P89V51RD2 and In-Application Programming (IAP)

Jan Waclawek, wek@efton.sk

Introduction

The **P89V51RD2**[1] is an 8051/8052-pin-compatible microcontroller by NXP (ex-Philips), with 64+8kB FLASH code memory, 768B internal RAM (ERAM/XRAM), 6-clock (x2) mode, and a couple of extended peripherals, such as the PCA unit, SPI interface and watchdog counter. The most remarkable feature is, however, that it's FLASH can be in-situ programmed (ISP) through UART; and also its self-programmability (in-application programmability, IAP).

The P89**V**51RD2 is a successor to the successful P89C51RD+[2]/P89**C**51RD2[3] line, which introduced the ISP/IAP paradigm to the higher-end FLASH-based 8-bit microcontrollers. Microcontrollers with very similar features - 8051-pin-compatible, with up to 64kB of FLASH, supporting IAP and ISP - are manufactured by multiple manufacturers, including NXP's own P89C66x/P89V66x[4], Atmel AT89C51RD2/AT89C51ED2[5] (successor to the Temic T89C51RD2), Nuvoton (ex-Winbond) W78ERD2A[6] and SST SST89E564RD/SST89E516RD[7]. Some of them offer also a mix of models with less FLASH (and Atmel also a model with 128kB FLASH, the AT89C51RE2) and 3V supply voltage. However, although they are remarkably similar, they usually significantly differ in the IAP/ISP method. Some even don't have factory-installed bootloader, even if they usually offer free bootloader firmware and associated PC software.

As Philips/NXP phased out the P89**C**51RD2 in favour of P89**V**51RD2, users started to complain about the rather different nature of the IAP procedure in the latter. This made NXP to introduce the P89**CV**51RD2[8] and family, which, although physically closer to the newer P89**V**51RD2, mimics closer the original IAP behaviour of the older P89**C**51RD2.

Family members

There are also modifications to P89V51RD2 offered by NXP, besides the three packaging options (traditional DIP40 and PLCC44, and the miniscule TQFP44 with 0.8mm pin pitch).

Devices with less FLASH are marked as P89V51R**B**2 for 16kB FLASH and P89V51R**C**2 for 32kB FLASH.

Low supply voltage (3V) devices are marked as "**LV**", such as in P89**LV**51RD2[9]. Unfortunately, NXP currently does not offer wide supply voltage devices in this family.

As most of the characteristics are the same across the whole family, often, "cumulative" marking such as "P89V51R**x**2" is used to denote any member of the family. However, throughout this document, "P89V51RD2" will be used consistently, marking also other members of the family as appropriate.

There were several versions of the P89V51RD2 datasheet issued by Philips/NXP. To avoid confusion, it is always a good idea to download the latest one. At the time of writing of this document, the latest datasheet is marked as "Rev.4 - 1. May 2007"[10].

P89V51RD2 and ISP

One of the key moments of success of the P89V51RD2's predecessors was the ability to program them in-situ through UART. This alleviated the need for a costly parallel device programmer, or even a specialised programming "cable"; which made these microcontrollers attractive for small enterprises and hobbyists, despite their higher price. It also allows easy field-update of firmware from any PC or other device equipped with standard serial port.

The P89V51RD2 continued in this trend, although with a slightly different communication protocol, and, more importantly, a different **bootloader entry method**. On the older models, bootloader was entered when a particular set of voltage levels was applied to various pins (including the PSEN/ pin).

On P89V51RD2, after reset, the bootloader waits for a predetermined time, until a "U" character (55h) arrives to the UART receiver. The datasheet states this time as "approximately 400ms", however, as it is in fact derived from the watchdog, this time is dependent on the system clock frequency, being around 400ms when f_{CLK} is approx. 3.5MHz; for other clock frequencies this time is proportionately shorter or longer.

This autobaud method has both its advantages and drawbacks. On one hand, it does not tie down any pin and does not require any extra hardware nor manipulation with shorting jumpers or switches, as the old entry method did. On the other hand, in applications where an external device sends continuous stream of data to the P89V51RD2's UART, upon reset, the bootlader may be entered inadvertently. The extra delay before the application itself starts, may be a hindrance in certain applications, too. Where a different entry method is required, bootloader version 7 offers alternatives - see below.

The autobauding process itself is not quite perfect either. It derives the UART's baudrate coefficient by measuring the time between a trailing and a leading edge on the RxD pin (i.e. duration of a "0" bit). It then starts the UART and waits for the "U" character. As the "measurement" involves a certain granularity (uncertainty in the edges detection by bootloader firmware), moreover on RS232/UARTs usually some small asymmetry between duration of 0s and 1s exists, there is a chance of errorneous detection of baudrate, even if a crystal normally allowing precise setting of a given baudrate is used[11]. The probability of correct baudrate detection generally increases with lower baudrates, so the conservative recommendation is to use 9600 Bauds or below, even if this may lead to increased programming times.

To start the autobauding, usually the P89V51RD2-containing device is simply powered up; however, in some cases a simple circuit triggering reset from some of the handshake lines (RTS, DTR) is built to the device, for added comfort. This then has to be handled by the PC-side software accordingly. (Note, that FlashMagic by default assumes such hardware to be present, which may cause unexpected behaviour of FM if this hardware is absent).

The ISP protocol is entirely ASCII based, and uses intelhex-like "records" to perform various operations (except for the response during FLASH readout, which can perhaps be described as "raw ascii hexadecimal with spaces"). This allows to use as a PC-side programming utility any general-purpose terminal emulator, such as the ubiquitous Hyperterminal, or Tera Term[12] on Windows, or minicom[13] on Linux/Unix-like OS. Autobauding can be performed "by hand", pressing and holding down the "U" key and relying on the keyboards autorepeat, while resetting the target device. Commands for device identification (which serves as the successful autobauding verification) and erasing can be either "typed in", or "uploaded" from a previously prepared file, if needed. The FLASH content itself, as it is programmed using the "normal" type 00 intelhex fields, found in the .hex files output from compilers and assemblers, can be simply "uploaded" from these files; to allow time for programming, an inter-line delay of some 100ms has to be set.

Even if possible, the above described "manual" method is rather tedious and may serve only as a backup emergency method of programming. The standard PC-side application for programming is FlashMagic (by ESAcademy)[14]. There is also a FlashMagic forum[15], with an extensive bootloading troubleshooting list[16].

Characteristics of FLASH in P89C51RD2

The code FLASH in P89C51RD2 consists of two big blocks:

- Block0, 64kB, mapped at 0000h-0FFFFh, intended to run the "normal" user application.

- Block1, 8kB, mapped at 0000h-1FFFh, containing the bootloader.

As the two blocks overlap in the 0000h-1FFFh area, which of them is "visible" is determined by two bits in the FCF special function register. Both bits and their meaning is in detail described in chapter 7.1.1 and Table 5 of the datasheet.

The FLASH can be written byte-by-byte. As is usual with FLASH, bytes have to be erased prior to be written. An erased byte contains 0FFh. FLASH can be erased by 128-byte sectors (pages), or by a whole block (using an external parallel device programmer, a third method, full chip erase, is also possible - this erases both blocks, the x2 flag and the security flag in one operation). During writing/erasing, execution is not possible, so the code writing to Block0 must reside in Block1 (i.e. the bootloader code has to be called to perform the write/erase in Block0).

The datasheet states endurance as 10.000 cycles and retention to 100 years, which together with the relatively small sector size makes the FLASH suitable for ocassionally rewritten data storage (EEPROM replacement e.g. for device setup parameters). The datasheet completely fails to specify erase and write times... Note, that the datasheet specifies a minimum clock frequency, 0.25MHz, for in-application programming. The datasheet does not specify supply current during programming, but a safety margin of a few tens of mA over the specified "normal" supply current, and decent supply decoupling, could never hurt.

According to the datasheet, there is a single (non-volatile) security bit, preventing reading of the FLASH using parallel programmer. This bit can be set both by parallel programmer and during ISP or IAP. This bit does not influence device readout through ISP (which has an independent security mechanism, based on a "serial number", see below), nor IAP.

Default bootloader

In FLASH Block1, there is a factory-programmed bootloader, enabling both ISP and IAP. From factory, Block1 also contains a core-mode monitor/debugger called SoftICE, but unfortunately NXP does not publish any more information on this piece of code except for its existence.

Note, that the bootloader will be erased when using chip erase in a parallel device programmer. The datasheet warns for this in Chapter 6.3.2. It is a good idea to read out and store the bootloader before performing chip erase, although an image of the bootloader can be downloaded from NXP (don't use the "upgrade" file for this purpose). Consult your device programmer's manual on detailed instructions.

The bootloader is around in various versions. Please note, that the versions below apply specifically for the P89V51R**D**2 - bootloaders for P89V51R**B**2 and P89V51R**C**2 are for some unknown reason numbered differently.

Version 4 was factory-programmed in older devices, but contained several errors, including a flaw preventing sector erase using ISP, and a relatively fatal error causing devices to be stuck permanently in the SoftICE mode, if inadvertently selected.

Version 5 was provided as a fix to these problems, and can be downloaded from NXP's site in both forms: as the binary (intelhex) image, and as an "upgrader"[17]. The image is intended to be programmed to Block1 using a device programmer. The "upgrader" contains both the image and a short utility, and is intended to be programmed using ISP to Block0, so when run afterwards, it rewrites Block1 itself. FlashMagic can be used to take care of the whole upgrading procedure. As a curiosity, neither the image nor the upgrade contains the softICE part of firmware.

Newer devices are factory-programmed with bootloader version 6; however, exact differences between version 5 and 6 are not known.

Version 7 is provided only as an "upgrader" from the FlashMagic website[18], and provides several alternative options for the bootloader entry method, including pin-level-dependent entry and permanently disabled bootloader.

Security of the user code in FLASH against unauthorised reading through ISP is accomplished through an elaborate mechanism, involving programming a "serial number" which serves as a pass-code. Once "serial number" of nonzero length is programmed, after each reset the ISP commands (except version information and block erase) can be used only if the "serial number" is entered. Block0 erase clears also the serial number. This mechanism does not influcence reading by a parallel programmer, nor IAP.

IAP

In-application programming (IAP) of Block0 FLASH in P89V51RD2 is performed through setting up a couple of registers, and making a call to a predetermined address - 1FF0h - in Block1. This, and a list of possible operations together with the related registers (Table 13, which we are not going to reproduce here, and the reader is requested to study it thoroughly), is roughly all the datasheet says about IAP. The reality is somewhat more complex.

As said above, the code performing programming of FLASH Block0 has to run from Block1. This is why the IAP routines are part of the default bootloader in Block1. So, to be able to run these routines, Block1 must be mapped as active at the lower part of code address space, 0000h-1FFFh. This is accomplished through clearing both SWR and BSEL bits in FCF special function register. Note, that this step has two consequences:

- the code "switching" the blocks (i.e. clearing both mentioned SFR bits) must lie above the "shared" area, i.e. within 2000h-FFFFh. It is handy therefore to create a short routine, which handles the FLASH block switching and the IAP entry point (1FF0h) call - see CallIAP routine in the example below - and locate it at some suitable high address. As it is with absolutely located routines, it must be made sure, that it is not overlapped with some other routine. In most applications, a location near the top of the available FLASH might be a suitable place.
- before the blocks switching, interrupts must be disabled. After blocks switching, the interrupt vector area at the beginning of code address space is occupied by Block1, with the default bootloader. As the default bootloader has no provisions for interrupt handling, any interrupt which would occur while Block1 is "visible", would execute some random code, leading to crash.

After return from the IAP routines, Block0 can be restored by setting BSEL bit (SWR bit is supposed to be set only by hardware), after which interrupts may be reenabled.

Using IAP, any byte in Block0 can be programmed, including the 0000h-1FFFh area. Care has to be practiced, of course, if areas with "living" code are programmed or erased, including the interrupt vector area. However, only bytes which contain 0xFF (i.e. which are erased) may be programmed. This means, that if a non-0xFF byte has to be reprogrammed, the 128-byte sector where this byte is located has to be erased. If other bytes in this sector have to be preserved, they must be stored into RAM (ERAM) before erasing and then reprogrammed back to that sector.

The IAP "protocol" contains confusing commands, too. There is a command for block 0 erase, which is a complete nonsense, as the routine calling IAP would be erased, too, and there would be no code to return to (an effective "suicide" of the application"). There is also a command for byte read, which is much easier to perform using some of the MOVC instructions.

Note also, that the datasheet does **not** specify **resources** used by the IAP routines. Even if these can be determined by disassembling the bootloader, there is no guarantee these will not change in some future versions. Some of the potentially problematic issues include:

- stack usage - the IAP routines perform at least two nested calls, so a conservative approach would be to reserve around 10 bytes of stack for the IAP
- register usage - a conservative approach would assume that all registers R0-R7 of the current bank, B and DPTR are changed bye IAP routines
- memory and SFR usage - it is unlikely that the IAP calls would use any memory and/or SFR (except the FLASH interface SFRs, which are undocumented anyway)
- sensitivity to register bank setting, and possible change to it - if the bootloader code would use absolute addressed registers, it would be sensitive to the particular register bank at the moment of IAP call. A conservative approach would be to stick to register bank 0 when calling IAP routines. However, it is unlikely that the IAP routines would actively change the bank.
- sensitivity to DPTR selection, and possible change to it - it is unlikely that the IAP routines would be sensitive to which DPTR is currently selected. It is also unlikely they would change the DPTR selection or modify the other than currently selected DPTR.

Timing of the IAP routines is also not specified. Read commands are certainly served within several tens of instruction cycles, but programming commands take certainly more time. A conservative estimate would be, that programming a single byte takes a couple of milliseconds, whereas a sector erase might take tens of milliseconds, and a block erase up to several seconds.

Timing of IAP routines, i.e. the time while the mcu is essentially out of the user's control, has to be taken into account not only for timer-based operations (including the PCA), but also for UART operation and SPI slave operations. Handshaking with the other-side device has to be employed wherever applicable.

Another timing-sensitive issue is the watchdog, which has to be served just before and after the IAP call, or, if this is insufficient, disabled during IAP (although disabling watchdog is generally a bad idea).

IAP and C

Unfortunately, most of the issues related to IAP call for solutions which are outside the scope of standard C, therefore relies on implementation-dependent details of the given toolchain, and entirely non-portable.

As there are many low-level issues involved in calling the IAP routines - allocating variables into particular registers, positioning the calling routine at an absolute address above 2000h - the lowest-level routine is best to be handled in assembler - most probably as a separate source file. It could contain a single routine, taking three parameters, which the routine would pick from the registers/addresses given by calling conventions of the given compiler, and move them into R1/DPTR/ACC. It would then

To place this routine to an absolute address above 2000h, a compiler/toolchain-dependent method must be used. Some toolchains allow to use ORG or similar directive in assembler routines. Other toolchains can handle absolute allocation during linking, either using a command-line switch, or through linker scripts. Consult the documentation of your toolchain for details, or try to find an example of similar character.

In C, calling a routine located at an absolute address is usually accomplished through a function pointer initialised to the given address; or a literal (a constant number) cast to a function pointer.

An example of P89V51RD2 IAP usage in SDCC can be found in the CMON51 project[19].

IAP example

The following assembly language example tries to show the usage of IAP in P89V51RD2 (although intended for the Metalink assembler[20], it could be easily modified for any standard '51 assembler following the Intel syntax):

```
$MOD52
FCF     EQU 0B1h

        ORG 0
        ljmp Reset

;---- interrupt vectors
;none here

;---- "main"
Reset:

;if we try to rewrite an already programmed byte without previously erasing it, this will fail:
        MOV      DPTR, #TestArea1
        MOV      A, #55h
        CALL     FLASH_PROGRAM_BYTE

;this is how a sector has to be erased prior bytes in it are being rewritten:
        MOV      DPTR, #TestArea2
        CALL     FLASH_ERASE_SECTOR
        MOV      DPTR, #TestArea2
        MOV      A, #0AAh
        CALL     FLASH_PROGRAM_BYTE

Stop:   SJMP     Stop

;----   data area, 128-byte sectors
;make sure that there is no code in the whole sector,
;as upon updating the data the whole sector gets erased

        ORG 0080h
TestArea1:
        db  0, 0

        ORG 0100h
TestArea2:
        db  0, 0

;end of TestArea2 at 017F

        ORG 0180h
;rest of code may go here

;---- FLASH API - must be located anywhere ABOVE 2000h

        ORG 03F00h

;programs a single byte in FLASH
;DPTR=addres, A=content of byte to be programmed
;the position to be programmed must be previously erased
FLASH_PROGRAM_BYTE:
         PUSH    IE           ;DISABLE INTERRUPTS
         CLR     EA
         MOV     R1,#02       ;SETUP OPERATION CODE -- write byte
         ANL     FCF,#0FCh    ;enable boot sector - !!! this command MUST be located ABOVE 2000h!!!
         CALL    01FF0H       ;call to ISP_API (modifies B register but no Rx)
         ORL     FCF,#001h    ;switch back to user FLASH
         POP     IE
         RET

;erases a 128-byte sector in FLASH
;DPTR=address of first byte of sector to be erased
FLASH_ERASE_SECTOR:
         PUSH    IE           ;DISABLE INTERRUPTS
         CLR     EA
         MOV     R1,#08       ;SETUP OPERATION CODE -- erase sector
         ANL     FCF,#0FCh    ;enable boot sector - !!! this command MUST be located ABOVE 2000h!!!
         CALL    01FF0H       ;call to ISP_API (modifies B register but no Rx)
         ORL     FCF,#001h    ;switch back to user FLASH
         POP     IE
         RET

;----- that's all, folks
         END
```

This example tries to illustrate several facts:

- the IAP API calling routines are located above 2000h

- these routines explicitly disable interrupts (even if in this "application" the interrupts are not used at all)

- the data area must be erased before rewriting

After assembling, program the resulting hex-file into a P89V51RD2 using FlashMagic. Perform also a verification - it should verify OK. If you read out the data, the two data areas should read as:

```
0080: 00 00 FF FF FF ...
```

and

```
0100: 00 00 FF FF FF ...
```

Now reset the P89V51RD2, so that the application may run - it needs less than a second. Now enter again bootloader mode with FlashMagic (e.g. by reading the signature bytes), and perform a verify: it will fail. Reading the data should show, that the first sector - which was not erased in the "application" - remained unchanged, and the second got erased and its first byte rewritten:

```
0080: 00 00 FF FF FF ...
0100: AA FF FF FF FF ...
```

References:

[1] http://www.nxp.com/pip/P89V51RD2.html
[2] http://www.nxp.com/pip/P89C51RD+.html
[3] http://www.nxp.com/pip/P89C51RD2.html
[4] http://www.nxp.com/pip/P89C668.html
[5] http://www.atmel.com/dyn/products/Product_card.asp?part_id=3044
[6] http://www.nuvoton.com/hq/enu/ProductAndSales/ProductLines/ConsumerElectronicsIC/Microcontroller/80C51Microcontroller12T/W78ERD2A.htm
[7] http://www.sst.com/products/?inode=41314
[8] http://www.nxp.com/pip/P89CV51RB2.html
[9] http://www.nxp.com/pip/P89LV51RB2.html
[10]http://www.standardics.nxp.com/products/80c51/datasheet/p89v51rb2.p89v51rc2.p89v51rd2.pdf
[11]http://www.efton.sk/t0t1/autobaud_error.htm
[12]http://ttssh2.sourceforge.jp/
[13]http://alioth.debian.org/projects/minicom/
[14]http://www.flashmagictool.com/
[15]http://forum.flashmagictool.com/
[16]http://forum.flashmagictool.com/index.php?topic=3232.0 and link given therein
[17]http://www.standardics.nxp.com/support/documents/microcontrollers/zip/boot.loader.p89(l)v51rd2.zip
[18]http://www.flashmagictool.com/assets/resources/P89V51Rx2_Bootloader_Update.zip
[19]http://cmon51.sourceforge.net/
[20]http://www.metaice.com/ASM51/ASM51.htm