Run Launchpad with MSPDebug

Tommaso Toffoli (tt@bu.edu)
Electrical and Computer Engineering, Boston University, MA 02215

MSPDebug (here, DEBUG for short) is a command-line tool, designed and mantained by Daniel Beer, for programming and debugging the MSP430 family of microcontroller units (MCUs). It supports a number of MSP430 development tools as well as a simulation mode. DEBUG can also be used as a remote *stub* (a software interface adapter) for GDB—the multiplatform GNU debugger.

DEBUG belongs to a rich cultural environment and is well supported. It works on Linux, Mac OS/X, and Windows. For downloading and documentation, see mspdebug.sourceforge.net. Installation and configuration details may be found in mspdebug.sourceforge.net/faq.html.

When started with appropriate options, DEBUG attempts to connect to the target development hardware (such as a TI LaunchPad) and identify the device under test (the MCU to be programmed and debugged); once connected, it iteratively prompts for commands. There are commands for reflashing the device, inspecting and modifying memory and registers, and controlling the CPU (*single step*, *run*, *stop*, *run to breakpoint*, and *reset and halt*).

Besides dealing directly with machine language, DEBUG has substantial high-level smarts. It can disassemble machine code present in a given area of memory. When provided with a *symbol table* (an assignment of meaningful names to specific memory address and data values), it can make use of these names as convenient mnemonic handles for the values themselves, as well as deal with (understand or generate) memory addresses and values expressed in terms of such names. As mentioned, it can even run and debug a program on a *simulation* of an MSP430 MCU.

This user guide is a reasoned and annotated expansion of DEBUG's man page. Our chief target audience is the TI LaunchPad user. To avoid clutter, we only include discussion of command-line options and device options that concern the LaunchPad, omitting information about oulandish (and mostly obsolete) MSP430 development hardware; for those, please refer to DEBUG's official man page.

1 A bird's eye view

The operation of DEBUG consists in processing a sequence of *commands* specific to it—which are listed and discussed in the next section (§2). Command-line *options* and other device drivers are discussed in §4 and §5.

Interactive mode. For the LaunchPad, DEBUG is typically called from the command line as follows

```
mspdebug rf2500
```

where `rf2500` is the device *driver* to be used for the LaunchPad[1] When launched in this way, DEBUG enters an *interactive loop*: it repeatedly prompts the user for a command, which it then executes.

Batch mode. A variant of the above command line has the form

```
mspdebug rf2500 commands
```

The *commands* that appear as arguments at the end of this line are all executed by DEBUG as a batch right after connecting to the LaunchPad, and then the progam exits—the interactive prompt never appears.

One of DEBUG's commands is `read`, which reads and executes commands, in a line-by-line fashion, from a given file; in this way, a commonly used sequence of DEBUG command can be "canned" and then issued with a single command.

On startup, DEBUG will look for a file called `.mspdebug` in the user's home directory. If this file exists, DEBUG performs a `read` on it before starting the interactive loop; in this way, the command loop can be run within a customized enviroment.

DEBUG's commands—which are its *raison d'être*—will be listed and described first, in §2. Insofar as they are of concern to the LaunchPad, *options* inserted between the `mspdebug` command proper and the *driver* argument `rf2500` in the above command-line examples, as well as driver arguments other than `rf2500`, mostly deal with set-and-forget parameters of a technical nature, and will discussed in later sections (§4 and §5).

2 Commands

DEBUG commands take arguments separated by spaces (an argument itself containing spaces must be enclosed in double quotation marks; within a quoted string, the usual C-style backslash substitutions can be used). A numerical argument may be given in the form of an *address expression*, as detailed in §2.2.

Any command can be specified by giving just enough of the first few characters to make the specification unambiguous. Some commands support automatic repeat; for these, ENTER at the prompt will cause repeat execution.

Some aspects of DEBUG's behaviour can be configured by setting *global options*, with the command `opt`.

Once a piece of machine code is loaded into a microcomputer's memory, many of DEBUG's commands can make good use of a *symbol table* which provides mnemonic handles for numerical addresses and values which appear in that code. These symbolic names may be introduced during compilation/assembly by explicit `define`, `set`, `equ`, `=` and similar directives, or simply in the form of instruction labels. This human-readable information is of no use to the computer and may ultimately be discarded, but is invaluable for debugging and may be retained in some form for this purpose by some of the binary output files produced by the compilation/assembly, such as `.elf` and `.hex` files.

[1] Some recent versions of the LaunchPad—such as the *MSP430 FR4133 LaunchPad Development Kit*—use a different driver. This driver is available as a stopgap patch for the current version of DEBUG, but should appear as a standard feature in future releases.

To be useable by DEBUG, this mnemonic information has to be made available in the form of a symbol table (§2.1), which DEBUG may at a later time augment or edit as desired, with the `sym` command and its subcommands.

List of commands

`= expression` Evaluate an address *expression* and show both its value and the result when that value is reverse looked up in the current symbol table. This result is of the form *symbol+offset*, where *symbol* is the name of the nearest symbol not past the address in question. See §2.2 for more information on the syntax of expressions.

`alias` [*name* [*command*]] The `alias` command allows one to use shortcuts for commonly used long commands. By itself, `alias` show a list of defined command aliases. Followed by just a *name*, it *removes* a previously defined command alias having that name.

With both the *name* and *command* arguments, it defines (or redefines) a command alias. The *command* text will be substituted for *name* when looking up commands. If the command in this text is followed by arguments, then the entire text must be wrapped in quotes when defining the alias. To avoid alias substitution when interpreting commands, prefix the command with "\" (a backslash character).

`break` Show a list of active breakpoints. Breakpoints can be added and removed with the `setbreak` and `delbreak` commands. Breakpoints are numbered starting from 0. Their maximum number depends on the target chip (and is reported by DEBUG at start-up).

`cgraph` *address length* [*address*] Construct the call graph of all functions contained or referenced in the given memory range. If a particular function is specified, then details for that node of the graph are displayed; otherwise, a summary of all nodes is displayed.

Information from the symbol table is used for hinting at the possible locations of function starts. Any symbol which does not contain a '.' is considered a possible function start. Caller and callee names are shown prefixed by a '*' when their connection is a *tail call* (see `en.wikepedia.org/wiki/Tail_call`).

`delbreak` [*index*] Delete breakpoints. If an index is given, the selected breakpoint is deleted; otherwise, *all* breakpoints are cleared.

`dis` *address* [*length*] Dissassemble a section of memory. Both arguments may be address expressions. If no length is specified, a default length of 64 bytes is disassembled and shown. If symbols are available, then all addresses used as operands are translated into *symbol+offset* form.

This command supports repeat execution. If repeated, it continues to disassemble another block of memory following that last printed.

`erase` [`all`|`segment`] [*address*] Erase the device under test. With no arguments, all code memory is erased (but not information or boot memory). With the argument `all`, a mass erase is performed (the results may depend on the state of the LOCKA bit in the flash memory controller). To erase an individual flash segment,[2] specify `segment` and give the segment's starting *address*.

[2] *Segments* are the erase-unit size for flash memory when one wants to erase individual small chunks of memory rather than erasing all code memory or the whole chip including information and boot memory. Segment sizes vary from chip to chip; a typical size is 128 bytes. Note that segment size for information memory may be different from that of main memory.

`exit` Exit from DEBUG.

`gdb` [*port*] Start a GDB remote stub, optionally specifying a TCP port to listen on. If no port is given, the default port is `2000`. DEBUG will wait for a connection on this port, and then act as a GDB remote stub until GDB disconnects.

GDB's `monitor` command can be used to issue DEBUG commands via the GDB interface. Supplied commands are executed noninteractively, and the output is sent back to be displayed by GDB.

`help` [*command*] Show a brief listing of available commands. If an argument is specified, show the syntax for the given command. The help text shown when no argument is given is also shown by DEBUG at start-up.

`hexout` *address length filename* Read the specified memory range of the device and save it to an Intel HEX file. The *address* and *length* arguments may both be address expressions. If the specified file already exists it will be overwritten. If you need to dump memory from several disjoint memory regions you can do this by saving each section to a separate file. The resulting files can then be concatenated to form a single valid HEX file.

`isearch` *address length* [*options ...*] Search over the given memory range (from *address* for *length*) for an instruction which matches the specified search criteria. The search may be narrowed by specifying one or more *match options*, detailed in §2.3.

`load` *filename* Program the device under test using the binary file supplied. This command is like `prog`, but it does not load symbols or erase the device before programming. The CPU is reset and halted before and after programming.

`locka` [`set`|`clear`] Show or change the status of the LOCKA bit in the chip's memory controller. The LOCKA bit is set on *power-on reset* (POR) and acts as a write-protect bit for *info segment* A, containing factory-configured calibration data which under normal circumstances should not be changed; however, writing to info segment A becomes possible if the LOCKA bit is cleared. The LOCKA bit affects in a similar way the behaviour of the `erase all` command: if LOCKA is set (this is the default setting), only main memory is erased; but if LOCKA is cleared, both main and info memory are erased.

`md` *address* [*length*] ("memory display" or "memory dump") Read the specified section of device memory and display it as a canonical-style hex dump. Both arguments may be address expressions. If no length is specified, a default-length (64 bytes) section is shown.

The output is split into three columns. The first column shows the starting address for the line. The second column lists the hexadecimal values of the bytes. The final column shows the ASCII characters corresponding to printable bytes, and '.' for non-printing characters.

This command supports repeat execution. If repeated, it continues to print another block of memory following that last printed.

`mw` *address bytes ...* ("memory write") Write a sequence of bytes at the given memory address. The address given may be an address expression. Byte values are two-digit hexadecimal numbers separated by spaces.

`opt` [*name* [*value*]] Query, set, or list DEBUG's global options. With no arguments, this command displays all available options. With just an option name as its argument, it displays the current value of that option. The option names and their possible values are as follows:

- `color` [*boolean*] If true, DEBUG will colorize debugging output.
- `gdb_loop` [*boolean*] Automatically restart the GDB server after disconnection. If this option is set, then the GDB server keeps running until an error occurs or the user interrupts with Ctrl+C.
- `iradix` [*numeric*] Default input radix (an integer greater than 1) for address expressions. For address values with no radix specifier, this value gives the input radix, which is 10 (decimal) as initial default. (For `iradix` values larger than sixteen, letters following `f` are used in alphabetic order as additional digits past the ordinary sixteen hexadecimal digits.)
- `quiet` [*boolean*] If set, DEBUG will suppress most of its debug-related output. This option defaults to 'false', but can be set to 'true' on startup by the command-line option `-q`.

`prog` *filename* Erase and reprogram the device under test using the binary file supplied. Supported file types are HEX, TI-Text, SREC, ELF, S-19, and COFF; the file format will be auto-detected. In the case of a file containing symbols, symbols will be automatically loaded from the file into the symbol table, discarding any pre-existing ones. The CPU is reset and halted before and after programming.

Often `prog` is run as the only task of a DEBUG batch run, as with the line `mspdebug prog "proj.hex"`, which simply burns the machine-code memory image `proj.hex` of a developed project onto the MCU's FLASH ROM.

`read` *filename* Read commands line-by-line from the given file and process them in that order. (Any line whose first non-space character is `#` is ignored.) If an error occurs while processing a command, the rest of the file is not processed.

`regs` Show the current value of all CPU registers in the device under test.

`reset` Reset (and halt) the CPU of the device under test.

`run` Start running the CPU. The interactive command loop is blocked when the CPU is started and the prompt will not appear again until the CPU halts. The CPU will halt if it encounters a breakpoint or if Ctrl-C is pressed by the user. After the CPU halts, the current register values are shown together with a disassembly of the first few instructions at the address specified by the program counter.

`set` *register value* Alter the value of a register. A *register* is specified as numbers from `0` through `15`. Any leading non-numeric characters are ignored (so a register may be specified as, for example, `R12`). The *value* argument is an address expression.

`setbreak` *address* [*index*] Add a new breakpoint. The breakpoint location is an address expression. An optional index may be specified, indicating that this new breakpoint should overwrite an existing slot. If no index is specified, then the breakpoint will be stored in the next unused slot.

`simio subcommand` [*parameters*] Through various subcommands, add, remove, or give info about peripherals implemented by the simulator. The simulator and its subcommands are described in §3.

`step` [*count*] Step the CPU through one or more instructions. After stepping, the new register values are displayed together with a disassembly of the instruction at the address specified by the program counter.

An optional *count* can be given, to make DEBUG step that many times. If no argument is given, the CPU steps just once. This command supports repeat execution.

`sym` *operation* Perform on the symbol table one of the following operations:

- `clear` Clear the symbol table, deleting all symbols.
- `set` *name value* Set or alter the value of a symbol. The value given may be an address expression.
- `del` *name* Delete the given symbol from the symbol table.
- `import` *filename* Load symbols from the specified file into the symbol table. The file format will be auto-detected. DEBUG supports a number of commonly used symbol-table styles, among which ELF32, Intel HEX, and BSD (such as the output produced by `nm`(1)).
- `import+` *filename* This command is similar to `sym import`, except that the symbol table is not cleared first. In this way, symbols from multiple sources can be combined.
- `export` *filename* Save to the given file all symbols currently defined. The symbols are saved as a BSD-style symbol table. Note that symbol types are not stored by DEBUG; all symbols are saved as type 't' (denoting "text," as opposed to "data").
- `find` [*regex*] Search for symbols. If a regular expression is given, then all symbols matching the expression are printed. If no expression is specified, then the entire symbol table is listed.
- `rename` *regex string* Rename symbols by searching for those matching the given regular expression and substituting the given string for the matched portion. The renamed symbols are displayed and their number reported.

2.1 Symbol table

To preserve in the final ELF file (the one containing the machine code image ready to be loaded into the microprocessor's ROM, PROM, or FLASH memory) the symbolic names and associated values encountered in a compilation, the latter must be run with an appropriate option (e.g., `-g` in the GCC toolchain). To extract from a file `base.elf` the symbol table in BSD format, use the Linux command

```
nm -a base.elf > base.bsd
```

(typically issued by `Makefile`). To import this table into DEBUG, use the command `sym import base.bsd`.

2.2 Address expressions

An address expression consists of an algebraic combination of values. An address value may be either a symbol name, a hex value preceeded by the specifier `0x`, a decimal value preceeded by the specifier `0d`, or a number in the default input radix, without a specifier (for more information on input radix, see the global option `iradix` in command `opt`).

The operators recognized are the usual algebraic operators `+ - * / \ % ( )`. Operator precedence is the same as in C-like languages, and the - operator may be used as a unary negation operator. The following are valid examples of address expressions

```
2+2
table_start + (elem_size + elem_pad)*4
main+0x3f
__bss_end-__bss_start
```

2.3 Match options

Here is a list of match options that can be use to qualify the scope of the `isearch command`.

`opcode` Match the specified opcode. Byte/word specifiers are not recognised, as they are specified with other options.

`byte` Match only byte operations.

`word` Match only word operations.

`aword` Match only address-word (20-bit) operations, available in the MSP430X ("eXtended")architecture.

`jump` Match only jump instructions (conditional and unconditional jumps, but not instructions such as BR which load the program counter explicitly).

`single` Match only single-operand instructions.

`double` Match only double-operand instructions.

`noarg` Match only instructions with no arguments.

`src` *address* Match instructions with the specified value in the source operand; this value may be given as an address expression. Specifying this option implies matching of only double-operand instructions.

`dst` *address* Match instructions with the specified value in the destination operand. This option implies that no-argument instructions are not matched.

`srcreg` *register* Match instructions using the specified register in the source operand. This option implies matching of only double-operand instructions.

`dstreg` *register* Match instructions using the specified register in the destination operand. This option implies that no-argument instructions are not matched.

`srcmode` *mode* Match instructions using the specified mode in the source operand. See below for a list of modes recognised. This option implies matching of only double-operand instructions.

`dstmode` *mode* Match instructions using the specified mode in the destination operand. See below for a list of recognized modes. This option implies that no-argument instructions are not matched. For single-operand instructions, the operand is considered to be the destination operand.

For the sake of expressing instruction mode in the above match options, the seven addressing modes used by the MSP430 are represented by single characters as follows:

`R` Register mode
`I` Indexed mode
`S` Symbolic mode
`&` Absolute mode
`@` Register-indirect mode
`+` Register-indirect mode with auto-increment
`#` Immediate mode

3 The MSP430 simulator

DEBUG provides, as a virtual MSP430 device, a simulation mode intended for testing changes to DEBUG and for aiding in the disassembly of MSP430 binaries (as all binary and symbol-table formats are still usable in this mode).

A 64k buffer is allocated to simulate the device memory. (While the instructions of the MSP430X architecture are recognized by the the disassembler, this architecture is not currently simulated by DEBUG.) During simulation, addresses below `0200` are assumed to be I/O memory. Programmed I/O writes to and from I/O memory are handled by the I/O simulator, described below, which can be configured and controlled with the `simio` command.

For I/O memory locations not covered by `simio` commands, the default behavior is as follows. When data is written to an I/O memory address, a message is displayed on the console showing the program counter location, address written to, and data. The data value is also written to simulated RAM at the relevant address. When data is read from I/O memory, the user is notified similarly and prompted to supply the data. At this prompt, address expressions can be entered. If no value is entered, the value is read from simulated RAM. The user can press Ctrl+C to abort an I/O request and stop execution.

The subcommands of the `simio` command are

`add` *class name* [*args ...*] Add a new peripheral to the I/O simulator. The *class* parameter may be any of the peripheral types named in the output of the `simio classes` command. The *name* parameter is a unique name assigned by the user to this peripheral instance, and is used with other commands to refer to this instance of the peripheral.

Some peripheral classes take arguments upon creation. These are documented in the output to the `simio help` command.

`classes` List the names of the different types of peripherals which may be added to the simulator. You can use the `simio help` command to obtain more information about each peripheral type.

`config` *name parameter* [*args ...*] Configure or perform some action on a peripheral instance. The param argument is specific to the peripheral type. A list of valid

configuration commands can be obtained by using the `simio help` command.

`del` *name* Remove a previously added peripheral instance. The *name* argument should be the name of a peripheral that was assigned with the `simio add` command.

`devices` List all peripheral instances currently attached to the simulator, along with their types and interrupt status. You can obtain more detailed information for each instance with the `simio info` command.

`help` *class* Obtain more information about a peripheral *class*. The documentation given will list constructor arguments and configuration parameters for the device type.

`info` *name* Display detailed status information for a particular peripheral. The type of information displayed is specific to each type of peripheral.

3.1 Simulator's IO subsystem

The simulator's IO subsystem consists of a database of device classes, and a list of instances of those classes. Each device class has a different set of constructor arguments, configuration parameters and information which may be displayed. This section describes the operation of the available device classes in detail.

In the list below, each device class is listed, followed by its constructor arguments.

`gpio` Digital IO port simulator. This device simulates any of the digital ports with or without interrupt capability. It has the following configuration parameters:

`base` *address* Set the base address for this port. Note that for ports without interrupt capability, the resistor enable port has a special address which is computable from the base address.

`irq` *vector* Enable interrupt functionality for this port by specifying an interrupt vector number.

`noirq` Disable interrupt functionality for this port.

`verbose` Print a state change message every time the port output changes.

`quiet` Don't print anything when the port state changes (the default).

`set` *pin value* Set the input pin state for the given pin on this port. The pin parameter should be an index between 0 and 7. The value should be either zero (for a low state) or non-zero (for a high state).

`hwmult` This peripheral simulates the hardware multiplier. It has no constructor or configuration parameters, and does not provide any extended information.

`timer` *size* This peripheral simulates Timer_A modules, and can be used to simulate Timer_B modules provided that the extended features aren't required. The constructor takes a size argument specifying the number of capture/compare registers in this peripheral instance. The number of such registers may not be less than 2, or greater than 7. The I/O addresses and IRQs used are configurable. The default I/O addresses used are those specified for Timer_A in the MSP430 hardware documentation.

`base` *address* Alter the base I/O address. By default, this is `0x0160`. By setting this to `0x0180`, a Timer_B module may be simu- lated.

`irq0` *number* Set the `TACCR0` interrupt vector number. By default, this is interrupt vector 9. This interrupt is self-clearing, and higher priority than the `TACCR1/TAIFG` vector.

`irq1` *number* Set the `TACCR1/TAIFG` interrupt vector. By default, this is interrupt vector 8.

`iv` *address* Alter the address of the interrupt vector register. By default, this is `0x012e`. By setting this to `0x011e`, a Timer_B module may be simulated.

`set` *channel value* When Timer_A is used in capture mode, the `CCI` bit in each capture register reflects the state of the corresponding input pin, and can't be altered in software. This configuration command can be used to simulate changes in input pin state, and will trigger the corresponding interrupts if the peripheral is so configured.

`tracer` *history-size* The tracer peripheral is a debugging device. It can be used to investigate and record the I/O activity of a running program, to benchmark execution time, and to simulate interrupts. The information displayed by the tracer gives a running count of clock cycles from each of the system clocks, and an instruction count. A list of the n most recent I/O events is also displayed (this is configurable via the history-size argument of the constructor). Each I/O event is timestamped by the number of `MLCLK` cycles that have elapsed since the last reset of the device's counter. The I/O events that it records consist of programmed I/O reads and writes, interrupt acceptance, and system resets. As well as keeping the I/O events in a rotating buffer, the tracer can be configured to display the events as they occur.

Note that since clock cycles don't advance while the CPU isn't running, this peripheral can be used to calculate execution times for blocks of code. This can be achieved by setting a breakpoint at the end of the code block, setting the program counter to the start of the code block, clearing the tracer and running the code. After the breakpoint is reached, the information displayed by the tracer will contain a count of `MLCLK` cycles elapsed during the last run.

The configuration parameters for this device class are:

`verbose` Start displaying I/O events as they occur, as well as recording them in the rotating buffer.

`quiet` Stop displaying I/O events as they occur, and just record them in the buffer.

`trigger` *irq* Signal an interrupt request to the CPU. This request will remain raised until accepted by the CPU or cleared by the user.

`untrigger` Clear a signalled interrupt request.

`clear` Reset the clock cycle and instruction counts to 0, and clear the I/O event history.

`wdt` This peripheral simulates the Watchdog Timer+, which can be used in software either as a watchdog or as an interval timer. It has no constructor arguments. The simulated state of the `nmi/rst#` pin can be controlled through a configuration parameter. Note that if this pin state is held low with the pin mode selected as a reset (the default), the CPU will not run.

The extended information for this peripheral shows all register states, including the hidden counter register. Configuration parameters are:

`nmi` *state* Set the `nmi/rst#` pin state. The argument should be zero to indicate a low state or non-zero for a high state.

`irq` *irq* Select the interrupt vector for interval timer mode. The default is to use interrupt vector 10.

Appendices

4 Command-line options

Most of these options have to do with configuring DEBUG for a particular development tool—especially arcane and legacy ones. Here, we only discuss those options that are meaningful with the LaunchPad.

`-q` Start in quiet mode (see global variable `quiet` in command `opt`).

`-U` *bus:device* Specify a particular USB device to connect to (by default, the first USB device of the appropriate type is opened). This option may become necessary if one has to choose among more LaunchPads plugged into the same host, or if during a computer session accidents cause a LaunchPad to be relocated to a different USB port than the initial one (see end of §6).

`-n` Do not process the startup file (`~/.mspdebug`).

`--help` Display a brief help message and exit.

`--fet-list` Display a list of MSP430 devices supported by the driver (`rf2500`) used for the lauchpad.

`--fet-force-id` *string* When using a FET device, force the connected chip to be recognised by DEBUG as one of the given type during initialization; this overrides the device ID returned by the FET. The given string should be a chip name in long form, for example "MSP430F2274."

`--fet-skip-close` When using a FET device, skip the JTAG close procedure when disconnecting. With some boards, this removes the need to replug the debugger after use.

`--usb-list` List available USB devices and exit.

`--force-reset` When using an FET device, always send a reset during initialization. By default, an initialization without reset will be tried first.

`--version` Show version and copyright information.

5 Drivers

The driver used for the LaunchPad must appear in the command line. For the LaunchPad, in (**??**) we call for the `rf2500` driver. Another driver of interest is that which uses as "hardware" a *simulator* of the MSP430 architecture (§3). Drivers for other MSP430 development hardware are not discussed here.

6 USB port

The USB port in which a LaunchPad may be plugged will be accessible to DEBUG if the latter is run with root permissions. However, if DEBUG is run by a normal user, one would typically get an error like

```
Trying to open interface 1 on 007
rf2500: can't claim interface: Operation not permitted
```

That is, the user needs to be given permission to use the USB bus for the desired kind of device. For this purpose, create in the folder `/etc/udev/rules.d/` a Linux *udev* rule file that will give to users belonging to a certain group—say, `plug`—access to the USB port used by LaunchPad. This file (which may be named, for instance, `91-msp430.rules` (where we used 91 for a number between 00 and 99 indicating in which order the file will be taken into consideration within the folder) shall contain a line like

```
ATTRS{idVendor}=="0451",‘ATTRS{idProduct}=="f432",MODE="0660",GROUP="plug"
```

where the attributes `idVendor` and `idProduct` can be discovered through the `lsusb` command, which should print a line like

```
Bus 003 Device 007: ID 0451:f432 TI Inc. eZ430 Development Tool
```

one of which shows `04b3` and `301b` as respectively vendor and product of my LaunchPad. Then restart the `udev` service. If during all this your USB tool is still plugged in, it has to be *unplugged and plugged in afresh* for the new *udev* rule to take hold.

Also insure that users are members of `plug`. After you add a user to a group, the user will have to log out and then log in again to enjoy the new membership without having to wait for a reboot.

Note that a group `plug` may not exist, or a user may not be member of it. Also, a user may not have permission to join that group or to create it if it does not exist. A sensible policy is for the administrator to choose for `GROUP` in the udev rule a group which the users are likely to already be members of, such as the group `users`.

Occasionally, the USB slot in which DEBUG automatically discovered a LaunchPad is for some reason abandoned without closing it (e.g., faulty chip or LaunchPad—perhaps the LaunchPad misbehaved and DEBUG had to be forcefully aborted); in this case, on reinsertion of a LaunchPad, the latter may turn out to have been have been assigned to a *different* USB slot while DEBUG may still look for it at the original USB address. The new *bus:device* address can be found by mean of the Linux command `lsusb` and passed on to DEBUG through the `-U` option (e.g., `-U 0451:f432`).