible to use
  address ranges such as 0xf000-0xf0ff or 0xf000/256, 0xfc00/1k.

  Multiple --upload options are allowed.

  -u ADDRESS, --upload=ADDRESS
                        upload a data block, can be passed multiple times
  -o DESTINATION, --output=DESTINATION
                        write uploaded data to given file
  -f TYPE, --output-format=TYPE
                        output format name (titext, ihex, bin, hex),
                        default:hex

Do before exit:
  -x ADDRESS, --execute=ADDRESS
                        start program execution at specified address, might
                        only be useful in conjunction with --wait
  -r, --reset           perform a normal device reset that will start the
                        program that is specified in the reset interrupt
                        vector
  -w, --wait            wait for <ENTER> before closing the port
  --no-close            do not close port on exit

Communication settings:
  -p PORT, --port=PORT
                        Use com-port
  --invert-test         invert RTS line
  --invert-reset        invert DTR line
  --swap-reset-test     exchenage RST and TEST signals (DTR/RTS)
  --test-on-tx          TEST/TCK signal is muxed on TX line

BSL settings:
  --no-start            no not use ROM-BSL start pattern on RST+TEST/TCK
  -s SPEED, --speed=SPEED
```

```
                        change baud rate (default 9600)
  --password=FILE       transmit password before doing anything else, password
                        is given in given (TI-Text/ihex/etc) file
  --ignore-answer       do not wait for answer to BSL commands
  --control-delay=CONTROL_DELAY
                        set delay in seconds (float) for BSL start pattern
  --replace-bsl         download replacement BSL (V1.50) for F1x and F4x
                        devices with 2k RAM
  --erase-cycles=EXTRA_ERASE_CYCLES
                        configure extra erase cycles (e.g. very old F149 chips
                        require this for --main-erase)
```

msp430.bsl5.hid

python -m msp430.bsl5.hid [OPTIONS] [FILE [FILE...]]:

```
Options:
  -h, --help            show this help message and exit
  --debug               print debug messages and tracebacks (development mode)
  -v, --verbose         show more messages (can be given multiple times)
  -q, --quiet           suppress all messages
  --time                measure time
  -S, --progress        show progress while programming

  Data input:
    File format is auto detected, unless --input-format is used. Preferred
    file extensions are ".txt" for TI-Text format, ".a43" or ".hex" for
    Intel HEX. ELF files can also be loaded.

    Multiple files can be given on the command line, all are merged before
    the download starts. "-" reads from stdin.

    -i TYPE, --input-format=TYPE
                        input format name (titext, ihex, bin, hex, elf)

  Flash erase:
    Multiple --erase options are allowed. It is also possible to use
    address ranges such as 0xf000-0xf0ff or 0xf000/4k.

    NOTE: SegmentA on F2xx is NOT erased with --mass-erase, that must be
    done separately with --erase=0x10c0 or --info-erase".

    -e, --mass-erase    mass erase (clear all flash memory)
    -m, --main-erase    erase main flash memory only
    --info-erase        erase info flash memory only (0x1000-0x10ff)
    -b, --erase-by-file
                        erase only Flash segments where new data is downloaded
    --erase=ADDRESS     selectively erase segment at the specified address or
                        address range

  Program flow specifiers:
    All these options work against the file(s) provided on the command
    line. Program flow specifiers default to "-P" if a file is given.

    "-P" usually verifies the programmed data, "-V" adds an additional
    verification through uploading the written data for a 1:1 compare.
```

```
    No default action is taken if "-P", "-V" or "-E" is given, say
    specifying only "-V" does a "check by file" of a programmed device
    without programming.

    Don't forget to erase ("-e", "-b" or "-m") before programming flash!

    -E, --erase-check   erase check by file
    -P, --program       program file
    -V, --verify        verify by file
    -U, --upload-by-file
                        upload the memory that is present in the given file(s)

  Data upload:
    This can be used to read out the device memory. It is possible to use
    address ranges such as 0xf000-0xf0ff or 0xf000/256, 0xfc00/1k.

    Multiple --upload options are allowed.

    -u ADDRESS, --upload=ADDRESS
                        upload a data block, can be passed multiple times
    -o DESTINATION, --output=DESTINATION
                        write uploaded data to given file
    -f TYPE, --output-format=TYPE
                        output format name (titext, ihex, bin, hex),
                        default:hex

  Do before exit:
    -x ADDRESS, --execute=ADDRESS
                        start program execution at specified address, might
                        only be useful in conjunction with --wait
    -r, --reset         perform a normal device reset that will start the
                        program that is specified in the reset interrupt
                        vector
    -w, --wait          wait for <ENTER> before closing the port
    --no-close          do not close port on exit

  Communication settings:
    -d DEVICE, --device=DEVICE
                        device name (default: auto detection)

  BSL settings:
    --password=FILE     transmit password before doing anything else, password
                        is given in given (TI-Text/ihex/etc) file
```

msp430.bsl5.uart

python -m msp430.bsl5.uart -h [OPTIONS] [FILE [FILE...]]:

```
Options:
  -h, --help            show this help message and exit
  --debug               print debug messages and tracebacks (development mode)
  -v, --verbose         show more messages (can be given multiple times)
  -q, --quiet           suppress all messages
  --time                measure time
  -S, --progress        show progress while programming
```

```
Data input:
  File format is auto detected, unless --input-format is used. Preferred
  file extensions are ".txt" for TI-Text format, ".a43" or ".hex" for
  Intel HEX. ELF files can also be loaded.

  Multiple files can be given on the command line, all are merged before
  the download starts. "-" reads from stdin.

  -i TYPE, --input-format=TYPE
                        input format name (titext, ihex, bin, hex, elf)

Flash erase:
  Multiple --erase options are allowed. It is also possible to use
  address ranges such as 0xf000-0xf0ff or 0xf000/4k.

  NOTE: SegmentA on F2xx is NOT erased with --mass-erase, that must be
  done separately with --erase=0x10c0 or --info-erase".

  -e, --mass-erase      mass erase (clear all flash memory)
  -m, --main-erase      erase main flash memory only
  --info-erase          erase info flash memory only (0x1000-0x10ff)
  -b, --erase-by-file
                        erase only Flash segments where new data is downloaded
  --erase=ADDRESS       selectively erase segment at the specified address or
                        address range

Program flow specifiers:
  All these options work against the file(s) provided on the command
  line. Program flow specifiers default to "-P" if a file is given.

  "-P" usually verifies the programmed data, "-V" adds an additional
  verification through uploading the written data for a 1:1 compare.

  No default action is taken if "-P", "-V" or "-E" is given, say
  specifying only "-V" does a "check by file" of a programmed device
  without programming.

  Don't forget to erase ("-e", "-b" or "-m") before programming flash!

  -E, --erase-check     erase check by file
  -P, --program         program file
  -V, --verify          verify by file
  -U, --upload-by-file
                        upload the memory that is present in the given file(s)

Data upload:
  This can be used to read out the device memory. It is possible to use
  address ranges such as 0xf000-0xf0ff or 0xf000/256, 0xfc00/1k.

  Multiple --upload options are allowed.

  -u ADDRESS, --upload=ADDRESS
                        upload a data block, can be passed multiple times
  -o DESTINATION, --output=DESTINATION
                        write uploaded data to given file
  -f TYPE, --output-format=TYPE
                        output format name (titext, ihex, bin, hex),
```

```
                        default:hex

  Do before exit:
    -x ADDRESS, --execute=ADDRESS
                        start program execution at specified address, might
                        only be useful in conjunction with --wait
    -r, --reset         perform a normal device reset that will start the
                        program that is specified in the reset interrupt
                        vector
    -w, --wait          wait for <ENTER> before closing the port
    --no-close          do not close port on exit

  Communication settings:
    -p PORT, --port=PORT
                        Use com-port
    --invert-test       invert RTS line
    --invert-reset      invert DTR line
    --swap-reset-test   exchenage RST and TEST signals (DTR/RTS)
    --test-on-tx        TEST/TCK signal is muxed on TX line

  BSL settings:
    --no-start          no not use ROM-BSL start pattern on RST+TEST/TCK
    -s SPEED, --speed=SPEED
                        change baud rate (default 9600)
    --password=FILE     transmit password before doing anything else, password
                        is given in given (TI-Text/ihex/etc) file
    --ignore-answer     do not wait for answer to BSL commands
    --control-delay=CONTROL_DELAY
                        set delay in seconds (float) for BSL start pattern
```

msp430.jtag.dco

python -m msp430.jtag.dco [options] frequency:

```
MSP430 clock calibration utility V1.1

This tool can measure the internal oscillator of F1xx, F2xx and F4xx devices,
display the supported frequencies, or run a software FLL to find the settings
for a specified frequency.

The target device has to be connected to the JTAG interface.

Examples:
  See min and max clock speeds:
    dco.py --measure

  Get clock settings for 2.0MHz +/-1%:
    dco.py --tolerance=0.01 2.0e6

  Write clock calibration for 1.5MHz to the information memory at 0x1000:
    dco.py 1.5e6 BCSCTL1@0x1000 DCOCTL@0x1000

Use it at your own risk. No guarantee that the values are correct.

Options:
  -h, --help            show this help message and exit
```

```
  -o FILE, --output=FILE
                        write result to given file
  --dcor                use external resistor
  -d, --debug           print debug messages
  -l LPT, --lpt=LPT     set the parallel port
  -m, --measure         measure min and max clock settings and exit
  -c, --calibrate       Restore calibration values on F2xx devices
  -t TOLERANCE, --tolerance=TOLERANCE
                        set the clock tolerance as factor. e.g. 0.01 means 1%
                        (default=0.005)
  --define              output #defines instead of assignments
  --erase=ERASE         erase flash page at given address. Use with care!
```

msp430.jtag.target

python -m msp430.jtag.target [OPTIONS] [FILE [FILE...]]:

```
Options:
  -h, --help            show this help message and exit
  --debug               print debug messages and tracebacks (development mode)
  -v, --verbose         show more messages (can be given multiple times)
  -q, --quiet           suppress all messages
  --time                measure time
  -S, --progress        show progress while programming
  --help-backend        show help about the different backends
  -l LIBRARY_PATH, --library-path=LIBRARY_PATH
                        search for libMSP430.so or libMSP430mspgcc.so in this
                        place first

  Data input:
    File format is auto detected, unless --input-format is used. Preferred
    file extensions are ".txt" for TI-Text format, ".a43" or ".hex" for
    Intel HEX. ELF files can also be loaded.

    Multiple files can be given on the command line, all are merged before
    the download starts. "-" reads from stdin.

    -i TYPE, --input-format=TYPE
                        input format name (titext, ihex, bin, hex, elf)

  Flash erase:
    Multiple --erase options are allowed. It is also possible to use
    address ranges such as 0xf000-0xf0ff or 0xf000/4k.

    NOTE: SegmentA on F2xx is NOT erased with --mass-erase, that must be
    done separately with --erase=0x10c0 or --info-erase".

    -e, --mass-erase    mass erase (clear all flash memory)
    -m, --main-erase    erase main flash memory only
    --info-erase        erase info flash memory only (0x1000-0x10ff)
    -b, --erase-by-file
                        erase only Flash segments where new data is downloaded
    --erase=ADDRESS     selectively erase segment at the specified address or
                        address range

  Program flow specifiers:
```

```
  All these options work against the file(s) provided on the command
  line. Program flow specifiers default to "-P" if a file is given.

  "-P" usually verifies the programmed data, "-V" adds an additional
  verification through uploading the written data for a 1:1 compare.

  No default action is taken if "-P", "-V" or "-E" is given, say
  specifying only "-V" does a "check by file" of a programmed device
  without programming.

  Don't forget to erase ("-e", "-b" or "-m") before programming flash!

  -E, --erase-check   erase check by file
  -P, --program       program file
  -V, --verify        verify by file
  -U, --upload-by-file
                      upload the memory that is present in the given file(s)

Data upload:
  This can be used to read out the device memory. It is possible to use
  address ranges such as 0xf000-0xf0ff or 0xf000/256, 0xfc00/1k.

  Multiple --upload options are allowed.

  -u ADDRESS, --upload=ADDRESS
                      upload a data block, can be passed multiple times
  -o DESTINATION, --output=DESTINATION
                      write uploaded data to given file
  -f TYPE, --output-format=TYPE
                      output format name (titext, ihex, bin, hex),
                      default:hex

Do before exit:
  -x ADDRESS, --execute=ADDRESS
                      start program execution at specified address, might
                      only be useful in conjunction with --wait
  -r, --reset         perform a normal device reset that will start the
                      program that is specified in the reset interrupt
                      vector
  -w, --wait          wait for <ENTER> before closing the port
  --no-close          do not close port on exit

Connection:
  NOTE: On Windows, use "USB", "TIUSB" or "COM5" etc if using MSP430.dll
  from TI. On other platforms, e.g. Linux, use "/dev/ttyUSB0" etc. if
  using libMSP430.so. If a libMSP430.so is found, it is preferred,
  otherwise libMSP430mspgcc.so is used.

  NOTE: --slowdown > 50 can result in failures for the RAM size auto
  detection (use --ramsize option to fix this). Use the --verbose option
  and watch the outputs. The DCO clock adjustment and thus the Flash
  timing may be inaccurate for large values.

  --backend=BACKEND   select an alternate backend. See --help-backend for
                      more information
  -p PORT, --port=PORT
                      specify an other parallel port or serial port for the
                      USBFET (the later requires libMSP430.so instead of
```

```
                        libMSP430mspgcc.so).  (defaults to "LPT1"
                        ("/dev/parport0" on Linux))
    --spy-bi-wire-jtag  interface is 4 wire on a spy-bi-wire capable device
    --spy-bi-wire       interface is 2 wire on a spy-bi-wire capable device
    --slowdown=MICROSECONDS
                        artificially slow down the communication. Can help
                        with long lines, try values between 1 and 50 (parallel
                        port interface with mspgcc's HIL library only).
                        (experts only)
    -R BYTES, --ramsize=BYTES
                        specify the amount of RAM to be used to program flash
                        (default: auto detected)

  JTAG fuse:
    WARNING: This is not reversible, use with care!  Note: Not supported
    with the simple parallel port adapter (7V source required).",

    --secure            blow JTAG security fuse

  Examples:
    Mass erase and program from file: "/home/lch/python-
    msp430-tools/msp430/jtag/target.py -e firmware.elf" Dump information
    memory: "/home/lch/python-msp430-tools/msp430/jtag/target.py
    --upload=0x1000-0x10ff"
```

msp430.jtag.profile

python -m msp430.jtag.profile [OPTIONS]:

```
Options:
  -h, --help            show this help message and exit
  -v, --verbose         show more messages (can be given multiple times)
  -o FILENAME, --output=FILENAME
                        write result to given file
```

msp430.gdb.target

python -m msp430.gdb.target [OPTIONS] [FILE [FILE...]]:

```
Options:
  -h, --help            show this help message and exit
  --debug               print debug messages and tracebacks (development mode)
  -v, --verbose         show more messages (can be given multiple times)
  -q, --quiet           suppress all messages
  --time                measure time
  -S, --progress        show progress while programming

  Data input:
    File format is auto detected, unless --input-format is used. Preferred
    file extensions are ".txt" for TI-Text format, ".a43" or ".hex" for
    Intel HEX. ELF files can also be loaded.

    Multiple files can be given on the command line, all are merged before
```

```
  the download starts. "-" reads from stdin.

  -i TYPE, --input-format=TYPE
                      input format name (titext, ihex, bin, hex, elf)

Flash erase:
  Multiple --erase options are allowed. It is also possible to use
  address ranges such as 0xf000-0xf0ff or 0xf000/4k.

  NOTE: SegmentA on F2xx is NOT erased with --mass-erase, that must be
  done separately with --erase=0x10c0 or --info-erase".

  -e, --mass-erase    mass erase (clear all flash memory)
  -m, --main-erase    erase main flash memory only
  --info-erase        erase info flash memory only (0x1000-0x10ff)
  -b, --erase-by-file
                      erase only Flash segments where new data is downloaded
  --erase=ADDRESS     selectively erase segment at the specified address or
                      address range

Program flow specifiers:
  All these options work against the file(s) provided on the command
  line. Program flow specifiers default to "-P" if a file is given.

  "-P" usually verifies the programmed data, "-V" adds an additional
  verification through uploading the written data for a 1:1 compare.

  No default action is taken if "-P", "-V" or "-E" is given, say
  specifying only "-V" does a "check by file" of a programmed device
  without programming.

  Don't forget to erase ("-e", "-b" or "-m") before programming flash!

  -E, --erase-check   erase check by file
  -P, --program       program file
  -V, --verify        verify by file
  -U, --upload-by-file
                      upload the memory that is present in the given file(s)

Data upload:
  This can be used to read out the device memory. It is possible to use
  address ranges such as 0xf000-0xf0ff or 0xf000/256, 0xfc00/1k.

  Multiple --upload options are allowed.

  -u ADDRESS, --upload=ADDRESS
                      upload a data block, can be passed multiple times
  -o DESTINATION, --output=DESTINATION
                      write uploaded data to given file
  -f TYPE, --output-format=TYPE
                      output format name (titext, ihex, bin, hex),
                      default:hex

Do before exit:
  -x ADDRESS, --execute=ADDRESS
                      start program execution at specified address, might
                      only be useful in conjunction with --wait
  -r, --reset         perform a normal device reset that will start the
```

```
                        program that is specified in the reset interrupt
                        vector
  -w, --wait            wait for <ENTER> before closing the port
  --no-close            do not close port on exit

Connection:
  -c HOST:PORT, --connect=HOST:PORT
                        TCP/IP host name or ip and port of GDB server
                        (default: localhost:2000)
```

CHAPTER 4

Utilities

msp430.memory.convert

This is a command line tool that can load multiple hex files, combine them and output a hex file of the same or different file type. (run as `python -m msp430.memory.convert`):

```
Usage: convert.py [options] [INPUT...]

Simple hex file conversion tool.

It is also possible to specify multiple input files and create a single,
merged output.

Options:
  -h, --help            show this help message and exit
  -o DESTINATION, --output=DESTINATION
                        write result to given file
  -i TYPE, --input-format=TYPE
                        input format name (titext, ihex, bin, hex, elf)
  -f TYPE, --output-format=TYPE
                        output format name (titext, ihex, bin, hex)
  -d, --debug           print debug messages
```

msp430.memory.compare

Compare two hex files. The files are loaded and a hex dump is compared. The diff between the hex dumps is output (unless the `--html` option is used). The tool also sets the shell exit code so that it could be used in shell/bat scripts.

(run as `python -m msp430.memory.compare`):

```
Usage: compare.py [options] FILE1 FILE2

```

```
Compare tool.

This tool reads binary, ELF or hex input files, creates a hex dump and shows
the differences between the files.


Options:
  -h, --help            show this help message and exit
  -o DESTINATION, --output=DESTINATION
                        write result to given file
  -d, --debug           print debug messages
  -v, --verbose         print more details
  -i TYPE, --input-format=TYPE
                        input format name (titext, ihex, bin, hex, elf)
  --html                create HTML output
```

msp430.memory.generate

Generate hex files filled with some pattern. The pattern can be a counter or a useful MSP430 instruction such as `JMP $` (0x3fff).

(run as `python -m msp430.memory.generate`):

```
Usage:     generate.py [options]

    Test File generator.

    This tool generates a hex file, of given size, ending on address
    0xffff if no start address is given.

Options:
  -h, --help            show this help message and exit
  -o DESTINATION, --output=DESTINATION
                        write result to given file
  -f TYPE, --output-format=TYPE
                        output format name (titext, ihex, bin, hex)
  -l SIZE, --length=SIZE
                        number of bytes to generate
  -s START_ADDRESS, --start-address=START_ADDRESS
                        start address of data generated
  -c, --count           use address as data
  --const=CONST         use given 16 bit number as data (default=0x3fff)
  --random              fill with random numbers
```

msp430.memory.hexdump

Show hex dump of files. Note that the same can be achieved with `msp430.memory.convert -f hex`.

(run as `python -m msp430.memory.hexdump`):

```
Usage: hexdump.py [options] [SOURCE...]

Hexdump tool.
```

```
This tool generates hex dumps from binary, ELF or hex input files.

What is dumped?
- Intel hex and TI-Text: only data
- ELF: only segments that are programmed
- binary: complete file, address column is byte offset in file

Options:
  -h, --help            show this help message and exit
  -o DESTINATION, --output=DESTINATION
                        write result to given file
  --debug               print debug messages
  -v, --verbose         print more details
  -i TYPE, --input-format=TYPE
                        input format name (titext, ihex, bin, hex, elf)
```

CHAPTER 5

Shell utilities

The module `msp430.shell` provides some useful scripts for the shell.

msp430.shell.command

This tool emulates a number of shell utilities. The idea is that makefiles or similar build tools can use these commands to be OS independent (so that the same set of commands works on Windows, MacOS, GNU/Linux, etc.).

Command collection:

- `cat` Show file contents.
- `cp` Copy files.
- `expand` Expand shell patterns ("*.c", "?" etc.).
- `false` Simply return exit code 1
- `list` This text.
- `mkdir` Create directories.
- `mv` Move/rename files.
- `rm` Delete files/directories.
- `touch` Update file date, create file.
- `true` Simply return exit code 0

More help with "command.py COMMAND –help"

Example:

```
python -m msp430.shell.command rm -f no_longer_needed.txt
python -m msp430.shell.command cp src.txt dst.txt
```

msp430.shell.watch

This tool watches one or multiple files for changes. When a change on one file is detected it runs a given command. This could be used e.g. to automatically trigger a download when a hex file has changed or trigger compilation when one of the source files has changed.

Usage: watch.py FILENAME [FILENAME...] –execute "some/program/ –"

Options:

-h, --help show this help message and exit

-x COMMAND, --execute=COMMAND run this command when watched file(s) changed, – is replaced by FILENAME(s)

CHAPTER 6

Assembler

The module `msp430.asm` provides an assembler for MSP430 and MSP430X CPUs. There is also a disassembler.

Additionally a (almost C compatible) preprocessor is provided.

The tutorial and command line sections cover the command line tools and how to use them. The API section is about the internals of the tools and may be interesting to developers that extended the tools or use them as a library.

Also available is a Forth cross compiler that can translate Forth programs to MSP430 assembler.

Tutorial

A simple example

The assembler `msp430.asm.as` reads source files (`*.S`) and creates object files (`*.o4`). Multiple object files are then linked together and a binary is created that can be downloaded to the MCU.

For example, `led.S`:

```
; Test program for msp430.asm.as and msp430.asm.ld
;
; This one toggles the pin P1.1. This is like the LED flashing example that
; comes preprogrammed on some of the eval boards from TI.

.text
        ; entry point after device reset
RESET:  mov     #0x5a80, &0x120         ; disable WDT
        bis.b   #1, &0x22               ; set pin to output

        ; loop toggling the pin and then doing a delay
.L1:    xor.b   #1, &0x21               ; toggle pin
        mov     #0xc350, R15            ; init delay loop
.L2:    dec     R15                     ; count down
        jnz     .L2                     ; jump while counter is not zero
```

```
        jmp     .L1                     ; loop the toggling part


; set the reset vector (and all the others) to the program start
.section .vectors
        .word  RESET, RESET, RESET, RESET, RESET, RESET, RESET, RESET
        .word  RESET, RESET, RESET, RESET, RESET, RESET, RESET
        .word  RESET                    ; reset vector
```

Assemble, link:

```
python -m msp430.asm.as led.S -o led.o4
python -m msp430.asm.ld --mcu MSP430G2211 led.o4 -o led.titext
```

Download

There are several ways to get a program into a MSP430.

Boot Strap Loader (BSL), Serial Using a serial connection and some ROM code in the MSP430 it is possible to read and write memory, including Flash.

Not all devices support BSL (e.g. the smaller value line (G2) and F2 devices)

Command example (F1x, F2x, F4x):

```
python -m msp430.bsl.target -e led.titext
```

Command example (F5x, F6x):

```
python -m msp430.bsl5.uart -e led.titext
```

Boot Strap Loader (BSL), USB HID Some MSP430 have a built in USB controller and they also support downloading through USB.

Command example:

```
python -m msp430.bsl5.hid -e led.titext
```

JTAG, 4-wire This interface gives access to the internals of the CPU so that it not only can be used to up and download memory, it is also possible to set breakpoints, single step and more debugging.

Some devices have shared GPIO pins, so that a TEST pin switches the function from normal IO pin to JTAG.

Command example:

```
python -m msp430.jtag.target -e led.titext
```

JTAG, spy-bi-wire This is a variation of the JTAG interface that only requires two pins and does not occupy GPIO pins. The same signals as in a 4-wire connection are serialized and transmitted over these two lines. This means that the maximum speed of the spy-bi-wire interface is slower than the 4-wire interface.

Many new MSP430 support this interface (not F1, F4).

Command example:

```
python -m msp430.jtag.target --spy-bi-wire -e led.titext
```

The python-msp430-tools also support downloading via remote-GDB-protocol. If a GDB server is running (same machine or a different one), `msp430.gdb.target` can be used. GDB servers are msp430-gdbproxy or mspdebug

Notes for JTAG

Windows The MSP430.dll can be downloaded from TI. With this installed, USB and parallel port adapters can be used with the `msp430.jtag.target` tool.

Linux / Others There is no (recent) MSP430.dll available.

USB JTAG adapters can be used with the tool mspdebug (also includes debug support).

Parallel port adapters can be used with MSP430mspgcc (no debug support).

Command example (Launchpad or ez430-rf2500 kits):

```
mspdebug rf2500 "prog led.titext" exit
```

Installing header files

The example above directly used the addresses of the peripheral modules - this is not comfortable. It is easily possible to use the header files from TI as a C preprocessor (`cpp`) is included, however the header files itself are not.

Downloading header files

A download and extraction script is located in the directory `msp430/asm/includes`. When executed (`python fetch.py`) it will download the `msp430mcu` archive from http://mspgcc.sf.net. Once downloaded, the files are extracted to a subdirectory called `upstream`.

The include and include/upstream directories are part of the search path for cpp. Files in these directories are found automatically.

Note: The file name that is downloaded is currently hard coded in the script. It may make sense to check the site online for newer files.

Using the msp430mcu package

On many GNU/Linux systems it is possible to install the package `msp430mcu` though the systems package management.

Debian/Ubuntu: apt://msp430mcu

Note: The header files from the package are currently not found automatically. The user has to provide the location with the `-I` parameter of cpp.

More Examples

A number of examples can be found in the `examples/asm` directory of the `python-msp430-tools` distribution.

Command line tools

msp43.asm.as

An assembler for MSP430(X).

> **Warning:** This tool is currently in an experimental stage. Is has been used to successfully create simple programs but it is not broadly tested.

Command line

Usage: as.py [options]

Options:

-h, --help	show this help message and exit
-x, --msp430x	Enable MSP430X instruction set
-o FILE, --outfile=FILE	name of the object file
--filename=FILE	Use this filename for input (useful when source is passed on stdin)
-v, --verbose	print status messages to stderr
--debug	print debug messages to stderr
-i, --instructions	Show list of supported instructions and exit (see also -x)

Supported directives

The instruction set as documented in the MSP430 family guides is supported as well as the following pseudo instructions:

- `.ASCII` Insert the given text as bytes
- `.ASCIIZ` Insert the given text as bytes, append null byte
- `.BSS` Select `.bss` section for output
- `.BYTE` Insert the given 8 bit values
- `.DATA` Select `.data` section for output
- `.EVEN` Align address pointer to an even address
- `.LONG` Insert the given 32 bit values
- `.SECTION` Select named section for output
- `.SET` Define a symbol with a value (can be used at link time)
- `.SKIP` Skip the given amount of bytes
- `.TEXT` Select `.text` section for output
- `.WEAKALIAS` Create alias for label in case it is not defined directly
- `.WORD` Insert the given 16 bit values

msp430.asm.ld

The linker processes one or multiple `.o4` files (the output from `as`) and creates a binary file that can be downloaded to a target.

Command line

Usage: ld.py [options] [FILE...]|-]

If no input files are specified data is read from stdin. Output is in "TI-Text" format.

Options:

-h, --help show this help message and exit

-o FILE, --outfile=FILE name of the resulting binary (TI-Text)

-T FILE, --segmentfile=FILE linker definition file

-m MCU, --mcu=MCU name of the MCU (used to load memory map)

--mapfile=FILE write map file

-v, --verbose print status messages

--debug print debug messages

msp430.asm.cpp

This is an (almost C compatible) preprocessor. It can work with macros (`#define`) and evaluate arithmetic expressions.

Supported directives are:

- `#define` Define a value or function like macro
- `#include` Read and insert given file
- `#if` Conditional compilation is predicate is true. `defined` is also supported.
- `#ifdef` Conditional compilation if given symbol is defined
- `#ifndef` Conditional compilation if given symbol is not defined
- `#else` For the inverse of `#if`/`#ifdef`/`#ifndef`
- `#endif` Finish `#if`/`#ifdef`/`#ifndef` / `#else`
- `#undef` Forget about the definition of a macro

Command line

Usage: cpp.py [options]

Options:

-h, --help show this help message and exit

-o FILE, --outfile=FILE name of the object file

-p FILE, --preload=FILE process this file first. its output is discarded but definitions are kept.

-v, --verbose print status messages

--debug print debug messages to stdout

-D SYMVALUE, --define=SYMVALUE define symbol

-I PATH, --include-path=PATH Add directory to the search path list for includes

To define symbols, use `-D SYMBOL=VALUE` respectively `--define SYMBOL=VALUE`

msp430.asm.disassemble

This is a disassembler for MSP430(X) code. It outputs an annotated listing. Each jump target is assigned an automatic label and a newline is inserted after each non conditional jump to make reading the source easier.

The disassembler currently has no knowledge about the memory map or usage of memory. Therefore it disassembles just anything, even if it is not code.

Provided with a symbol file, it can insert the names and named bits of accessed peripherals (for details see `msp430/asm/definitions/F1xx.txt`).

Warning: This tool is currently in an experimental stage. It is not fully tested and especially the cycle counts are not verified.

Command line

Usage: disassemble.py [options] [SOURCE...]

MSP430(X) disassembler.

Options:

-h, --help show this help message and exit

-o DESTINATION, --output=DESTINATION write result to given file

--debug print debug messages

-v, --verbose print more details

-i TYPE, --input-format=TYPE input format name (titext, ihex, bin, hex, elf)

-x, --msp430x Enable MSP430X instruction set

--source omit hex dump, just output assembler source

--symbols=NAME read register names for given architecture (e.g. F1xx)

Forth Cross Compiler

The package also includes a limited Forth like language cross compiler.

Warning: This feature is under development.

Attention: Documentation is incomplete. Also reading the source is recommended.

Traditional methods to use Forth on a different target system involve meta-compilers and the assembler for the target is often implemented in Forth itself. Not this one.

The idea of this tool chain is to cross compile Forth into assembler for the MSP430 and then use the normal assembler and linker. This means that other assembler (or C modules etc.) can be combined in one program.

```
led.forth -> led.S -> led.o4     --+--> led.titext
          intvec.S -> intvec.o4  --/
         startup.S -> startup.o4 --/
```

Available Words

A list of supported words is available here:

Availability of words depends on their definition. There are words that can only be executed on the host. These words can not be used within definitions that are cross compiled. In contrast, `CODE` words are only allowed within definitions that are cross compiled. Normal definitions using `:` can run on host and target, unless they depend on words described before. In that case the restrictions of that type applies.

Command line tools

msp430.asm.forth

This is itself a Forth interpreter and is used to do the conversion into an assembler file for the MSP430.

msp430.asm.h2forth

This tool can be used to convert C header to a Forth file. Each `#define` will be turned into a `CONSTANT`. It's main purpose is to get access to the peripheral an bit definitions from the TI header files.

Cross compilation

The file given to `msp430.asm.forth` is executed on the host.

Only words used in the program for the target are translated. This makes the use of libraries simple - it does not matter how many words are defined or used by the program on the host.

There are three dictionaries in the host interpreter.

- built-in name space: The words here are implemented in Python (the language the host interpreter is written in). These can not be cross compiled (unless they are provided in the target dictionary).
- normal name space: normal `:/;` definitions go here. The words here can be used on the host as well as on the target.
- target name space: `CODE` definitions go here. These words can be referenced by the host but they can not be executed (they won't do the expected).

All three dictionaries are available to the host but cross compiled words must exist in one of the later two.

Cross compilation of normal words (`:`) The words are already translated into a list of references by the host interpreter. They can be output 1:1 as list of words for the target (exceptions are branch distances, they are multiplied by two).

Cross compilation of `CODE` words These words are executed to get the cross compilation. So each `CODE` word simply outputs assembler code. Forth words must include the code for `NEXT`.

This allows that `CODE` word definitions can also be used to generate helper functions for other assembler parts.

Cross compilation of `INTERRUPT` words A special assembler start code is included and the exit functionality is handled specially. The function itself is translated the same way a normal word is.

MSP430 specific features

- `VARIABLE` and `VALUE` are supported, they allocate a 16 bit variable in RAM (`.data` segment).
- The word `INTERRUPT` defines a new interrupt function. It takes the vector number from the stack and it expects that a function name follows. It creates symbols `vector_NN` which have to be manually added to the interrupt vector table. Interrupt declarations end with the word `END-INTERRUPT`.
- Words defined using `CODE`/`END-CODE` are executed when cross compiling. They are expected to write out MSP430 assembler.
- `INCLUDE` loads and executes the given file. There is a search path that can be influenced with the `-I` command line option. Some files are built-in, e.g. `core.forth` is part of the package.
- Escapes are decoded in strings. e.g. `." \n"` outputs a newline.
- The `DEPENDS-ON` word can be used in `CODE` words. It adds the given word as dependency so that it is also cross compiled. This is useful when the assembler in the `CODE` words wants to use some shared code.

Internals

- SP is used as data stack.
- Interrupts and other asm/C functions called so then use the data stack.
- The return stack and instruction pointers are also kept in registers. All other registers can be used in Forth words (see generated assembler for register usage).

> **Caution:** There is no stack depth checking implemented. Not maintaining the stack balance usually ends up in executing random parts or the program (A.K.A. "crash").

Limitations

The current language is not quite Forth. Some important words such as `DOES>` are missing.

- Not ANS Forth compliant.
- The Forth interpreter only reads in words. There is no access to the characters of the source file and whitespace and newlines are discarded. So the strings `." Hello World"` and `." Hello World"` are identical. Some words such as `."` partially emulate byte wise access by processing each word by character (that's why the closing `"` is detected in the previous examples).
- `\` Comments are not supported due to the limitation discussed above.
- `*`, `/`, `/MOD` are currently not available on the target.
- `CREATE`, `ALLOT` have limited functionality.
- `ERASE` was named `ZERO` due to a naming conflict with a bit of the MSP430 Flash module.

- No double precision math.
- Input/output functions are missing.
- There are more...

Thanks

A number of core Forth words that are implemented in Forth itself are taken from JonesForth (Licensed as Public Domain) which is a well documented experimental Forth for x386 computers (used in `msp430/asm/forth/__init__.forth`).

API Documentation

This section is about the internals of the `msp430.asm` module. It may be interesting for developers that work on this module or who are interested in using the functions the module provides in their own code.

Object file format

The file format of `.o4` files is a bit unusual. It actually contains something that could be labeled as (specialized) Forth code. So the linker is some sort of Forth interpreter. This has the advantage that the object files can be debugged without any special tools, just a text editor. It also makes the format quite universal; it could produce binaries for all sorts of CPUs (single special case: the directive `JMP` is MSP430 specific).

A list of supported words can be found in the following document:

For more details also take a look at the sources of `ld.py`.

MCU Definition file format

MCU memory definitions can be provided in a file with Forth like syntax.

A list of supported words can be found in the following document:

For more details also take a look at the sources of `mcu_definition_parser.py`.

Modules

msp430.asm.as

This module implements the MSP430(X) assembler. When the module is executed (e.g. using `python -m msp430.asm.as`), it acts as a command line tool.

class `msp430.asm.as.`**MSP430Assembler**

__init__(*msp430x=False*, *debug=False*)

Parameters

- **msp430x** – Set to true to enable MSP430X instruction set.
- **debug** – When set to true dump some internal data so sys.stderr while compiling.

Create an instance of the assembler.

assemble(*f*, *filename=None*, *output=sys.stdout*)

Parameters

- **f** – A file like object that supports iterating over lines.
- **filename** – An optional string that is used in error messages.
- **output** – File like object used to write the object code to.

This method takes assembler source and transforms it to object code that can be forwarded to the linker.

exception msp430.asm.as.**AssemblerError**

This instances of this class are raised by the MSP430Assembler in case of errors in the source. It may be annotated with the source filename and line number where the error occurred.

filename

line

msp430.asm.ld

This module implements the linker. When the module is executed (e.g. using python -m msp430.asm.ld), it acts as a command line tool.

class msp430.asm.ld.**Segment**

__init__(*name*, *start_address=None*, *end_address=None*, *align=True*, *programmable=False*, *little_endian=True*, *parent=None*, *mirror_of=None*)

__getitem__(*segment_name*)

Parameters **segment_name** – name of an sub segment.

Raises **KeyError** – when no segment with given name is found

Easy access to subsegment by name.

sort_subsegments(*by_address=False*)

Parameters **by_address** – Sort by address if true, otherwise sort by name.

Sort list of subsegments either by order of definition or by order of start address.

clear()

Clear data. Recursively with all subsegments. Segments are not removed, just their data.

__len__()

Get the number of data bytes contained in the segment.

__cmp__(*other*)

Compare function that allows to sort segments by their start_address.

__lt__(*other*)

Compare function that allows to sort segments by their start_address.

print_tree(*output*, *indent=''*, *hide_empty=False*)

Parameters

- **output** – a file like object (supporting write)
- **indent** – a prefix put before each line.
- **hide_empty** – when set to true omit empty segments (no data) in output.

Output segment and subsegments.

shrink_to_fit(*address=None*)
Modify start- and end_address of segment to fit the data that it contains. Recursively applied to the tree of segments. Typically called with `address=None`.

write_8bit(*value*)

Parameters **value** – an integer (8 significant bits)

Write one byte.

write_16bit(*value*)

Parameters **value** – an integer (16 significant bits)

Write two bytes. Order in memory depends on endianness of segment.

write_32bit(*value*)

Parameters **value** – an integer (32 significant bits)

Write four bytes. Order in memory depends on endianness of segment.

class `msp430.asm.ld.`**Linker**

__init__(*instructions*)

Parameters **instructions** – list of directives for the linker

Initialize a linker instance. The given instructions are essentially what is read from a `.o4` file as sequence of words.

segments_from_definition(*segment_definitions*)

Parameters **segment_definitions** – dictionary describing the memory map

This sets the memory map used for linking. See `mcu_definition_parser` for a way to load this description.

update_mirrored_segments()
Called before writing the final output. In case the memory map contains segments that mirror the contents of other segments, they are updated. This is typically used for `.data_init` which contains the initial values that are copied by startup code to the `.data` segment in RAM.

pass_one()
Run the linkers 1st pass. It iterates through the instructions and places the data into segments.

pass_two()
Run the linkers 2nd pass. It iterates through the instructions and finds all the labels and saves their position.

pass_three()
Run the linkers 3rd pass. It iterates through the instructions and creates the final binary with all known labels set to their target address.

exception `msp430.asm.ld.`**LinkError**
Exception object raised when errors during linking occur. May be annotated with the location of the line within the original source file causing the error.

filename

lineno

column

msp430.asm.cpp

This module implements the preprocessor. When the module is executed (e.g. using `python -m msp430.asm.cpp`), it acts as a command line tool.

msp430.asm.cpp.**line_joiner**(*next_line*)
Given an iterator for lines, yield lines. It joins consecutive lines with the continuation marker (\\) to a single line.

class msp430.asm.cpp.**AnnoatatedLineWriter**
This object is used by the preprocessor to write out the preprocessed text. It adds notes in the form `#line <line> "<filename>"`. These notes are used by the assembler to know where a source line originally came from (as preprocessed text may contain additional lines etc.)

__init__(*output*, *filename*)

Parameters

- **output** – file like object to write to
- **filename** – the filename used in the notes

write(*lineno*, *text*)

Parameters

- **linno** – line number being written
- **text** – the actual contents of the line

class msp430.asm.cpp.**Preprocessor**

preprocess(*infile*, *outfile*, *filename*)

Parameters

- **infile** – file like object to read from
- **outfile** – file like object to write to
- **filename** – original file name of the input (infile)

This runs the preprocessor over the given input.

exception msp430.asm.cpp.**PreprocessorError**
Exception object raised when errors during preprocessing occur.

msp430.asm.disassemble

This module implements the disassembler. When the module is executed (e.g. using `python -m msp430.asm.disassemble`), it acts as a command line tool.

class msp430.asm.disassemble.**MSP430Disassembler**

__init__(*memory*, *msp430x=False*, *named_symbols=None*)

Parameters

- **memory** – A msp43.memory.Memory instance containing the binary.
- **msp430x** – Set to true to enable MSP430X instruction set.

- **named_symbols** – An (optional) instance of `NamedSymbols` which is used to label peripherals and bits.

Initialize the disassembler with data.

disassemble(*output*, *source_only=False*)

Parameters

- **output** – A file like object used for the resulting text.
- **source_only** – When set to true, the address and data columns are omitted from the output.

Run the disassembler, result is written to output.

msp430.asm.rpn

This module implements the an RPN calculator. The calculator can be tested by executing the module (e.g. using `python -m msp430.asm.rpn`).

class msp430.asm.rpn.**Word**(*unicode*)

This class is used to wrap words so that their source location can be tracked. This is useful for error messages.

__new__(*cls*, *word*, *filename*, *lineno*, *text*)

Parameters

- **cls** – Class for __new__
- **word** – The word (unicode)
- **filename** (*unicode or None*) – Filename where the word was read from.
- **lineno** (*int or None*) – Line number within the file.
- **text** (*unicode or None*) – The complete line (or context).

Create new instance with a word that was read from given location.

class msp430.asm.rpn.**RPN**

An RPN calculator. It provides a data stack and implements a number of basic operations (arithmetical and stack)

interpret(*next_word*)

Parameters **next_word** – A function return the next word from input when called.

Interpret a sequence of words given by the iterator next_word.

msp430.asm.rpn.**annotated_words**(*sequence*, *filename=None*, *lineno=None*, *offset=None*, *text=None*)

Create an generator for *Word*, all annotated with the given information.

msp430.asm.rpn.**words_in_string**(*data*, *name='<string>'*)

Parameters

- **data** – String with (lines) of text.
- **name** – Optional name, used in error messages.

Create a generator for annotated *Word* in string (`splitlines()` is used).

msp430.asm.rpn.**words_in_file**(*filename*)

Parameters **filename** – Name of a file to read from.

Create a generator for annotated *Word* read from file given by name.

msp430.asm.rpn.**rpn_function**(*code*)

Parameters **code** – A string in RPN notation

Returns A Python function.

Return a wrapper - a function that evaluates the given RPN code when called. This can be used to insert functions implemented as RPN into the name space.

msp430.asm.rpn.**word**(*name*)
Function decorator used to tag methods that will be visible in the RPN built-in name space.

msp430.asm.rpn.**val**(*words*, *stack=[]*, *namespace={}*)

Parameters

- **words** – Sequence of words.
- **stack** – Optional initial stack.
- **namespace** – Optional namespace.

Returns The top element from the stack

Evaluate sequence of words.

msp430.asm.rpn.**python_function**(*code*, *namespace={}*)

Parameters

- **code** – RPN code to execute.
- **namespace** – Optional namespace.

Returns A python function that executes `code` when called.

Create a Python function that will execute given code when called. All parameters given to the Python function will be placed on the stack and the top of stack will be returned.

msp430.asm.rpn.**interpreter_loop**(*namespace={}*, *debug=False*)
Run an interactive loop. Can be used as calculator.

exception msp430.asm.rpn.**RPNError**
Exception type used for errors when parsing or executing RPN code. It may be annotated with the source position where the word causing the error came from.

filename

lineno

offset

text

msp430.asm.peripherals

This module implements a parser for a file format describing the peripherals and their bits of a MCU. The module can be executed (e.g. using `python -m msp430.asm.peripherals`) to test definition files.

class msp430.asm.peripherals.**SymbolDefinitions**(*msp430.asm.rpn.RPN*)
This class implements the parser and keeps the result. It inherits from `RPN`.

msp430.asm.peripherals.**load_symbols**(*filename*)

Parameters **filename** – Load symbols from a file named like this.

Returns instance of *SymbolDefinitions*

Load definitions from a file of given name.

msp430.asm.peripherals.**load_internal**(*name*)

Parameters **name** – Name of an internal file.

Returns instance of *SymbolDefinitions*

This tries to load internal data (using pkgutil).

exception msp430.asm.peripherals.**SymbolError**
Exception object used for errors in the definition file.

msp430.asm.mcu_definition_parser

This module implements the a parser for files describing the memory map of a CPU. The module can be executed (e.g. using python -m msp430.asm.mcu_definition_parser) to test definition files.

class msp430.asm.mcu_definition_parser.**MCUDefintitions**(*msp430.asm.rpn.RPN*)
This class implements the parser and keeps the result. It inherits from *msp430.asm.rpn.RPN*. Loaded definitions may contain the memory maps of many MCUs and also partial maps (that may depend on each other).

msp430.asm.mcu_definition_parser.**load_from_file**(*filename*)

Parameters **filename** – Load definitions from file of given name.

Returns instance of *MCUDefintitions*

msp430.asm.mcu_definition_parser.**load_internal**()

Returns instance of *MCUDefintitions*

Load internal list. The default list is included in msp430/asm/definitions/msp430-mcu-list.txt

msp430.asm.mcu_definition_parser.**expand_definition**(*memory_maps*, *name*)

Parameters

- **memory_maps** (MCUDefintitions) – Memory map descriptions.
- **name** – Name of an MCU that should be extracted

Returns Dictionary with recursively expanded memory map.

Return the memory map of a specific MCU. If the definition depends on others, it is expanded so that a single, complete description is returned.

msp430.asm.infix2postfix

This module implements a converter that can translate infix (arithmetical) notation to postfix notation (RPN). It is used by the preprocessor and assembler when evaluating expressions.

msp430.asm.infix2postfix.**infix2postfix**(*expression*, *variable_prefix=''*, *scanner=Scanner*, *precedence=default_precedence*)

Parameters

- **expression** – Input string in infix notation.
- **variable_prefix** – A string that is prepended to symbols found in the expression.
- **scanner** – The class that is used to parse the expression.

- **precedence** – A dictionary returning the priority given an operator as key.

Returns A string with the expression in postfix notation.

msp430.asm.infix2postfix.**convert_precedence_list**(*precedence_list*)

Parameters **precedence_list** – A list of lists that defines operator priorities.

Returns A dictionary mapping operators to priorities.

Input will look like this:

```
default_precedence_list = [
        # lowest precedence
        ['or'],
        ['and'],
        ['not'],
        ['<', '<=', '>', '>=', '==', '!='],
        ['|', '^', '&'],
        ['<<', '>>'],
        ['+', '-'],
        ['*', '/', '%'],
        ['~', 'neg', '0 +'],
        ['(', ')'],
        # highest precedence
    ]
```

CHAPTER 7

Internals

Target APIs

This is the API description for the target tools (up- and download of data to MCU using different interfaces). See also the individual tools above and *Commandline Tools*.

The `Target` class defines an interface that is implemented by all the Target connections described here. This interface could be used for example in custom programming tools or testing equipment in manufacturing.

Target base class

`msp430.target.`**`identify_device`**(*device_id*, *bsl_version*)

Parameters

- **`device_id`** (*int*) – 16 bit number identifying the device
- **`bsl_version`** (*int*) – 16 bit number identifying the ROM-BSL version

Returns *F1x*, *F2x* or *F4x*

Identification of F1x, F2x, F4x devices.

class `msp430.target.`**`Target`**(*object*)

This class implements a high level interface to targets. It also provides common code for command line tools so that e.g. the JTAG and BSL tools have a similar set of options.

`memory_read`(*address*, *length*)

Read from memory

`memory_write`(*address*, *data*)

Write to memory.

`mass_erase`()

Clear all Flash memory.

main_erase()
Clear main Flash memory (excl. infomem).

erase(*address*)
Erase Flash segment containing the given address.

execute(*address*)
Start executing code on the target.

version()
The 16 bytes of the ROM that contain chip and BSL info are returned.

reset()
Reset the device.

Additional methods that can be override in subclass.

open_connection()
Open the connection.

def close_connection()
Close the connection.

High level functions.

flash_segment_size(*address*)

Parameters **address** – Address within MCU Flash memory

Returns segment size in bytes

Determine the Flasg segment size for a given address.

get_mcu_family()

Returns *F1x*, *F2x* or *F4x*

Get MCU family. It calls `Version()` to read from the device.

erase_infomem()
Erase all infomem segments of the device.

upload(*start*, *end*)

Parameters

- **start** – Start address of memory range (inclusive)
- **end** – End address of memory range (inclusive)

Upload given memory range and store it in `upload_data`.

def upload_by_file()
Upload memory areas and store it in `upload_data`. The ranges uploaded are determined by `download_data`.

program_file(*download_data=None*)

Parameters **download_data** – If not None, download this. Otherwise `download_data` is used.

Download data from `download_data` or the optional parameter.

verify_by_file()
Upload and compare to `download_data`.

Raises an exception when data differs.

erase_check_by_file()
Upload address ranges used in download_data and check if memory is erased (0xff). Raises an exception if not all memory is cleared.

erase_by_file()
Erase Flash segments that will be used by the data in self.download_data.

Command line interface helper functions.

create_option_parser()

Returns an optparse.OptionParser instance.

Create an option parser, populated with a basic set of options for reading and writing files, upload, download and erase options.

parse_args()
Parse sys.argv now.

main()
Entry point for command line tools.

add_extra_options()
The user class can add items to parser.

parse_extra_options()
The user class can process options he added.

Actions list. This list is build and then processed in the command line tools.

add_action(*function*, **args*, ***kwargs*)
Store a function to be called and parameters in the list of actions.

remove_action(*function*)
Remove a function from the list of actions.

do_the_work()
Process the list of actions

exception msp430.target.**UnsupportedMCUFamily**
Exception that may be raised by *Target* when the connected MCU is not compatible.

msp430.target.**F1x**

msp430.target.**F2x**

msp430.target.**F4x**

BSL Target

Interface to the BSL in F1x, F2x, F4x.

class msp430.bsl.bsl.**BSL**(*object*)
Implement low-level BSL commands as well as high level commands.

MAXSIZE
Maximum size of a block that can be read or written using low level commands.

checksum(*data*)

Parameters **data** – A byte string with data

Returns 16 checksum (int)

Calculate the 16 XOR checksum used by the BSL over given data.

Low level functions.

BSL_TXBLK(*address*, *data*)

Parameters

- **address** – Start address of block
- **data** – Contents (byte string)

Write given data to target. Size of data must be smaller than *MAXSIZE*

BSL_RXBLK(*address*, *length*)

Parameters

- **address** – Start address of block
- **length** – Size of block to read

Returns uploaded data (byte string)

Read data from target. Size of data must be smaller than *MAXSIZE*

BSL_MERAS()
Execute the mass erase command.

BSL_ERASE(*address*, *option=0xa502*)

Parameters

- **address** – Address within the segment to erase.
- **option** – FCTL1 settings.

Execute a segment or main-erase command.

BSL_CHANGEBAUD(*bcsctl*, *multiply*)

Parameters

- **bcsctl** – BCSCTL1 settings for desired baud rate
- **multiply** – Baud rate multiplier (compared to 9600)

Change the baud rate.

BSL_SETMEMOFFSET(*address_hi_bits*)

Parameters **address_hi_bits** – Bits 16..19.

For devices with >64kB address space, set the high bits of all addresses for BSL_TXBLK, BSL_RXBLK and BSL_LOADPC.

BSL_LOADPC(*address*)

Parameters **address** – The address to jump to.

Start executing code at given address. There is no feedback if the jump was successful.

BSL_TXPWORD(*password*)
Transmit password to get access to protected functions.

BSL_TXVERSION()
Read device and BSL info (byte string of length 16). Older ROM-BSL do not support this command.

High level functions.

check_extended()
Check if device has address space >64kB (BSL_SETMEMOFFSET needs to be used).

See also *msp430.target.Target* for high level functions

version()
Read version. It tries *BSL_TXVERSION()* and if that fails *BSL_RXBLK()* from address 0x0ff0. The later only word if the device has been unlocked (password transmitted).

reset()
Try to reset the device. This is done by a write to the WDT module, setting it for a reset within a few milliseconds.

exception msp430.bsl.bsl.**BSLException**(*Exception*)
Errors from the slave.

exception msp430.bsl.bsl.**BSLTimeout**(*BSLException*)
Got no answer from slave within time.

exception msp430.bsl.bsl.**BSLError**(*BSLException*)
Command execution failed.

msp430.bsl.target

This module can be executed as command line tool (`python -m msp430.bsl.target`). It implements the BSL target tool.

class msp430.bsl.target.**SerialBSL**(*bsl.BSL*)
Implement the serial port access.

open(*port=0*, *baudrate=9600*, *ignore_answer=False*)

Parameters

- **port** – Port name or number
- **ignore_answer** – If set to true enables a mode where answers are not read.

Open given serial port (pySerial).

When `ignore_answer` is enabled, no answer will be read for any command. Instead a small delay will be made. This can be used for targets where only the TX line is connected. However no upload and or verification of downloaded data is possible.

close()
Close serial port

bsl(*cmd*, *message=''*, *expect=None*)

Parameters

- **cmd** – Command number to send
- **message** – Byte string with data to send.
- **expect** – The number of bytes expected in a data reply or None to disable check.

Returns None on success with simple answers or a byte string for data answers.

Raises

- ***bsl.BSLError*** – In case of unknown commands, broken packets
- ***bsl.BSLTimeout*** – If no answer was received within time

Implement the low level transmit-receive operation for BSL commands over the serial port. The `cmd` is filled in the data header, `message` appended and the checksum calculated for the sent packet.

Received answers are checked. If `expect` is set a data reply must be received and its length must match the given number, otherwise a `bsl.BSLError` is raised.

set_RST(*level=True*)

Parameters **level** – Signal level

Set the RST pin to given level

set_TEST(*level=True*)

Parameters **level** – Signal level

Set the TEST or TCK pin to given level

set_baudrate(*baudrate*)

Parameters **baudrate** – New speed (e.g. 38400)

Send the change baud rate command and if successful change the baud rate of the serial port to the same value.

class `msp430.bsl.target.`**SerialBSLTarget**(*SerialBSL*, *msp430.target.Target*)

Combine the serial BSL backend and the common target code.

add_extra_options()

Adds extra options to configure the serial port and the usage of the control lines for RST and TEST/TCK.

parse_extra_options()

Used to output additional tool version info.

close_connection()

Close serial port.

open_connection()

Open serial port, using the options from the command line (in `options`). This will also execute the mass erase command and/or transmit the password so that executing other actions is possible.

This is also the place to download replacement BSL or the patch.

BSL_TXBLK()

Override the block write function to activate the patch if needed.

BSL_RXBLK()

Override the block read function to activate the patch if needed.

reset()

Override the reset methods to use the RST control line signal (instead of the WDT access)

BSL5 Target

Interface to the BSL in F5x and F6x devices. UART and USB-HID are supported.

class `msp430.bsl5.bsl5.`**BSL5**

check_answer(*data*)

Parameters **data** – the data received from the target

Returns None

Raises BSL5Error with the corresponding message if `data` contained an error code.

Note that the length for the following memory read/write functions is limited by the packet size of the interface (USB-HID, UART).

BSL_RX_DATA_BLOCK(*address*, *data*)

Parameters

- **address** – Location in target memory
- **data** – Byte string with data to write

Write given data to target memory.

BSL_RX_DATA_BLOCK_FAST(*address*, *data*)

Parameters

- **address** – Location in target memory
- **data** – Byte string with data to write

Write given data to target memory. The target will not perform any checks and no respons is sent back.

BSL_TX_DATA_BLOCK(*address*, *length*)

Parameters

- **address** – Location in target memory.
- **length** – Number of bytes to read.

Returns Byte string with memory contents.

Read from target memory.

def BSL_MASS_ERASE()

Execute the mass erase command.

def BSL_ERASE_SEGMENT(address)

param address An address within the segment to erase.

Erase a single Flash memory segment.

BSL_LOAD_PC(*address*)

Parameters **address** – Location in target memory.

Start executing at given address. There is no check if the command is successful as the execution starts immediately.

BSL_RX_PASSWORD(*password*)

Parameters **password** – Byte string with password (32 bytes)

Transmit the password in order to unlock protected function of the BSL.

BSL_VERSION()

Returns A tuple with 5 numbers.

The return value contains the following numbers:

•BSL vendor information

•Command interpreter version

•API version

•Peripheral interface version

BSL_BUFFER_SIZE()

Returns The maximal supported buffer size from the BSL.

BSL_LOCK_INFO()
Toggle lock flag of infomem segment A (the one with calibration data).

BSL_CRC_CHECK(*XXX*)

High level functions.

detect_buffer_size()
Auto detect buffer size. Saved the result in `buffer_size`. Silently ignores if the command is not successful and keeps the old value.

memory_read(*address*, *length*)

Parameters

- **address** – Location in target memory.
- **length** – The number of bytes to read.

Returns A byte string with the memory contents.

Raises ***BSL5Error*** – when `buffer_size` is undefined

Read from memory. It creates multiple BSL_TX_DATA_BLOCK commands internally when the size is larger than the block size.

mass_erase()
Clear all Flash memory.

erase(*address*)

Parameters **address** – Address within the segment to erase.

Erase Flash segment containing the given address

#~ def main_erase(self): #~ Erase Flash segment containing the given address.

execute(*address*)

Parameters **address** – Location in target memory.

Start executing code on the target.

password(*password*)

Parameters **password** – Byte string with password (32 bytes)

Transmit the BSL password.

version()
Get the BSL version. The 16 bytes of the ROM that contain chip and BSL info are returned.

reset()
Reset target using the WDT module.

exception `msp430.bsl5.bsl5.`**BSL5Exception**(*Exception*)
Common base class for errors from the slave

exception `msp430.bsl5.bsl5.`**BSL5Timeout**(*BSL5Exception*)
Got no answer from slave within time.

exception `msp430.bsl5.bsl5.`**BSL5Error**(*BSL5Exception*)

msp430.bsl5.hid

This module can be executed as command line tool (`python -m msp430.bsl5.hid`). It implements the BSL protocol over USB-HID supported by F5xx devices with buil-in USB hardware.

Currently implementations for Windows (pywinusb) and GNU/Linux are provided (hidraw).

class `msp430.bsl5.hid.`**HIDBSL5Base**

bsl(*cmd*, *message=''*, *expect=None*, *receive_response=True*)

Parameters

- **cmd** – BSL command number.
- **message** – Byte string with data for command.
- **expect** – Enable optional check of response length.
- **receive_response** – When set to false, do not receive response.

Low level access to the HID communication.

This function sends a command and waits until it receives an answer (including timeouts). It will return a string with the data part of the answer. The first byte will be the response code from the BSL

If the parameter "expect" is not None, "expect" bytes are expected in the answer, an exception is raised if the answer length does not match. If "expect" is None, the answer is just returned.

Frame format:

```
+------+-----+-----------+
| 0x3f | len | D1 ... DN |
+------+-----+-----------+
```

class `msp430.bsl5.hid.`**HIDBSL5**

open(*device=None*)

Parameters **device** – Name of device to use or `None` for auto detection.

Connect to target device.

Auto detection searches for a device with USB VID:PID: 2047:0200. It may pick a random one if multiple devices with that ID are connected.

Examples for the `device` parameter under GNU/Linux: `/dev/hidraw4`. Windows currently does not support passing an explicit device (only auto detection).

close()

Close connection to target.

write_report(*data*)

Parameters **data** – Byte string with report for target. 1st byte is the report number.

Write given data to the target device. The first bye of the data has to be the HID report number.

read_report()

Returns Byte string with report from target. 1st byte is the report number.

Read a HID report from the target. May block if no data is sent by the device.

class msp430.bsl5.hid.**HIDBSL5Target**(*HIDBSL5*, *msp430.target.Target*)
Combine the HID BSL5 backend and the common target code.

add_extra_options()
Populate the option parser with options for the USB HID connection and password.

close_connection()
Close connection to target.

open_connection()
connect to USB HID device using the options from the command line (in `options`). This will also execute the mass erase command and/or transmit the password so that executing other actions is possible.

As USB devices only have a stub BSL, this also downloads a full BSL to the device RAM. The BSL is kept in the package as `RAM_BSL.00.06.05.34.txt` (loaded using `pkgdata`).

reset()
Try to reset the device. This is done by a write to the WDT module, setting it for a reset within a few milliseconds.

msp430.bsl5.uart

This module can be executed as command line tool (`python -m msp430.bsl5.uart`). It implements the BSL target tool for F5xx/F6xx devices w/o USB hardware (it uses the UART).

msp430.bsl5.uart.**crc_update**(*crc*, *byte*)
Calculate the 16 bit CRC that is used by the BSL. Input is byte-wise. The function can be used with `reduce`:

```
crc = reduce(crc_update, b"data", 0)
```

class msp430.bsl5.uart.**SerialBSL5**(*bsl5.BSL5*)

extra_timeout
Extend timeout for responses by given number of seconds (int).

invertRST
Invert signal on RST line (bool).

invertTEST
Invert signal on TEST/TCK line (bool).

swapResetTest
Exchange the control lines on the serial port (RTS/DTR) which are used for RST and TEST/TCK.

testOnTX
Send BREAK condition on TX line (bool), additionally to use of TEST/TCK control line.

blindWrite
Do not receive and responses (bool).

control_delay
Delay in seconds (float) that is waited after each change of RTS or TEST/TCK line change.

open(*port=0*, *baudrate=9600*, *ignore_answer=False*)
Initialize connection to a serial BSL.

close()
Close serial port.

BSL_CHANGE_BAUD_RATE(*multiply*)

Parameters **multiply** – Multiplier of baud rate compared to 9600.

Low level command to change the BSL baud rate on the target.

bsl(*cmd*, *message=''*, *expect=None*)

Parameters

- **cmd** – BSL command number.
- **message** – Byte string with data for command.
- **expect** – Enable optional check of response length.
- **receive_response** – When set to false, do not receive response.

Low level access to the serial communication.

This function sends a command and waits until it receives an answer (including timeouts). It will return a string with the data part of the answer. In case of a failure read timeout or rejected commands by the slave, it will raise an exception.

If the parameter "expect" is not None, "expect" bytes are expected in the answer, an exception is raised if the answer length does not match. If "expect" is None, the answer is just returned.

Frame format:

```
+-----+----+----+-----------+----+----+
| HDR | LL | LH | D1 ... DN | CL | CH |
+-----+----+----+-----------+----+----+
```

set_RST(*level=True*)

Parameters **level** – Signal level.

Controls RST/NMI pin (0: GND; 1: VCC; unless inverted flag is set)

set_TEST(*level=True*)

Parameters **level** – Signal level.

Controls TEST pin (inverted on board: 0: VCC; 1: GND; unless inverted flag is set)

set_baudrate(*baudrate*)
Change the BSL baud rate on the target and switch the serial port.

start_bsl()
Start the ROM-BSL using the pulse pattern on TEST and RST.

class msp430.bsl5.uart.**SerialBSL5Target**(*SerialBSL5*, *msp430.target.Target*)
Combine the serial BSL backend and the common target code.

add_extra_options()
Populate the option parser with options for the serial port and password.

parse_extra_options()
Prints additional version info.

close_connection()
Close connection to target.

open_connection()
Open serial port, using the options from the command line (in `options`). This will also execute the mass erase command and/or transmit the password so that executing other actions is possible.

reset()
Try to reset the device. This is done by a write to the WDT module, setting it for a reset within a few milliseconds.

JTAG Target

interface to JTAG adapters (USB and parallel port).

msp430.jtag.clock

Note: This module is currently only supported with parallel port JTAG adapters and MSP430mspgcc.dll/so

msp430.jtag.clock.**getDCOFreq**(*dcoctl*, *bcsctl1*, *bcsctl2=0*)

Returns frequency in Hz

Measure DCO frequency on a F1xx or F2xx device.

msp430.jtag.clock.**setDCO**(*fmin*, *fmax*, *maxrsel=7*, *dcor=False*)

Returns (frequency, DCOCTL, BCSCTL1)

Software FLL for F1xx and F2xx devices.

msp430.jtag.clock.**getDCOPlusFreq**(*scfi0*, *scfi1*, *scfqctl*, *fll_ctl0*, *fll_ctl1*)

Returns frequency in Hz.

Measure DCO frequency on a F4xx device

msp430.jtag.clock.**setDCOPlus**(*fmin*, *fmax*)

Returns (frequency, SCFI0, SCFI1, SCFQCTL, FLL_CTL0, FLL_CTL1)

Software FLL for F4xx devices.

msp430.jtag.dco

Note: This module is currently only supported with parallel port JTAG adapters and MSP430mspgcc.dll/so

MSP430 clock calibration utility.

This tool can measure the internal oscillator of F1xx, F2xx and F4xx devices that are connected to the JTAG. It can display the supported frequencies, or run a software FLL to find the settings for a specified frequency.

This module can be executed as command line tool (`python -m msp430.jtag.dco`).

msp430.jtag.dco.**adjust_clock**(*out*, *frequency*, *tolerance=0.02*, *dcor=False*, *define=False*)
Detect MSP430 type and try to set the clock to the given frequency. When successful, print the clock control register settings.

This function assumes that the JTAG connection to the device has already been initialized and that the device is under JTAG control and stopped.

msp430.jtag.dco.**measure_clock**()

Returns A dictionary with information about clock speeds (key depend on MCU type).

Measure fmin and fmax of RSEL ranges or hardware FLL.

msp430.jtag.dco.**calibrate_clock**(*out*, *tolerance=0.002*, *dcor=False*)
Recalculate the clock calibration values and write them to the Flash.

Note: currently for F2xx only

msp430.jtag.jtag

JTAG programmer for the MSP430 embedded processor.

Requires Python 2.7+ and the binary extension _parjtag or ctypes and MSP430mspgcc.dll/libMSP430mspgcc.so or MSP430.dll/libMSP430.so and HIL.dll/libHIL.so

Constants used to identify backend implementations:

msp430.jtag.jtag.**PARJTAG**

msp430.jtag.jtag.**CTYPES_MSPGCC**

msp430.jtag.jtag.**CTYPES_TI**

msp430.jtag.jtag.**locate_library**(*libname*, *paths=sys.path*, *loader=None*)
Search for a .DLL or .so library on the given list of paths.

msp430.jtag.jtag.**init_backend**(*force=None*)

Parameters **force** – One of PARJTAG, CTYPES_MSPGCC, CTYPES_TI or `None`.

Initializes the global `backend` with a class that gives access to the JTAG library.

The backend implementation is selected automatically when `force` is `None`.

class msp430.jtag.jtag.**JTAG**
High level access to the target for upload, download and funclets. Uses the `backend` to communicate.

exception msp430.jtag.jtag.**JTAGException**(*Exception*)

msp430.jtag.target

This module can be executed as command line tool (`python -m msp430.jtag.target`).

class msp430.jtag.target.**JTAGTarget**(*object*)

def memory_read(address, length)
Read from memory.

memory_write(*address*, *data*)
Write to memory.

def mass_erase()
Clear all Flash memory.

main_erase()
Clear main Flash memory (excl. infomem).

erase(*address*)
Erase Flash segment containing the given address.

execute(*address*)
Start executing code on the target.

version()
The 16 bytes of the ROM that contain chip and BSL info are returned.

reset()
Reset the device.

close()
Close connection to target.

class msp430.jtag.target.**JTAG**(*JTAGTarget*, *msp430.target.Target*)
Combine the JTAG backend and the common target code.

help_on_backends()
Implement option `--target-help`.

add_extra_options()
Populate option parser with options for JTAG connection.

parse_extra_options()
Apply extra options (such as forcing a backend implementation)

close_connection()
Close connection to target.

open_connection()
Connect to target.

msp430.jtag.target.**main**()
Implements the command line frontend.

msp430.jtag.profile

Statistical profiler for the MSP430.

It works by sampling the address bus and counting addresses seen. The problem there is, that it is not sure that we're reading a valid address every time. An other issue is the relatively slow sampling rate compared to the execution speed of the MCU, which means that several runs are need to get meaningful numbers.

This module can be executed as command line tool (`python -m msp430.jtag.profile`).

Note: This module is currently only supported with parallel port JTAG adapters and MSP430mspgcc.dll/so

---

msp430.jtag.profile.**main**()
Command line frontend. It connects to a target using JTAG. It then samples the address bus as fast as possible (which is still much slower that the typical CPU speed). When the tool is aborted with `CTRL+C`, it outputs a list of addresses and the number of samples that were hit.

The idea is that the data can be used to create a statistical analysis of code coverage and usage.

There are a number of problems, so this tool has to be considered as experimental:

- Sampling is relatively slow.
- There is no guarantee that the value read from the address bus is correct. Samples may occur when the CPU is altering the value.
- There is no difference between instruction fetch and data access.

GDB Target

Interface to GDB servers (msp430-gdbproxy, mspdebug). This can be used to up- and download data via the GDB servers. No debugging support is provided.

msp430.gdb.gdb

exception msp430.gdb.gdb.**GDBException**(*Exception*)
Generic protocol errors.

exception msp430.gdb.gdb.**GDBRemoteTimeout**(*GDBException*)
If target does not answer.

exception msp430.gdb.gdb.**GDBRemoteTooManyFailures**(*GDBException*)
If target does not answer.

exception msp430.gdb.gdb.**GDBUnknownCommandError**(*GDBException*)
If target does not know this command.

exception msp430.gdb.gdb.**GDBRemoteError**(*GDBException*)

getErrorCode()

Returns The error code that was received from the GDB server.

class msp430.gdb.gdb.**ClientSocketConnector**(*threading.Thread*)
Make a connection through a TCP/IP socket. This version connects to a server (i.e. is a client).

__init__(*host_port*)
The host/port tuple from the parameter is used to open a TCP/IP connection. It is passed to `socket.connect()`.

write(*text*)
Just send everything

class msp430.gdb.gdb.**GDBClient**(*ClientSocketConnector*)

__init__(**args*, ***kwargs*)
All parameters are passed to *ClientSocketConnector.__init__()*

handle_partial_data(*data*)

Parameters **data** – Byte string with received data from the GDB server.

Process received data. It may be partial, i.e. no complete line etc.

handle_packet(*packet*)

Parameters **packet** – A packet received from the GDB server.

Called by *handle_partial_data()* when a complete packet from the GDB server was decoded.

Callbacks (can be overridden in subclasses):

output(*message*)
Called on `o` (output) packages. These are used by the GDB server to write messages for the user.

Commands:

set_extended()
Set extended mode. Expected answer empty string or or `OK`

last_signal()

Returns Stop Reply Packets

Get last signal.

cont(*startaddress=None*, *nowait=False*)

Returns Stop Reply Packets

Continue execution on target.

cont_with_signal(*signal*, *startaddress=None*)

Parameters **signal** – Signal number that is sent to the target.

Returns Stop Reply Packets

Continue with signal.

read_registers()

Returns List of values of the regsiters (1 ... 16)

Read all registers.

write_registers(*registers*)

Parameters **registers** – List with values for all registers (list of 16 ints).

Write all registers.

cycle_step(*cycles*, *startaddress=None*)

Parameters **cycles** – Run the given number of cycles on the target.

Cycle step (draft).

read_memory(*startaddress*, *size*)

Parameters

- **startaddress** – Address on target.
- **size** – Number of bytes to read.

Returns Byte string with memory contents

Read memory.

write_memory(*startaddress*, *data*)

Parameters

- **startaddress** – Address on target.
- **data** – Byte string with memory contents

Read memory.

read_register(*regnum*)

Parameters **regnum** – Register number.

Returns integer (register contents)

Read single register.

write_register(*regnum*, *value*)

Parameters

- **regnum** – Register number.
- **value** – integer (register contents)

Write single register. expected answer 'OK' or 'Enn"'""

query(*query*, *nowait=False*)

Parameters **query** – String with query for the GDB server.

Send general query to GDB server.

set(*name*, *value*)

Parameters

- **name** – Name of the setting.
- **value** – New value for the setting.

Configure a setting.

step(startaddress = None):

Returns Stop Reply Packets

Single step on target.

step_with_signal(*signal*, *startaddress=None*)

Parameters **signal** – Signal number that is sent to the target.

Returns Stop Reply Packets

Step with signal.

write_memory_binary(*startaddress*, *data*)
Write data to target, with binary transmission to GDB server. May not be supported by all servers.

remove_breakpoint(*type*, *address*, *length*)
Remove break or watchpoint (draft)

set_breakpoint(*type*, *address*, *length*)
Insert break or watchpoint (draft).

monitor(*command*, *nowait=False*)
Pass commands to a target specific interpreter in the GDB server. Servers for the MSP430 often implement commands such as `erase`.

interrupt()
Send Control+C. May be used to stop the target if it is running (e.g. after a continue command). No effect on a stopped target.

msp430.gdb.target

Remote GDB programmer for the MSP430 embedded processor.

This module can be executed as command line tool (`python -m msp430.gdb.target`).

class `msp430.gdb.target.`**GDBTarget**(*object*)

memory_read(*address*, *length*)
Read from memory.

memory_write(*address*, *data*)
Write to memory.

mass_erase()
Clear all Flash memory.

main_erase()
Clear main Flash memory (excl. infomem).

erase(*address*)
Erase Flash segment containing the given address.

execute(*address*)
Start executing code on the target.

version()
The 16 bytes of the ROM that contain chip and BSL info are returned.

reset()
Reset the device.

open(*host_port*)

Parameters **host_port** – A tuple `(str, port)` with target host address and port number.

close()

class msp430.gdb.target.**GDB**(*GDBTarget*, *msp430.target.Target*)
Combine the GDB backend and the common target code.

add_extra_options()
Populate option parser with GDB client specific options.

close_connection()
Close connection to target.

open_connection()
Connect to target applying the command line options.

Utility APIs

msp430.memory

class msp430.memory.**DataStream**(*object*)
An iterator for addressed bytes. It yields all the bytes of a *Memory* instance in ascending order. It allows peeking at the current position by reading the *address* attribute. `None` signals that there are no more bytes (and *next()* would raise `StopIteration`).

__init__(*self*, *memory*)
Initialize the iterator. The data from the given memory instance is streamed.

next()
Gets next tuple (address, byte) from the iterator.

address
The address of the byte that will be returned by *next()*.

msp430.memory.**stream_merge**(**streams*)

Parameters **streams** – Any number of *DataStream* instances.

Merge multiple streams of addressed bytes. If data is overlapping, take it from the later stream in the list.

class msp430.memory.**Segment**(*object*)

Store a string or list with memory contents (bytes) along with its start address.

__init__(*startaddress = 0*, *data=None*)

Parameters

- **startaddress** – Address of 1st byte in data.
- **data** – Byte string.

Initialize a new segment that starts at given address, containing the given data.

__getitem__(*index*)

Parameters **index** – Index of byte to get.

Returns A byte string with one byte.

Raises **IndexError** – offset > length of data

Read a byte from the segment. The offset is 0 for the 1st byte in the block.

__len__()

Return the number of bytes in the segment.

__cmp__(*other*)

Compare two segments. Implemented to support sorting a list of segments by address.

class msp430.memory.**Memory**(*object*)

Represent memory contents.

__init__()

Initialize an empty memory object.

append(*segment*)

Parameters **segment** – A *Segment* instance.

Append a segment to the internal list. Note that there is no check for overlapping data.

__getitem__(*index*)

Returns *Segment* instance

Raises **IndexError** – index > number of segments

Get a segment from the internal list.

__len__()

Returns Number of segments in the internal list.

get_range(*fromadr*, *toadr*, *fill='xff'*)

Parameters

- **fromadr** – Start address (including).
- **toadr** – End address (including).
- **fill** – Fill value (a byte).

Returns A byte string covering the given memory range.

Get a range of bytes from the memory. Unavailable values are filled with `fill` (default 0xff).

get(*address*, *size*)

param address Start address of block to read.

param size Size of the of block to read.

return A byte string covering the given memory range.

exception ValueError unavailable addresses are tried to read.

Get a range of bytes from the memory.

set (*address*, *contents*)

Parameters
- **address** – Start address of block to read.
- **contents** – Bytes to write to the memory.

Raises **ValueError** – Writing to an undefined memory location.

Write a range of bytes to the memory. A segment covering the address range to be written has to be existent. A `ValueError` is raised if not all data could be written (attention: a part of the data may have been written!). The contents may span multiple (existing) segments.

merge (*other*)

Parameters **other** – A Memory instance, its contents is copied to this instance.

Merge an other memory object into this one. The data is merged: in case of overlapping, the data from `other` is used. The segments are recreated so that consecutive blocks of bytes are each in one segment.

msp430.memory.**load** (*filename*, *fileobj=None*, *format=None*)

Parameters

- **filename** – Name of the file to open
- **fileobj** – None to let this function open the file or an open, seekable file object (typically opened in binary mode).
- **format** – File format name, `None` for auto detection.

Returns Memory object.

Return a Memory object with the contents of a file. File type is determined from extension and/or inspection of content.

msp430.memory.**save** (*memory*, *fileobj*, *format='titext'*)

Parameters

- **fileobj** – A writeable file like object (typically opened in binary mode).
- **format** – File format name.

Save given memory object to file like object.

msp430.listing

This module provides parser for listing/map files of the IAR and mspgcc C compilers. This can be used in tools that need to know the addresses of variables or functions. E.g. to create a checksum patch application.

Sub-modules:

- `msp430.listing.iar`
- `msp430.listing.mspgcc`

Each module provides such a function:

msp430.listing.**label_address_map** (*filename*)

Parameters **filename** – Name of a listing or map file.

Returns A dictionary mapping labels (key) to addresses (values/int).

File format handlers

Overview

The file format handler modules each provides a load and/or save function on module level.

msp430.listing.**load**(*filelike*)

Parameters **filelike** – A file like object that is used to write the data.

Returns *msp430.memory.Memory* instance with the contents loaded from the fike like object.

Read from a file like object and fill in the contents to a memory object. The file like should typically be a file opened for reading in binary mode.

msp430.listing.**save**(*memory*, *filelike*)

Parameters

- **memory** – *msp430.memory.Memory* instance with the contents loaded from the fike like object.
- **filelike** – A file like object that is used to write the data.

Write the contents of the memory object to the given file like object. This should typically be a file opened for writing in binary mode.

Handlers

msp430.memory.bin

Load and save binary data. Note that this is not practical for MSP430 binaries as they usually are not one block and do not start at address null. The binary format can not keep track of addresses.

msp430.memory.elf

ELF object file reader (typical file extension `.elf`). There is currently no support for writing this type.

msp430.memory.hexdump

Read and write hex dumps.

msp430.memory.titext

Read and write TI-text format files (often named `.txt`).

msp430.memory.intelhex

Read and write Intel-HEX format files (often named `.a43`).

CHAPTER 8

License

This is the simplified BSD license.

```
Copyright (c) 2001-2017 Chris Liechti <cliechti@gmx.net>
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the copyright holder nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

-------------------------------------------------------------------------
Note:
```

```
Individual files contain the following tag instead of the full license text.

    SPDX-License-Identifier:    BSD-3-Clause

This enables machine processing of license information based on the SPDX
License Identifiers that are here available: http://spdx.org/licenses/
```

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W