

Opgui and OpenProg user's guide

v.1.2

November 2014

This document refers to the applications called OpenProg and Opgui, used to control the USB programmer called Open Programmer; OpenProg runs exclusively under Windows (2000,XP,Vista,7,8), Opgui both under Linux and Windows (after installing the [GTK runtime](#)); the ICD debugger and other functions are only supported in Opgui.

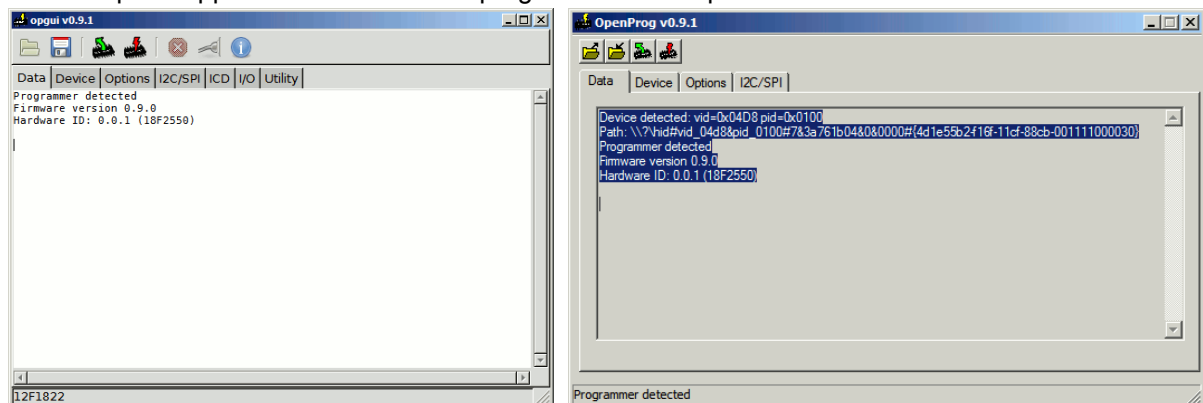
More info can be found on the project web site: www.openprog.altervista.org

From here on I assume that the programmer is working correctly and is recognized by the system.

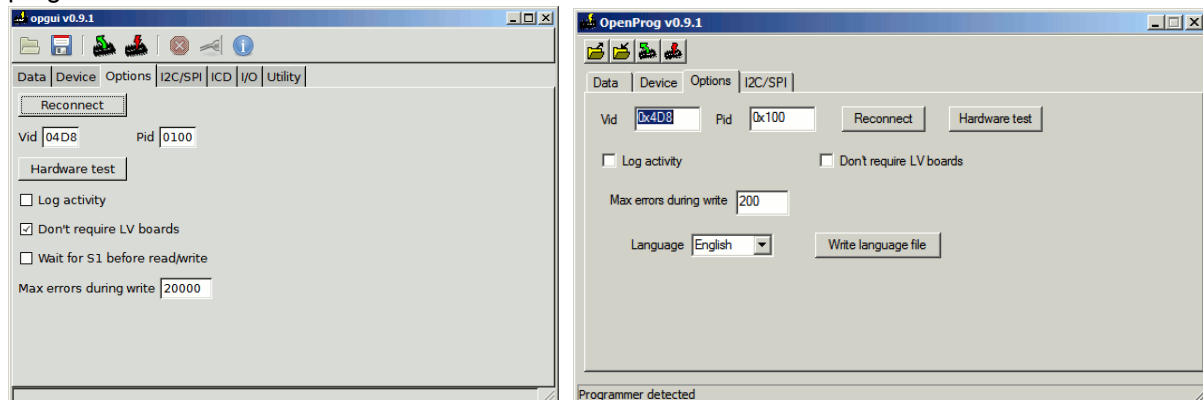
Every figure contains two screenshots, Opgui on the left, OpenProg (if available) on the right.

Step 1: connecting to the programmer

At start-up the applications search for a programmer and report the result:



If not found it's possible to search again using *Reconnect* in the *Options* tab; this also resets the programmer.



Within the "Options" tab it's possible to change the VID and PID codes used to identify the programmer; this is useful if the firmware was recompiled with different values than 0x4D8:0x100.

Log activity saves all USB packets exchanged with the programmer; it's useful in case of errors.

Don't require LV boards avoids checking for low voltage expansion boards.

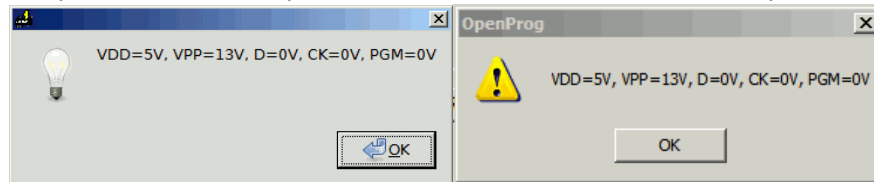
Wait for S1 before read/write starts programming when S1 is pressed; this can be useful in case of loose ICSP cables.

Max errors in writing is the maximum number of errors allowed before interrupting a write.

The *hardware test* is used to check the programmer itself; it should be executed at least once before using the programmer.

At each step a message specifies what voltage should be present on various lines; if this is true then the circuit is working; all digital I/O lines are then scanned to check their functionality.

Chapter Most common problems lists some common errors that prevent correct operation.



Step 2: device choice

On tab "Device" you can choose the device you want to use, considering that:

PICs with suffix LF are the same as with F, e.g. 16F628 = 16LF628;

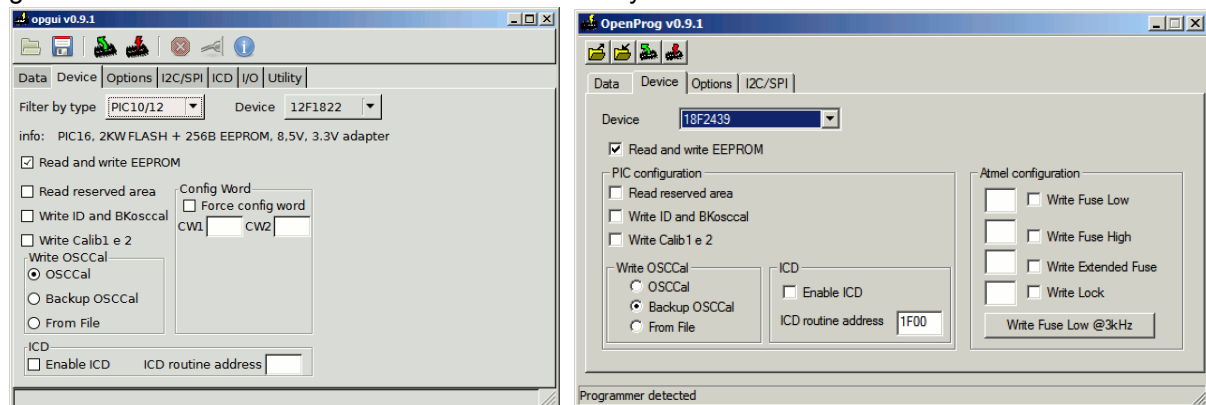
Atmel AVR with various suffixes are grouped when they use the same algorithm, e.g. ATmega8A = ATmega8;

EEPROMs type 24xx and 25xx include all versions with VDDmax=5V, e.g. 242LC56, 24AA256, etc.

Some FLASH SPI memories are also supported; they require an expansion board to lower the supply voltage to 3.3V.

EEPROMs type 93xx require two different algorithms, one for 93S series and one for all the others, called 93x; 93xA series are the same as 93x but organized as 8 bit instead of 16.

Read and write EEPROM: use the internal EEPROM of some microcontrollers; an error message is generated if the file to be written does not contain any EEPROM data.



PIC Configuration

The various options are used only if supported by the device chosen.

Read reserved area: most devices have a memory area beyond Config Words that is used for production test data or calibration; this option forces a read of this area as well; on PIC24-30-33 the executive area is also read.

Write ID and BKosccal: write ID locations (address 0x2000-2003 for PIC16) and backup calibration if specified in the file (i.e. if < 0x3FFF).

Write Calib1 and 2: write calibration words 1 and 2 if specified in the file.

Write OSCCal: some devices (e.g. 12F5xx) store the internal oscillator calibration value in the last address of the program memory, and in some cases also in a backup location after the Config area; after device erase the calibration value should be written back; the options are: *OSCCal*, the original

value, which was automatically saved before erase; *Backup Oscal*, the backup location, if exists; *From File*, which writes what is specified in the file.

Force config word: during write override the config settings present in the hex file; values are hexadecimal.

Enable ICD: write the address of the debugger routine; in order to use ICD it's also necessary to enable the DEBUG bit in the Config Word and to include the debugger routine in the code; see chapter Using the ICD debugger.

Atmel AVR configuration

Unlike PICs, these devices don't map their configuration words on the program memory, so this info is not present in the hex file and has to be entered manually.

Write the hex value of config/lock bytes and check the "write" box.

It's not mandatory to write all of the config locations, some of them may be fine with the current value.

More info on configuring the devices is present on the respective datasheet.

The CPU speed limits the communication speed to fCPU/4; the algorithm tries to find the highest speed possible; this of course influences the total read/write time.

In case of big devices it could be better to first change the CPU speed (by writing an empty file and setting the necessary Fuse bits) then write the final file and configuration.

Some models can have a very low CPU speed (16 kHz), which cannot be matched by the programmer.

In these cases first use the command "Write Fuse Low @ 3 kHz" to set a different CPU speed, then write with the usual procedure.

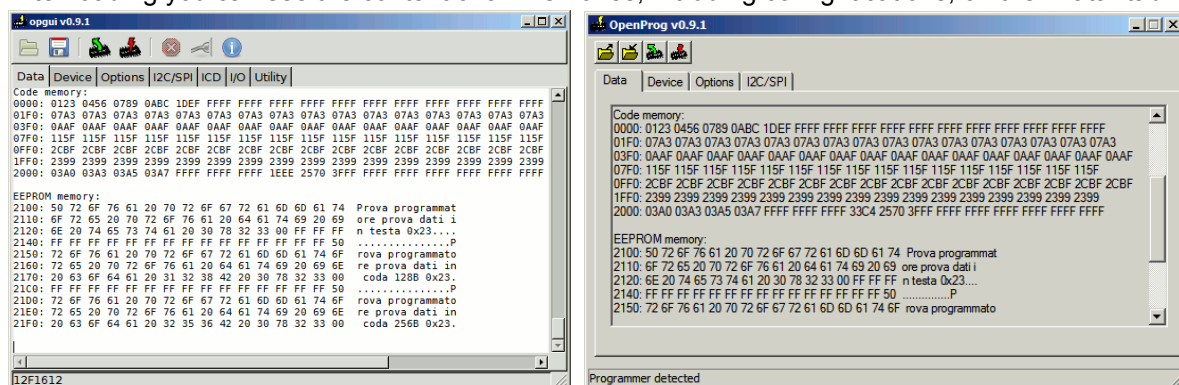
Step 3: load a file (only if you want to write)

Hex files are supported for all devices; binary files for serial EEPROMs only.

In PIC devices the EEPROM area is mapped in the same address space as program memory, so they are both included in the same hex file.

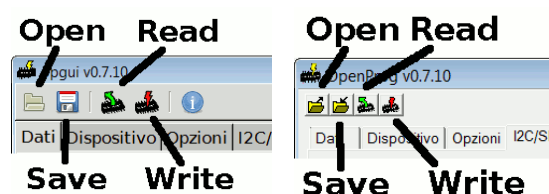
For Atmel micros you need a separate EEPROM file (hex or binary, often with extension .EEP).

After loading you can see the content of all memories, including config locations, on the "Data" tab.



Step 4: read or write to the device

Use the buttons in the tool bar:



During read/write the graphic interface does not respond, but shows the completion status as a percentage.

Press “stop” to abort immediately.

Total write time depends from the memory size, some seconds for 1KB, minutes for 128KB; often the microcontroller internal EEPROM has a higher cell write time than program memory but being small it does not slow too much the total operation; generally writing a small code on a large device takes little time.

Device erase, when necessary, is done automatically before write.

Also code verification is automatic; in some devices during write, otherwise after it; in any case the number of errors is shown at the end.

Errors relative to config words need some more explanation: usually not all bits are writable, some of them are fixed to 0; so if the word to be written specifies 1 a false error will be reported.

A frequent case is to write a program which was compiled for a different device, with different config bits.

You should always compare the specified config word with possible values in the datasheet to see if errors are real or not.

Reading is usually faster than writing; the device ID is read when available (on most microcontrollers).

Step 5: save a file (if the device was read)

Same considerations as step 3.

In case of microcontrollers the application does not save empty locations (with value $\geq 0x3FFF$) when possible.

Other operations

Erasing a device

Every device is erased automatically before write; if you want to erase it manually it's sufficient to write a hex file with valid data ($<0x3FFF$) beyond the memory space.

For example, with PIC12-16:

```
:020000040000FA
```

```
:0144010000BA
```

```
:00000001FF
```

Or PIC18:

```
:020000040002F8
```

```
:020000000000FE
```

```
:00000001FF
```

Changing Configuration Word

It's possible to override the hex file config settings using *Force config word* in the *Device* tab (opgui only, for PIC10-12-16-18).

Otherwise there are two methods:

modify the source code and recompile it;

modify the hex file directly: on a PIC16 the config word is stored at address 0x2007, which corresponds to hex address 0x400E.

The right line starts with “:02400E”; the last byte is the checksum, calculated as the two's complement of the sum of all preceding bytes:

```
:02400E00XXXXYY with XXXX new value and YY checksum
```

For example if XXXX=0370 YY=-(02+40+0E+03+70)=-C3=3D

Verifying that a device is empty

It's sufficient to read it and look at the data; only lines with valid data (<0x3FFF for PICs, <0xFF for the others) are shown; no data means that the device is empty.

Enabling access to the programmer under Linux

Under Linux the applications communicate through device /dev/usb/hiddevX (where X is the number assigned by the system to the programmer circuit); this file has to be readable so you have to change the "r" flag; for example in case of hiddev0:

```
> sudo chmod a+r /dev/usb/hiddev0
```

To permanently enable a user you can do the following (on Ubuntu and other Debian based distributions, check for others):

as root create a file /etc/udev/rules.d/10-openprogrammer.rules

if you want to enable a user group write:

```
KERNEL=="hiddev[0-9]", ATTRS{idProduct}=="0100", ATTRS{idVendor}=="04d8",  
GROUP="<group>", SYMLINK+="openprogrammer"
```

where <group> is one of the user groups (to get a list type "groups"); select a suitable group and if your user doesn't belong to it execute "addgroup <user> <group>"

or, if you want to enable all users, change reading permissions:

```
KERNEL=="hiddev[0-9]", ATTRS{idProduct}=="0100", ATTRS{idVendor}=="04d8", MODE="0664",  
SYMLINK+="openprogrammer"
```

restart udev to apply changes:

```
>udevadm control --reload-rules
```

```
>udevadm trigger
```

Now whenever the system detects the programmer, the corresponding /dev/usb/hiddevX will have the correct permissions, and a link /dev/openprogrammer will be created.

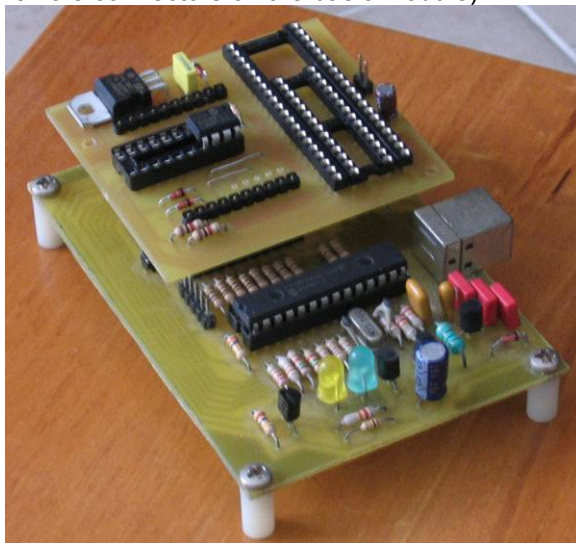
If after plugging-in the device you can't find /dev/usb/hiddevX (and LED2 doesn't blink at 1 Hz), it's sufficient to execute a few times lsusb to force enumeration, or unplug and plug the cable.

If not otherwise specified the program looks for an USB device with vid&pid=0x4d8:0x100.

Using expansion boards

The basic programmer module can host PIC devices with up to 20 pins (5V power supply) and I2C EEPROMs; all other devices require additional expansion boards (see [web site](#) for details).

These are designed so that their expansion connectors align exactly with those on the basic module, so it's possible to stack the boards and connect them vertically (this is why it is advisable to use female connectors on the basic module).



Important!! Don't use 3.3V devices without the 3.3V expansion boards, otherwise permanent damage may occur; the software verifies that such adapters are present before starting to program, but obviously you need to select the proper device; the 3.3V devices are:

12F1xxx, 16F1xxx, 18FxxJxx, 18FxxKxx, 24Fxxx, 24Hxxx, 33Fxxx.

Some of them have 5V variants; if you need to program at 5V you can select *Force config word* in the *Device* tab (opgui only, for PIC10-12-16).

Also don't put any 24F-33F on the 30F socket, which works at 5V.

Programming via ICSP

It's possible to program PIC devices directly on the application board, using the ICSP (In Circuit Serial Programming) method; this requires five signals: VPP, VCC, PGD, PGC, GND, all present on the expansion connectors.

A dedicated ICSP connector is present on some expansion boards.

Note that the 5 pin connector on the basic module is called ICSP-IN, and is a slave ICSP port to be used to program the main microcontroller from outside.

In order to ensure correct operation it is advisable not to load too much the ICSP lines (with capacitive or static loads); application boards often have jumpers to isolate the ICSP signals during programming.

Keep in mind that the maximum current flowing through VDD is in the order of 200-300 mA; exceeding this value will lower the power supply voltage and cause a programmer reset.

Using the ICD debugger

On Opgui exclusively is present an "ICD" tab, which allows operating the in-circuit debugger for PIC16 devices supporting this feature.

As with ICSP programming it's necessary to connect the standard five signals from programmer to application board; however this time VDD is optional, as the application circuit can also work with its own power supply, provided it's the same as the one from ICSP (i.e. 5V or 3.3V if using a 3.3V expansion board).

During a debugging session the device jumps to a particular routine, called debugger function or monitor, whenever a halt condition is encountered; this routine then communicates with the host and performs all kinds of debug operations.

Some preliminary steps are necessary before starting a debug session:

- compile/assemble the application program including the debugger function in one of two ways:
 - a) insert the debugger asm code directly in the main code, starting in the final part of program memory; a directive `ORG 0x1F00` is required before the debugger function code, which means it's starting 256 words from the end of 0x2000 word devices; use `ORG 0xF00` for those with 0x1000 words, etc. Debugger variables are stored in registers 0x6B-0x72, so make sure the main program does not use them.
 - b) include the precompiled module *debugger_mon.o* and a corresponding linker script; the module has to be added to the "Object files" in an MPLAB project; the linker script is a modification of the standard linker found in the lkr directory under MPASM; it's necessary to add a code page for the debugger code (at 0x1F00 for 0x2000 word devices, at 0xF00 for 0x1000 word devices, and so on):

```
CODEPAGE NAME=pageICD START=0x1F00 END=0x1FFF
SECTION NAME=ICD ROM=pageICD // ICD routine
```

It's also advisable to shorten the preceding page to 0x1EFF in order not to overlap with pageICD.

These modifications are similar to what Microchip does for its own debugger (look at the first part of the standard script, between `#IFDEF _DEBUG` and `#ELSE`), just with a different name for the debugger code page.

If really short on program memory it's possible to move the new page towards the end, keeping at least about 165 words for the debugger code.

To prevent using the debug memory locations (0x6B-0x72) it would be better to shorten the available memory sections:

`DATABANK NAME=gpr0 START=0x20 END=0x6A`

`SHAREBANK NAME=gpmobnk START=0x73 END=0x7F`

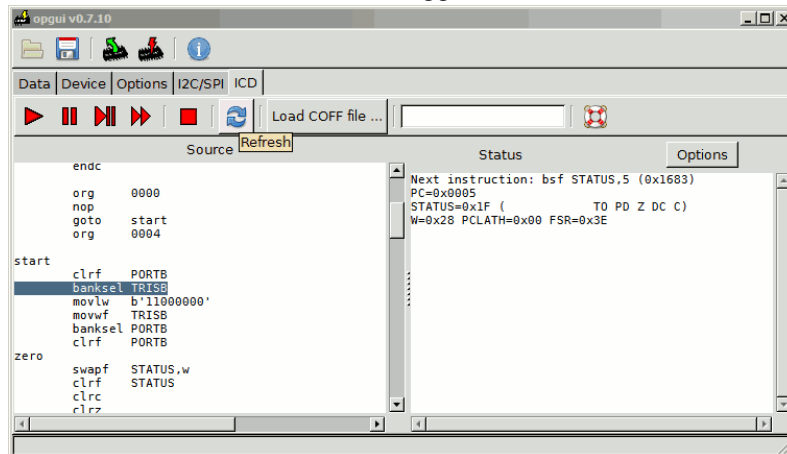
Anyways, make sure not to use locations 0x6B-0x72 in the program.

Other development environments need about the same steps described before.

- Enable the debugging bit in the configuration options:
`__FUSES (...) & _DEBUG_ON`
- Use NOP as the first instruction at address 0 (so before the usual `goto main`).
- In the programmer software enable option ICD on the “device” tab and set the correct address for the debugger routine (i.e. the starting address of `pageICD`, see above); in this way the device will jump to the debugger routine when a halt condition is met.
 It’s sufficient to do this once, because the ICD address is not erased during normal programming.
- Finally write the compiled program (which now contains also the debugger function) on the device.


At this point it’s possible to connect the programmer to the application board via ICSP connector.



The “ICD” tab contains all the debugger controls:





The “Source” pane shows the disassembled code around the current execution address; the other pane the overall execution status and other data.

A nice feature is the possibility to use the real source file; this is possible after loading the `.coff` file generated by the compiler; this file contains the definition of all variables used and the correspondence between memory line and source line.

Press “go/continue”  to start; the device will exit reset state and stop at address 1; the execution status is updated as well as the source code (or the disassembled code if no `.coff` file was loaded).

Press “step”  to execute one instruction at time; to do the same but jumping over calls press “step over” .

Press “Go/continue” again to run code at nominal speed, until “halt”  is pressed.

“Stop”  resets the device and turns off the power supply.

Going step by step to the point of interest may be too long; in this case it’s useful to use a breakpoint; just double click on the line you want to stop at (a message will appear on the status pane), and restart execution with “go/continue”.

The device then executes its program at nominal speed until the program counter matches the break address; after completing the current instruction it jumps to the debug routine; this means that it stops **after** the break address, which has to be considered when breaking before or after a function call.

A variable’s value is displayed double-clicking on its name in the source pane (only after loading a `.coff` file); the value will be updated throughout the debug session; double click again to remove it.

Using the “Options” button it’s also possible to display an entire bank of memory or the internal EEPROM.

Some other interesting functions are available through the command line (on the ICD toolbar); other than basic commands (run, step, step over, halt, stop) there are:

break, to manually set a breakpoint;

print, to show variables, registers, locations in RAM, program memory, or EEPROM;

watch, to display variables at every step;

freeze, to stop peripherals like watchdog and timers when the device is halted;

help, to list all commands;

To change the value of a variable or memory location write:

variable=x

e.g. PORTB=1F

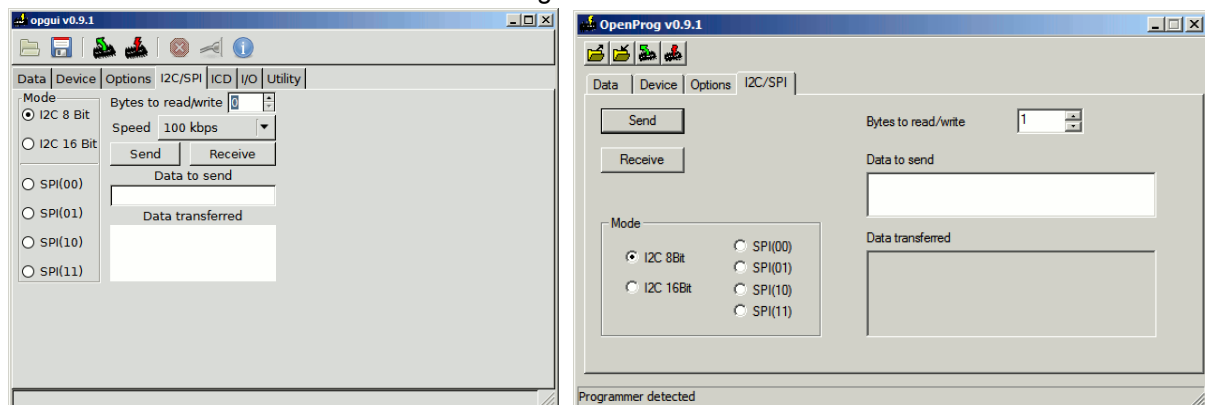
or: 0x23=FF

To test the debugger you can download the [example project](#); it’s compiled for the 16F873 but it’s adaptable to any other device; it contains also the source code for the debugger monitor and the .co module described above; a recompilation is required to update file paths in the .coff file.

A more technical explanation of how the debugger works can be found [here](#).

Communicating via I2C and SPI

The I2C/SPI tab allows to communicate with generic I2C and SPI devices.



I2C

Select the I2C mode between 8 bit and 16 bit, and speed between 100kbps and 500kbps.

“Bytes to send/receive” are the effective data bytes; every transfer needs also a control byte and an address byte, or two address bytes when in 16 bit mode; the RW bit in the control byte is handled automatically.

So for a read you put two or three bytes in the “data to send”; for a write the same two or three plus the data bytes to be transferred.

Not specifying anything equals to writing 0 in the relevant bytes.

An acknowledge error is shown in case the slave device is missing or does not respond.

For example in order to write 5 bytes on a 24xx16 memory at address 64 you select 5 bytes to send/receive and write: A0 40 1 2 3 4 5

To read from the same locations only the first two bytes are considered.

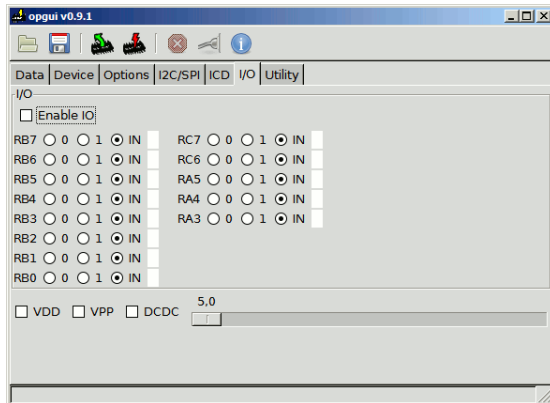
SPI

Select the SPI mode between the four standard modes and speed between 100kbps and 800kbps, then read or write.

CS (RB3) is high during every transfer.

Controlling I/O lines and power supplies

Using the *I/O* tab it's possible to control interactively all I/O lines, the two power supplies, and the DCDC converter; all options should be disabled before programming a device.

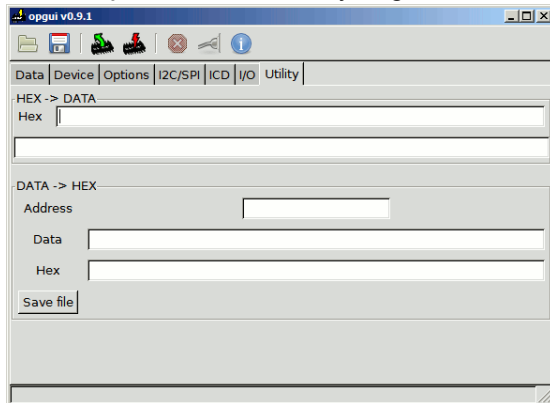


When *Enable IO* is checked every line can be driven high, low, or in high impedance (i.e. an input); the input value is refreshed every 100ms.

VDD and *VPP* control the same signals on the programmer; *DCDC* turns on/off the switching regulator and sets the voltage as specified (from 5 to 15, although the regulator may not be able to generate exactly that value).

Verifying or generating a hex file line

The *Utility* tab allows to verify or generate a line from a .hex file.



Hex>data extract address and data from a hex string; *data>hex* does the opposite. The hex string can be saved to a file.

Changing the user interface language

It's possible to change the language of Opgui using some command-line options:

-langfile generates the string file *languages.rc*; this file contains all the strings and an ID for each of them.

Translate all strings to your language and rename the session (the name is at the beginning enclosed in square brackets []).

If *languages.rc* is found and there is a session named like the system language ID (e.g. *en_en* for English), it will be automatically selected; otherwise use option:

-lang <language> to manually choose the language.

It's possible to change the language of OpenProg user interface by selecting it in the *Options* tab; the application comes with English and Italian strings.

In order to add more it's necessary to first generate the file `languages.rc`, choosing *Write language file*.

Translate all strings as described above; the session name will then appear among the language options.

Sending commands to the programmer

With Opgui you can manually send an USB packet with a command sequence to the programmer:

`-command [commands]`

The packet is filled with 0 to the final size (64 bytes).

The response packet will be also printed.

A detailed description of all commands is available on the project website.

Most common problems

Description	Cause and solution
HV regulator error	DCDC regulator components of wrong value or not correctly mounted.
Synchronization error with Atmel AVR	Device mounted on the wrong socket; common error: ATtiny2313 has to be mounted on the 28 pin socket. Serial download not enabled on the device: enable it using a parallel programmer. Serial download cable too long: use a shorter cable. Not all power supply pins are connected: connect all of them. Pin X2 fixed to GND: release it.
PGD (RB5) not working during hardware test	Main microcontroller not correctly programmed; usually this comes from using the LVP option while programming.
Programmer not detected, LED2 blinks quickly	Reset during USB enumeration. 10µF capacitor not mounted or of incorrect value.
VUSB < 4.5V error	Anomalous power consumption on VDD, check the transistors. The USB Hub may be overloaded: disconnect all other devices.
3,3V regulator not detected	The device selected requires an expansion board with a 3.3V regulator. Note that the expansion is needed if any of the device variants require 3.3V; e.g. 16F1936 (5V) vs. 16LF1936 (3.3V). If you are sure to program at 5V you can skip the check with <i>Don't require LV boards</i> .
All Hardware test errors	Shorts between traces or open traces: check the PCB.

If you have doubts or suggestions you can contact me at albertom78@gmail.com