Tales of memory and 8051

Contents

Copying

Introduction

In the eye of a 21st century firmware developer, the memory organisation of the 8051 and its descendants might seem unbelievably, and unnecessarily awful. The AVR MCU family, for instance, despite being born in the same ancient times as the 8051, has a straightforward memory organisation.

However, with a clear explanation of this peculiar memory organisation, using a contemporary 8051 is as easy as using any other 8-bit MCU.

Harvard architecture

The most visible of the 8051's peculiarities is the separation of program memory and data memory, which is referred to as "Harvard architecture".

The nice thing, from a developer's perspective, is that with 16-bit addresses, you can have both 64KB program memory **and** 64KB data memory – which was not possible on other 8-bit CPU (e.g. the famous Z80) without added complexity and constraints.

The price to pay for this comfort is a specific instruction to read static data (e.g. menus, error messages, look-up tables) from program memory, which after all isn't a big deal, at least for the developer.

Data memory

When the 8051 was originally designed, 1KB was considered a comfortable RAM size, so 128 bytes were deemed largely enough for a microcontroller. Then, why bother using 16-bit addresses for data memory access? 8-bit addresses would do most of the time, wouldn't they?

And for those (supposedly) few applications requiring more RAM, the 8051 offered the possibility to use external data RAM, with specific transfer instructions taking their 16-bit address from a specific register, the DPTR.

It turned out to be a not-so-brilliant idea, but once the tools and the code base are there, hardware improvements cannot break backward compatibility.

This is why, in order to improve the speed of memory transfers, today's 8051 usually provide a dual-DPTR, allowing to switch between the two with a bit in a SFR.

Addressing modes

An addressing mode is a specific way to access data. Common addressing modes include:

- immediate: the data is constant and provided immediately after the opcode,
- direct: you provide the address of the data,
- indirect: you provide the address of the location of the address of the data,
- indexed: you provide the address of the data as a base address and an offset,
- register: the "address" of the data is the name of a register of the CPU,
- implied: the nature of the instruction determines the location of the data.

Note: with indirect and indexed addressing modes, addresses may be memory addresses or registers; the offset is also usually stored in a register.

The 8051 provides variations and combinations of these basic addressing modes as well as specific ones (e.g. bit addressing).

Why is this important? Well, because accessing specific memory regions can be tied to specific addressing modes.

The most obvious example is the sharing of the 0x80-0xff address range between RAM and special function registers, as we'll see later on.

Special function registers (SFR)

In the 8051 terminology, special function registers are what is more generally called "memory-mapped I/O", i.e. memory locations controlling peripherals (e.g. GPIO, timers, UART). In the original 8051, the 0x80-0xff address range was used for SFR, and there still was room for improvements.

Of course, today's 8051 descendants offer a lot more features than their ancestor, hence need a lot more memory space, which is obviously a threat to backward compatibility. Bank switching is used to solve this dilemma.

Bank switching

A common technique for extending the amount of addressable memory with a fixed address width is to define regions of the address space that can be switched between different storage units ("banks").

Because SFR need more memory than the 128 bytes originally reserved for them, a specific SFR will be dedicated to the selection of the "active" memory bank.

With Nuvoton's N76E003, the 0x80-0xff data memory range can be switched between 2 SFR banks, offering a 255-byte SFR space in a 128-byte data memory space.

STC chose to switch the extended memory space between RAM and SFR, keeping the lower 256 bytes of data memory unaffected by the switch, and potentially offering 64KB for SFR.

The sharing of the 0x80-0xff space between RAM and SFR is another form of bank switching, this time determined by the addressing mode used to access this particular region: indirect for RAM access, direct for SFR access.

Finally, the 8051 has 8 data registers named R0 to R7 that are also memory-mapped. However, their state has to be preserved when handling an interrupt, which would take a lot of time and use a significant portion of the scarce stack space if they were individually saved. The decision was thus made to provide 4 banks of registers so that context could be preserved just by switching the "active" register bank.

Data memory segments

Memory region have names whose meanings, of course, evolved over time. For instance, today's 8051 microcontrollers provide more than 256 bytes on-chip RAM, so "XDATA" more appropriately refers to "extended data RAM" than to "external data RAM".

Anyway, here's the current 8051 memory terminology:

- **code**: Program memory. Can be read using the MOVC instruction.
- **data**: The first 128 bytes of data memory.
- **bdata**: Refers to the bit-addressable portion of DATA, i.e. what is more generally called "bit-banding", in the 0x20-0x2F range.
- **idata**: The 128 bytes of data memory in the 0x80-0xff range, accessed exclusively with **indirect** (The "I" in IDATA) addressing, i.e. MOV @R*n* instructions. Note: The combination of DATA + IDATA may be referred to as "scratch-pad RAM".
- **sfr**: The 128 bytes of data memory in the 0x80-0xff range, accessed exclusively with **direct** addressing.
- **xdata**: Extended data memory accessed indirectly, using the MOVX @DPTR instruction. As the "X" in MOVX suggests, it's a distinct address space, meaning address 0 in XDATA is (normally) not the same as address 0 in DATA.
- **pdata**: The first 256 bytes of XDATA, accessed indirectly using the MOVX @R*n* instruction (R0 and R1 only).

These names are important because they can be used in C to specify the allocation class of an object, which determines the instructions used to access it, as well as where it will be stored.

Memory models

How much RAM you have on your 8051 will define where you'll store your variables:

- If you only have 128 bytes of RAM, you'll only be able to use the DATA segment for everything. Nowadays, you should – fortunately! – not face such an extreme (and desperate) situation.
- If you have 256 bytes of RAM, you'll be able to use the DATA and IDATA segments, and may prefer using IDATA for your variables so as to leave the 0x30-0x7f range for a small stack.
- If you have 512 bytes of RAM, i.e. 256 bytes scratch-pad RAM + 256 bytes extended RAM, you may want to use the 0x30-0xff range for a larger stack and the PDATA segment for your variables.
- If you have more than 512 bytes of RAM, you'll have the full capacity of XDATA for your variables. Congratulations!
- If you have more than 64KB RAM, using some form of bank switching... Wait, wait, wait! Nowadays, if you need more than 64KB RAM, your application will likely need more features and more compute power than an 8051 can offer, so you'll be using an ARM / RISC-V / C-Sky MCU instead, period.

When developing in C, storage allocation strategies are called "memory models", and you use a compiler option to select the memory model that fits your application's needs:

- small: allocate variables in IDATA by default, for really small applications.

- **medium**: allocate variables in PDATA by default, for small to medium applications.
- **large**: allocate variables in XDATA by default, for larger applications.
- huge: allocate variables in XDATA by default and use bank switching. As explained earlier, forget it.

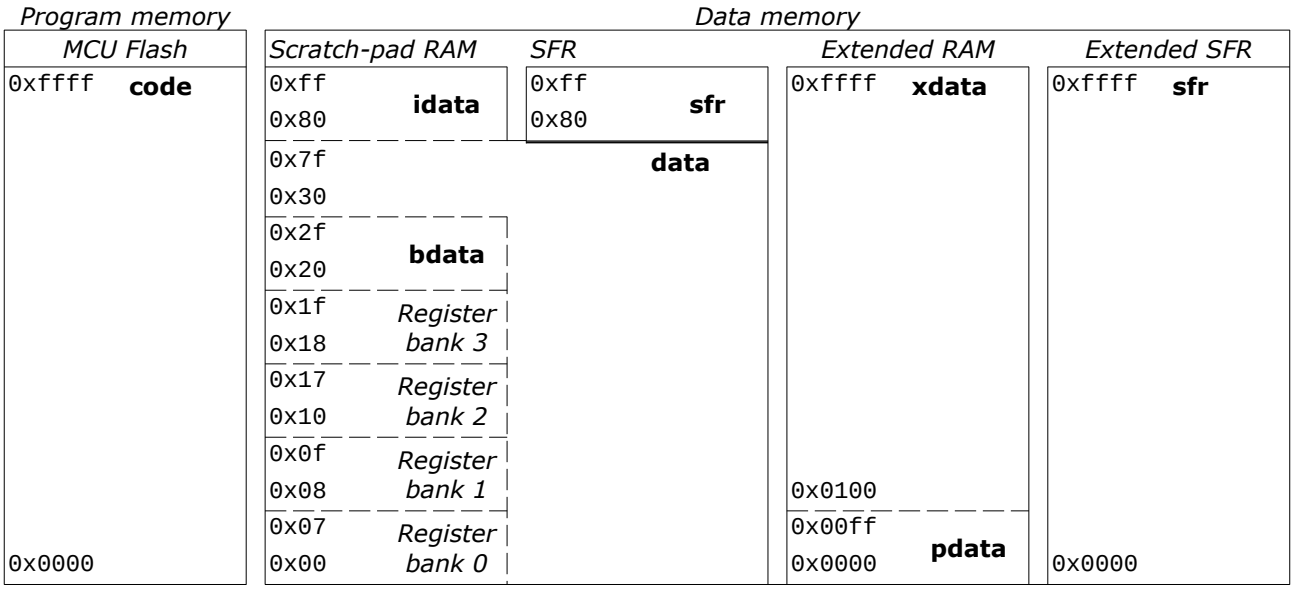
Specifying a memory model as a compiler option will save you the tedium of manually declaring the allocation class of each variable, making your code much more readable and maintainable.

But of course, if there's slightly not enough room in the default segment you chose, you can manually assign a few variables to a different one (e.g. idata, when pdata or xdata is full) so the linker can do its job.

You can also explicitly assign specific variables to specific segments to guarantee a faster access to these data. For instance, if your application uses the large memory model and experiences timing issues when handling interrupts, you could try and assign variables shared by ISR and the main loop to pdata to save a few cycles on each access.

The Big Picture

A picture is worth a thousand words, here's a quite talkative one:



The Devil's share

The Devil is said to be in the details, here are a few.

Memory access illustrated: assembly snippets

BDATA access (bit variables only)

```
; Set bit at BDATA address 0x00.
; Cost: 2 bytes, 1 cycle on an STC8 MCU
SETB 0x00
```

DATA access

```
; Store immediate value 0x55 at direct DATA address 0x30.
; Can also be used to access *SFR* in the 0x80-0xff range.
; Cost: 3 bytes, 1 cycle on an STC8 MCU
MOV 0x30, #0x55
```

IDATA access

```
; Store immediate value 0x55 at IDATA address 0x80.
; Can also be used to access *RAM* in the 0x00-0x7f range.
; Cost: 4 bytes, 2 cycles on an STC8 MCU
MOV R0, #0x80
MOV @R0, #0x55
```

XDATA access

```
; Store immediate value 0x55 at XDATA address 0x0180.
; Can also be used to access *extended* SFR with bank switching.
; Cost: 6 bytes, 4 cycles on an STC8 MCU
MOV DPTR, #0x0180
MOV A, #0x55
MOVX @DPTR, A
```

PDATA access

```
; Store immediate value 0x55 at PDATA address 0x0084.
; Cost: 5 bytes, 5 cycles on an STC8 MCU
MOV R0, #0x84
MOV A, #0x55
MOVX @R0, A
```

CODE access (read-only)

```
; Read byte at address 0x0240 in program memory
; Cost: 5 bytes, 6 cycles on an STC8 MCU
MOV DPTR, #0x0240
CLR A
MOVC A, @A+DPTR
```

External memory

Because the 8051 has been in use for 40+ years, a whole lot of information is available online, a significant portion of which being no longer relevant, particularly when memory-related. This is why, sometimes, a few words about the past help focus on the present.

The original 8051 used 2 GPIO ports plus 5 signals to access external memory. Here's how it worked.

External **data** memory **read**

- The ALE (Address Latch Enable) pin was driven high.
- Bits 0..7 of the address were pushed to P0 and bits 8..15 to P2.
- The external memory chip was expected to latch the address on the falling edge of ALE.

- The $\overline{RD}$ (data ReaD) pin was driven low so the external memory chip knew it's time to deliver data to P0, then latched by the 8051 on the rising edge of $\overline{RD}$.

External **data** memory **write**

- The ALE pin was driven high.
- Bits 0..7 of the address were issued on P0 and bits 8..15 on P2.
- The external memory chip was expected to latch the address on the falling edge of ALE.
- The 8051 issued data on P0.
- The $\overline{WR}$ (data WRite) pin was driven low so the external memory chip knew it's time to store the byte present on P0.

External **program** memory **read**

- In order to use an external program memory chip, the $\overline{EA}$ (External Access enable) pin had to be permanently tied to GND.
- The ALE pin was driven high.
- Bits 0..7 of the address were issued on P0 and bits 8..15 on P2.
- The external memory chip was expected to latch the address on the falling edge of ALE.
- The $\overline{PSEN}$ (Program Store ENable) pin was driven low so the external memory chip knew it's time to deliver data to P0, then latched by the 8051 on the rising edge of $\overline{PSEN}$.

  By the way, have you noticed that the only difference between program memory read and data memory read is the use of $\overline{PSEN}$ instead of $\overline{RD}$? This detail is what reveals a Harvard architecture on the hardware side.

Today, less and less 8051 descendants allow the use of external memory, so this no longer matters much. However, it's good to know this because you might come across forum posts, source code, or documentation where this is mentioned or implied. Having heard about it, you'll know you should disregard these information.

Another reason for mentioning this is that recent 8051 descendants – such as the STC8H family, and the not yet generally available STC32G – still have an EXTRAM bit in their AUXR SFR. If you accidentally turn it on, you may notice undesirable behaviour on the P0, P2 and P3 GPIO lines, as well as erratic firmware behaviour. But as long as EXTRAM is disabled, GPIO ports are yours forever.

XPAGE

Most 8051 descendants define PDATA as the first 256 bytes of the XDATA segment. However, some microcontrollers did allow to change the location of PDATA by using a special SFR called XDATA to define the high-order byte of the PDATA address.

You're unlikely to use such a microcontroller nowadays, so if you ever see XDATA mentioned somewhere, you'll know it doesn't apply to your situation.