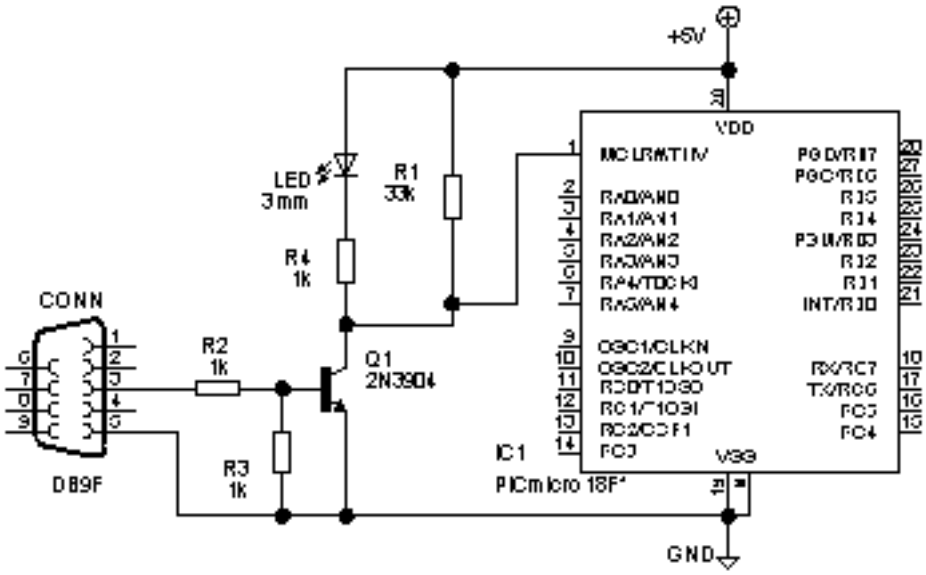
ZPL: a Zero Pin Loader for the PICmicro 18F family

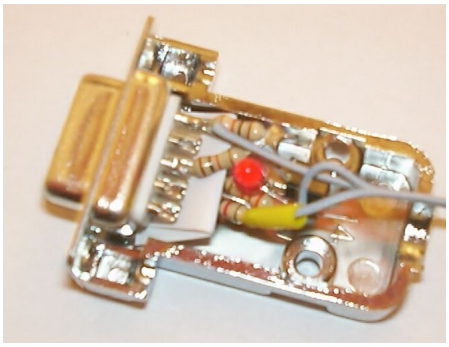
Mad Dash for Flash Cash project nr M285

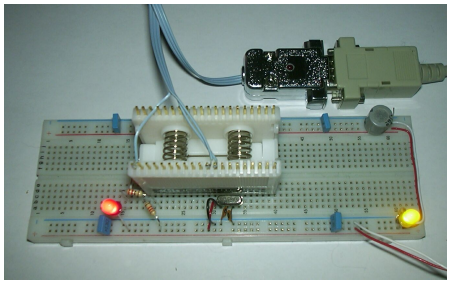
ABSTRACT

This bootloader for the PICmicro 18F series avoids the use of I/O pins by exploiting the /MCLR pin as the only interface between the host PC and the PICmicro.

The interface between the PC serial port and the target uses just six components. One of these is the standard /MCLR pull-up resistor and two others are optional. Die-hards might even leave out two more resistors, leaving just a single transistor. The interface circuit can easily be put in a DB9 shell, with a DIP clip for the connection to the target.







For most development environments the use of this bootloader is totally transparent, the only limitation being that the highest 384 instructions are not available to the application. The bootloader puts a jump to its own code at the first addresses, but the application's code for these addresses is relocated to a location within the bootloader so these instructions appear to be available to the application.

The host PC software is written in Python so it runs on Windows and Linux (and other POSIX-compatible systems), and can be easily ported to other operating systems.

The download time for a full 18F452 (32 k instructions minus the 384 for the bootloader) is clocked at 70 seconds. A small program downloads in just a few seconds.

ARTICLE

Bootloaders

One aspect of cross development is that you need a way to load your compiled or assembled application program into your target system. Before the arrival of FLASH-based microcontrollers this typically required removing the target chip from its system, using an EPROM eraser and a programmer, and re-inserting the target in its system. The arrival of affordable FLASH (or EEPROM) based microcontrollers that can be programmed in-circuit eased this process tremendously, and has contributed significantly to the popularity of the now (honorably) retired 16C84. Beside a PC for the cross development, and of course a target system, the user now needs only a programmer that is capable of ICSP (In Circuit Serial Programming).

With the 16F877 family Microchip introduced self-writing: these chips can write to their own code memory, which makes it possible to do away with the programmer. A small so-called bootloader program, running in the in the target chip itself, takes care of the communication with the host PC and writes the application program to the (remaining) code memory. The only hardware required is the interface between the host PC and the target chip, typically an RS232 interface chip, for instance a MAX232.

Transparency

The use of a bootloader is not without its specific problems. The bootloader must claim some resources of the microcontroller, either permanently or during the boot (reset). To ease the use of a bootloader (as alternative for an ICSP programmer) the extra requirements put on the development process should be small. Ideally the use of a bootloader should be totally transparent (indistinguishable from using an ICSP programmer).

In reality every bootloader requires some code space. This can be as little as 256 instructions, or as much as 1024. When the application program is assembled or compiled it can be necessary to instruct the assembler or compiler to avoid the code space used by the bootloader.

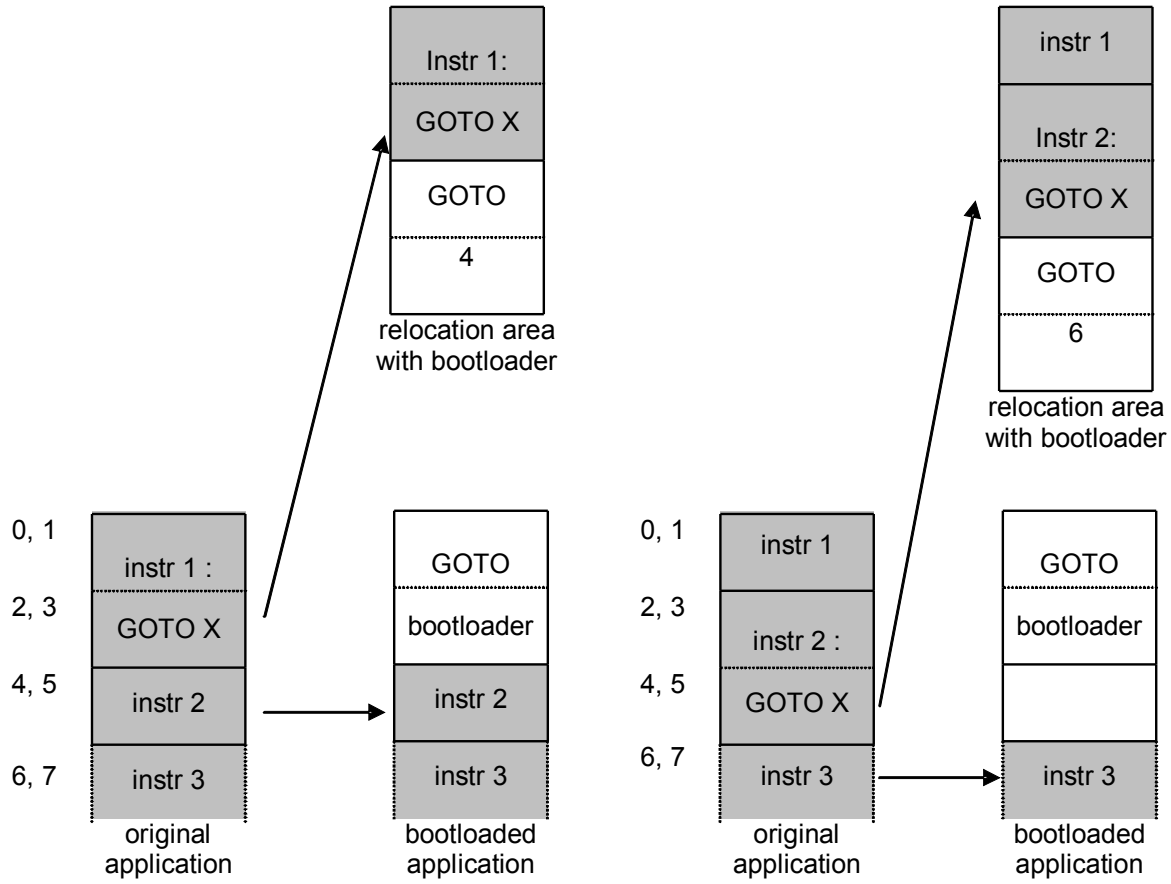
A bootloader must always be able to download a new application, so it can not depend on the current application to give it control. (There might not be a current application yet, or it could be malfunctioning.) So it must get control when the PICmicro starts executing its first instruction, which for FLASH PICmicro's resides at address 0. But an application that is meant to run on a PICmicro likewise starts at address 0, and in nearly all cases grows from there. To get the bootloader 'out of the way' it must be put at the last addresses an application would use, which means the top of the FLASH address range. Now the only remaining conflict is a jump at address 0 to activate the bootloader. The application also has an instruction to be put at address 0, but this instruction can be relocated (put elsewhere), immediately followed by a jump to the second instruction of the application, which can be at its 'natural' address. To activate the application the bootloader code establishes the 'reset' conditions, and jumps to the relocated instruction 0. This instruction either jumps directly to some other part of the application, or when it does not jump the jump that has been put after it transfers control to the second instruction of the application (at address 1), exactly what would happen if instruction 0 was executed from its natural location. This makes the bootloader practically invisible to the application code.

On the 18F series an instruction occupies either two addresses of 8 bits each (one word, 16 bits), or four addresses (two words, 32 bits). An instruction always starts at an even address. A jump (GOTO) is one of the few instructions that occupy four addresses. The jump that activates the bootloader occupies addresses 0..3.

The simple transparency approach would be to relocate the first 4 bytes of the application, followed by a jump to address 4. But when the application happens to have a two word instruction at addresses 2..5 this instruction would be split, which is obviously not what we want. Splitting after address 5 will likewise fail when the application has a two word instruction at addresses 4..7, so the only solution is to check for two word instructions and act accordingly. Fortunately the second word of a two-word instruction is easily recognized (the highest four bits are ones). The bootloader checks the word at addresses 4..5. If it is the second word of a two-word instruction, it splits after it. If it is not, it splits after the addresses 2..3. The picture below shows how the application’s first instructions are relocated in both cases.

Note that there is still one case in which this approach will fail: when the first instruction is a branch (relative jump) or (not very likely) a relative call. This (single word) instruction will be copied faithfully to the relocation area, but when it executes there it would transfer control to the wrong location. Luckily Microchip has put an interrupt vector at addresses 8, so there is little benefit in using a branch instead of a full (absolute) GOTO at

address 0, because the saved locations can not easily be put to use. Consequently most compilers just put a GOTO to the main code at address 0.



relocation of the application's first instructions

Communication

A bootloader must communicate with the host system (PC). This (traditionally) requires the use of one or more I/O pins. Using the target's hardware UART can reduce the code size of the bootloader, but requires the use of the UART's fixed I/O pins. Using a bit-banged (software) UART gives full freedom in choosing the I/O pins used for communication (and with some cleverness one pin is sufficient), but at the cost of more code.

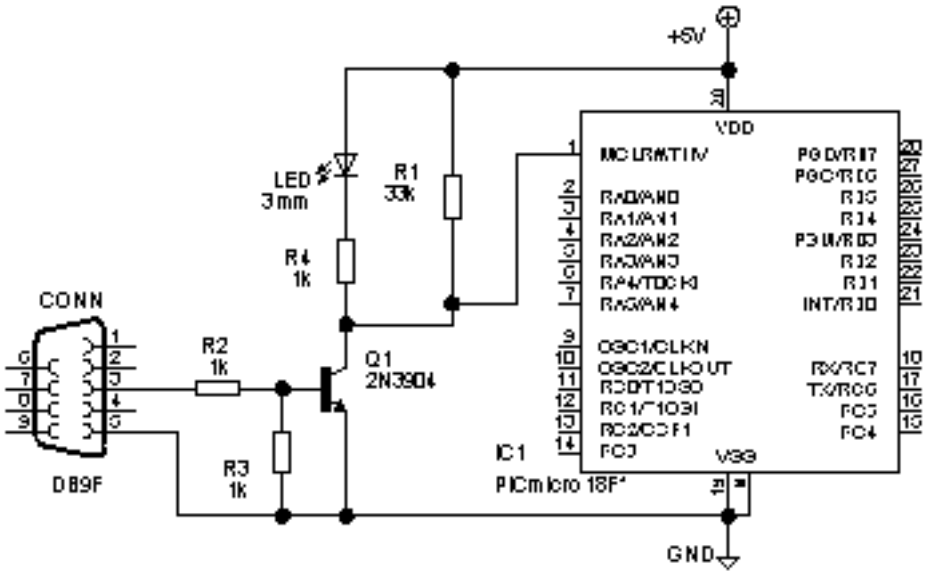
The historical trend is that new microcontrollers are faster and have more memory and peripherals than their predecessors, at a price that is only a fraction higher (or sometimes even lower!). The 18Fxxx family illustrates this trend: compared to the 16Fxxx chips they offer twice the code space, twice the speed, much more RAM, a more powerful CPU, and more and better peripherals. The number of I/O pins however does increase at the same rate, because it is limited by the available packages. So with each new generation of chips the chance increases that the availability of I/O pins will be the limiting factor for your application.

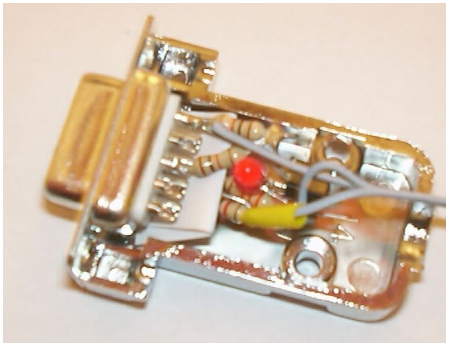
The bootloader described here can of course not avoid the use of some code space, but it does avoid the use of I/O pins by using an often neglected pin for communication: the /MCLR (Master CLear and Reset) pin. This may sound like black magic, but the principle is very simple: the PC manipulates the reset pin so the processor gets to run varying amounts of time. The processor records the length of time it was allowed to run, and when it gets to run again it interprets a short previous run time as a 0 and a longer time as a 1. Voila, a communication channel. Modern PCs running Windows or Linux are not capable of very precise timing in the microseconds region, so you might fear that complicated hardware is involved. Luckily the ancient serial port with its UART is perfect for this purpose. For modern hardware that has no serial port available an USB->serial adapter can be used, which is in fact the way the bootloader was developed.

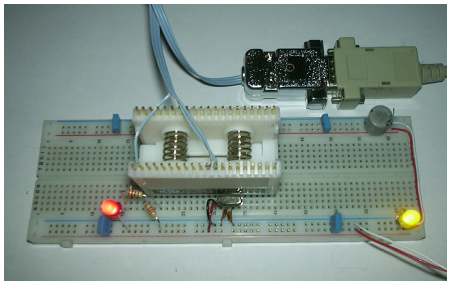
Hardware

The hardware to interface the serial port to the target is minimal: four resistors, a LED and one general-purpose small signal transistor (I used a 2N3904, but a BC547 or a 2N2222 should work just as well). The circuit can be incorporated in the target hardware, or (as shown below) be put in a DB9 connector, using a DIP clip to connect to the target. When no host is connected (or even when one is connected but is does not send any data) the circuit will not interfere with the normal operation of the target circuit. The circuit shows a LED and series resistor that can be useful to check that the correct serial port is used, but these components are not necessary for the functionality of the bootloader. The /MCLR pull-up resistor is mentioned as part of the circuit because it must not be too large, but in most cases this resistor will already be part of the target circuit.

I have successfully tested the circuit without the resistors near the base of the transistor (base of the transistor connected directly to the serial input). This relies on the current limitation of the sending RS232 port to set the base current, which is definitely not recommended for serious use.







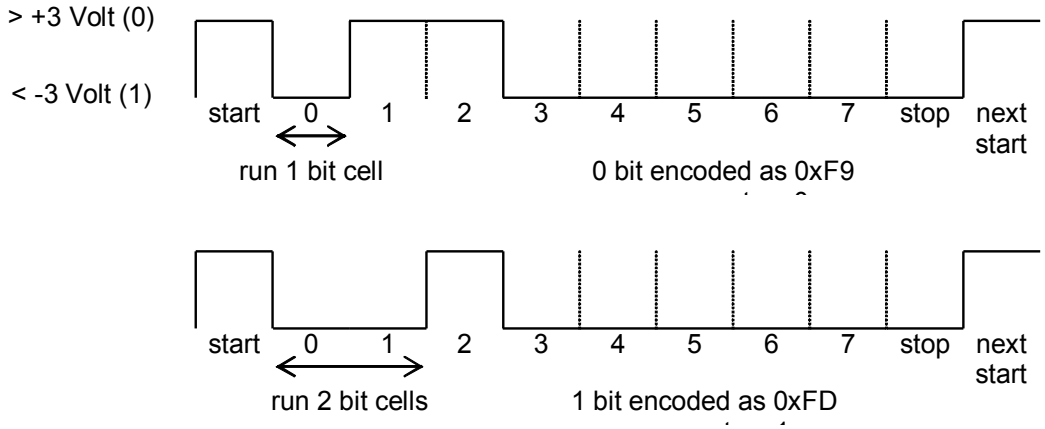
When the bootloader is used with an existing target circuit care must be taken that the rise and fall times of the /MCLR signal are short (preferably 1 µs or less). With only a resistor as pull-up and the transistor as pull-down this is easily met, but any capacitance on the /MCLR (for instance for reset timing) will pose a problem. The author experienced this the hard way in a 6-hour bug hunt because his in-circuit programmer placed a rather large capacitor on the /MCLR line, which caused the communication to be succeed only when there were large gaps between the transmitted characters.

Communication protocol

The figure below shows how a character is transmitted asynchronously using RS232 levels, and how this is used to transmit a 0 or 1 bit. The 8/N/1 format is used, which means that a byte is transmitted as one start bit (high), 8 data bits (least significant bit first, inverted polarity), and one stop bit (low). The next byte can be transmitted immediately after the stop bit, or there can be an idle period with the same level as the stop bit.

The hardware uses a transistor which inverts the signal, so the PICmicro will be in reset while the RS232 signal is high (> 3 Volt), and it will be running when the signal is low (< -3 Volt). The low level is the idle state, so when no character is transmitted the PICmicro will be running without interference. The highest baudrate available on a PC (115200 baud) is used with the characters show in the figure, so transmission of a 0 bit results in the target for running one bit cell (approximately 8.7 us), and transmission of a 1 bit in

running for two bit cells, both followed by running for at least six bit cells. (Six bit cells is when the next byte starts immediately after the stop bit, so the actual run time can be much longer.) The transmission rate is compareable to 14k4 asynchronous (10 / 80 of 115k2).



This basic communication allows for the sending of 1's and 0's, but without any byte boundaries or distinction between commands and data. Hence a 'reserved' command indicator pattern (xA5) is used to identify a command, and bit stuffing (or more correctly: bit appending) is used to avoid this pattern in the data: when the command indicator pattern has been transmitted, the next bit identifies it as part of the data stream (bit = 0) or a command indication proper (bit = 1). Note that 0-bit insertion in the data stream is not limited to byte boundaries, because no byte boundaries can be identified in the data stream until a proper command indicator has been found.

With these ingredients a simple message format is defined, as shown below. Each message consists of 64 data bytes (D0 .. D31) followed by 3 address bytes (A0..A3, A0 = least significant bit), a command byte (Cmd), a checksum byte (XOR) and a command indicator (Ind). Each byte is transmitted least significat bit first. The size of 64 data bytes was chosen to match the amount of data that is erased by an internal FLASH erase operation. The Checksum is the byte-wise XOR of the data, address and command bytes. As implied by the bit stuffing an extra 1 is sent at the end of the command indicator, and an extra 0 is sent after every occurrence of the command indicator bit pattern within the remainder of the bit stream. This stuffing has no influence on the checksum.

D0	D1	…	D30	D31	A0	A1	A2	Cmd	XOR	Ind

message format

The table below shows the commands that are defined. A START command is required before other commands are recognized. The START command also resets the write counter. After a WRITE FLASH or WRITE EEPROM command the host must refrain from sending subsequent data for the indicated Wait time, to allow the internal operations to complete. Unfortunately the datasheets do not provide a maximum time (neither for a FLASH erase / update, nor for an EEPROM write), so a conservative estimate is used based on the typical times (which are provided). The RUN command checks wether the number of writes indicated in A1:A0 matches the number of WRITE FLASH and WRITE EEPROM commands (A2 is ignored). If the numbers match the application is activated. The check avoids activation of an image that has not been succesfully downloaded. The counting of WRITE FLASH and WRITE EEPROM commands ignores repeated writes to the same address. This can be used to make the communication resistant to an occasional communication error.

Command	**Cmd value**	**Address interpretation**	**Data interpretation**	**Wait time**
START	0x33	don't care	Don't care	30 ms
WRITE FLASH	0x44	A2:A1:A0 = FLASH address to write data to	Data to be written to FLASH	100 ms
WRITE EEPROM	0x55	A0 = EEPROM address to write data to	Data to be written to EEPROM	250 ms
RUN	0x66	A1:A0 = number of writes	Don't care	n.a.

Note that even though the 18F chips can write to their configuration fuses memory, no commands are provided to do so. Doing so would have complicated the bootloader firmware, and for most configuration fuse bits writing is not desirable anyway because it could interfere with the working of the bootloader, thus preventing furher downloads.

The communication is entirely one-way, the PC does not even have a way to check that the interface circuit or the target are present at all. When a communication problem occurs the best that can be done is not enabling the application program. Therefor it is advisable that the user arranges for a way to check that the application program is successfully loaded and running. This can be as simple as a LED that lights up when the application runs, or more sophisticated like a version number that is shown on an LCD.

When the rate of communication errors is deemed too high the host PC software can be instructed to send each command a specified number of times. This will reduce the error rate significantly, but at the expense of an increased download time.

Host PC software

The host PC software is written in Python so it runs on Windows and Linux (and other POSIX-compatible systems), and can be easily ported to other operating systems. It will run out-of-the-box on most Linux systems where Python is installed by default, but when Python is not installed at the default location the first line of the zpl.py script might need to be changed accordingly. On Windows the Python interpreter (available from www.python.org) and the Python Win32 extensions (available from www.python.org/windows/win32) must be installed.

The host software is a command line program that accepts the following command syntax:

```
[python] zpb.py <port> [repeat N] go <file>
```

The python prefix can be needed on windows. The <port> can be COM1 etc on Windows systems, or tty0 etc. on Linux. Repeat N can be used to repeat each write command N times (the default is once). As explained above this will reduce the failure rate drastically, but it will also increase the download time with a factor N.

The <file> must be the name of the .hex file (extension can be omitted) that must be programmed into the target. Code and EEPROM are written with the values specified in the hex file, but configuration fuses information is ignored.

PICmicro firmware

The bootloader firmware is written in MPASM (the Microchip assembler). It assumes that the target runs at 40 MHz internal clock (10 MHz crystal with *4 PLL). The target chip can be specified on the MPASM command line, like

```
/DTARGET=18F242
```

The default target is the 18F452. The only aspect of the target that is important is the amount of FLASH, so when a target is not supported a supported target can be selected that has the same amount of FLASH. The clock frequency and baudrate can also be modified using command line defines, but this has not been tested. The code checks whether propper working at the specified effective clock frequency and baudrate is still possible. It is expected that the baudrate and the effective clock frequency must be kept at approximately the same ratio.

The bootloader is designed to be compatible with all members of the 18F family, but it has been tested only on the chips that I had available: 18F242, 18F252, 18F442, 18F452, 18F458.

Other uses

The protocol and relevant parts of the PC software and PICmicro firmware used for the bootloader could be adapted for use in a normal PICmicro application. This might save day when (one way) communication must be added to an application that already uses all available I/O.