Using STC MCU with open-source tools

Contents

Whats, whys, and whatnots

Copying

Credits

Feel free to email me if you notice any error or omission (vincent *dot* defert *at* posteo *dot* net).

Introduction

All the code provided by STC is for Keil's C51 compiler, which is not exactly suited for the hobbyist or the self-learner. Fortunately, SDCC and STCGAL provide a good open-source toolchain for 8051 MCU that can run on any operating system, in particular Linux distributions, but also BSD operating systems.

SDCC was deemed good enough (and adopted enough in the professional sphere) for Silicon Labs to publish an application note (AN198) describing how to integrate it in their IDE, which can be seen as a reliable quality indicator. And that was in 2005.

The only downside of SDCC is that it currently targets only MCS-51 devices and will not be able to take advantage of all the capabilities of the STC16F and STC32G MCU, which are MCS-251 devices.

I'll play with these new MCU later – I got my hands on a few STC16F40K128 and STC32G12K128-Beta! :) :) :)

For now, the information given in this document has "only" been tested on STC90, STC12, STC15 and STC8 MCU.

Why use an 8051 MCU in the 21st century?

The 8051 family began shipping in 1980 and, at the time, it was revolutionary: it offered in a single chip what until then required at least 5 or 6 chips!

However, more than 40 years have passed since then, and even though the 8051's descendants are much more capable than their ancestor, today's norm is to use ARM / RISC-V / C-Sky microcontrollers that outperform them by several orders of magnitude, so asking why to use an 8051 descendant today is quite legitimate.

The answer can be formulated as another question: why use a supersonic jet air plane to go just 500 metres away?

As a business

A great many useful applications require very few resources to be addressed, and an 8-bit microcontroller is not only quite capable in such situations, but also much more cost-effective.

When considering costs, people usually only focus on supply and manufacturing costs to make their decisions. However, other less obvious factors affect the profitability of a product, calling for a bit of risk management.

As is often the case, your hardware and firmware engineers may already work at full capacity, thus not being available to develop a new product. Then, qualified personnel is scarce, both for you and your subcontractors, so in order not to miss a business opportunity, you might need to hire someone with little experience and train them on the job.

Your new hire will need to deal with a lot more complexity with, say an ARM MCU, than with a modern 8051, and this person will thus more frequently interrupt his colleagues to get the information or help he needs, impacting their own performance. The learning curve of more complex devices is also necessarily longer.

With a simple product using a simple technology, your new hire will have gained experience, learnt to work with his colleagues, and got used to your company's procedures. At this point, there will be much less for this person to learn in order to work on a product requiring a more complex technology.

Also, if you use the same chips as everyone else for everything you do, you'll have to compete with everyone else to source your parts. In chip shortage times, this means it may even not be worth starting the project. Using different MCU technologies for different needs may help mitigate this risk.

As you can see, using modern 8051 MCU may still make sense today in some specific situations.

As an individual

Some people practice electronics and firmware development as a hobby, others do so in order to prepare a career change. In either case, using modern 8051 makes a lot of sense.

8051 descendants offer low pin count packages (SOP8, SOP16, TSSOP20, LQFP32), and old parts (say, 2015-2020) even offer DIP packages, making breadboarding a breeze. Most of them also have an internal RC oscillator and accept a wide supply voltage range.

You have an idea? Just slap a chip on a board, add a pair of decoupling capacitors and you're all set!

Furthermore, even though recent 8051 MCU (e.g. STC8H parts) offer up-to-date peripherals, these are very simple to use. You just read the corresponding part of the reference manual and your program is written within minutes. You don't have to get used to – and sometimes struggle with – vendor HAL and IDE.

As an individual, you have very little free time to spend on your hobby or self-training, and you're usually tired after a day (or a week) of work when this free time comes, at last. Using parts matching the modest level of complexity you can afford in your projects is essential to being able to make this scarce free time **a creative and rewarding moment**.

Moreover, in the case of a self-training, you absolutely need to divide to rule. There's nothing much complicated in embedded software, but there's really a lot to learn, and you may easily feel overwhelmed and discouraged.

Starting with an 8-bit MCU (e.g. 8051) helps you make your learning progressive enough to avoid these pitfalls.

Everything you'll learn on these simple devices will help you painlessly learn 32-bit MCU later, and enjoy the additional power and flexibility they bring without being discouraged by the additional complexity that comes with them. Learning with an 8-bit MCU is a learning that scales well.

Finally, the idea is also to use professional-grade tools from the start, so that switching to vendor tools later will be as fast and effortless as possible.

Why use an STC MCU?

There are objective reasons to choose to use an STC MCU:

- They're cheap and very easily available. LCSC has always some stock, and you can easily find cheap breakout boards on AliExpress for quick breadboarding.
- Using them doesn't necessitate sophisticated or expensive software or equipment.
- They're supported by good open-source tools.
- Their documentation is available both in Chinese and in English.
- They are widely used, and many resources are available online.

Besides these, there are also subjective reasons:

- STC are committed to make it possible for anyone to learn and use technology. They give away e-books, finance and equip training centres across China, and (co-)organise competitions.
- STC are also committed to offer a KISS alternative to mainstream ARM technology, and are thus constantly improving their technologies, as illustrated by their STC8H and STC32G families.

Using their MCU provides an occasion to witness their effort and, to some extent, to share a little bit of this adventure. I don't know any other silicon vendor capable of offering this.

Why use open-source software?

Though the open-source movement started as a rebellion against abusive commercial licenses, there's a reason why it got adopted by IBM and Google at the turn of the century, and why the rest of the industry followed, including even Microsoft!

Let's take the emblematic example of Linux, an operating system kernel. Let's also consider 2 major server and workstation vendors of the time, Sun and HP. Do you think their customers, when making their choice, did browse the source code of the kernels of Solaris and HP-UX?

This example highlights the fact that a general-purpose operating system kernel, such as Linux, is not a key differentiator in a company's offer. It's an enabler, not a differentiator – you must have it to play the game, but it doesn't make you any better than your competitors by itself.

As a consequence, all the capital you spend on your OS kernel will not pay back, because it's not what will make you win sales. And still, you have to spend it, otherwise you wouldn't even have a single opportunity to sell.

This is why, in an ideal world, all the competitors on a given market would share the development of enablers so as to maximise the capital spent on their differentiators. And this is exactly what open-source allows.

In other words, **open-source software actively contributes to capital efficiency.**

Even Microsoft finally understood that, to the point of migrating their Microsoft Azure infrastructure to Linux. Wow!

The same process can be observed today with RISC-V: an ISA and the tools needed to make it any useful (e.g. compilers) are enablers, so the cost of their development should be shared by all the actors willing to create and sell profitable products and services on top of it in order to maximise said profit.

Prerequisites

Software

We will be using the SDCC toolchain to build our firmware, and STCGAL to upload the compiled binary firmware to the MCU.

- http://sdcc.sourceforge.net/
- https://github.com/grigorig/stcgal

SDCC and STCGAL can be used on Linux, macOS, FreeBSD, NetBSD, OpenBSD and Microsoft Windows.

SDCC can be downloaded from its SourceForge page, but is also very likely already available as a prebuilt package of your Linux distribution, or your BSD flavour.

STCGAL is a Python application, so it can simply be installed with PIP.

Besides these essential tools, you'll need:

- a text editor (I like Geany and Neovim),
- a build management system (I provide you with Makefile examples, but you can also use Tup or Meson if you want),
- a terminal emulator (e.g. Minicom, CuteCom),
- and, optionally, doxygen for API documentation.

Hardware

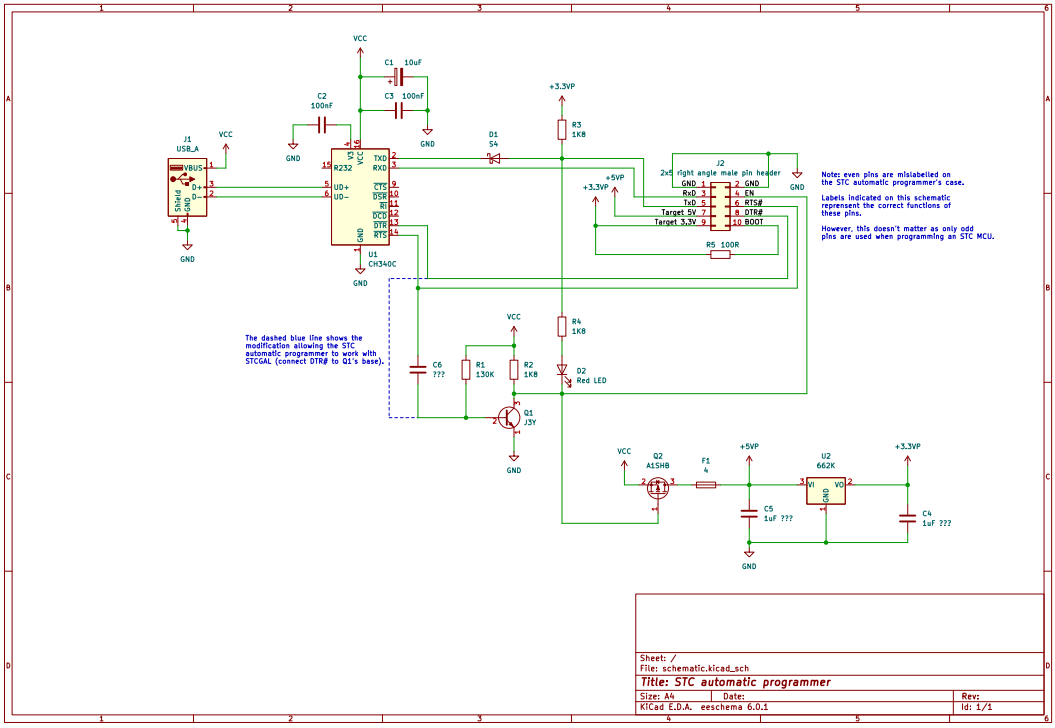
STC Auto Programmer

Besides the usual breadboarding stuff, you'll only need a USB-to-serial adapter. However, as the MCU programming procedure requires to power cycle it, you may find much more convenient to buy, in the same low price range, an "**STC Auto Programmer USB-TTL**". You can easily find it on AliExpress, for instance. Here's what it looks like:

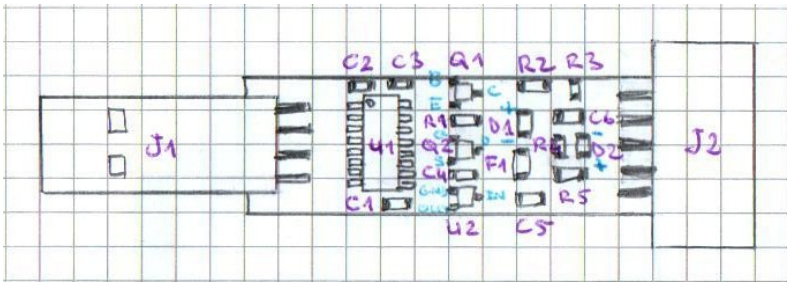
However, this adapter was designed for use with STC-ISP and the way STCGAL power cycles the MCU is a little different, which will require a little modification.

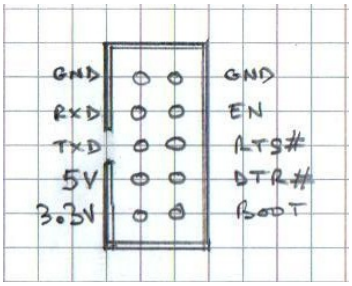
Hardware modification for use with STCGAL

Apparently, STC-ISP wiggles $\overline{\text{RTS}}$ to power cycle the MCU, whereas STCGAL just asserts $\overline{\text{DTR}}$ low for 0.5 second, so the required modification is pretty straightforward, as shown on the schematic below:

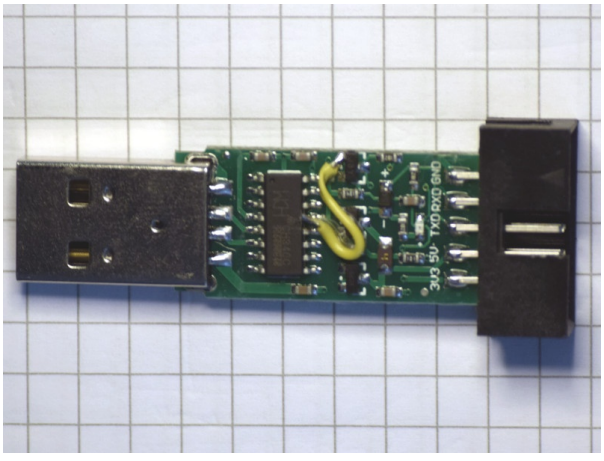


Here are also the board layout and connector pinout for reference:





As you can see, the modification is as easy as a piece of wire and 2 drops of solder:



Compiling firmware with SDCC

First and foremost

If you're not quite familiar with the very peculiar memory architecture of the 8051, please do read the "Tales of memory and 8051" appendix before going any further.

Header files

An 8051 is a simple MCU and all its features can be described in a single C header file. STC's documentation uses Keil's 8051.h or 8052.h and defines STC-specific SFR in the C code, which is acceptable for a small example, but not for professional-grade development.

STC sometimes provides MCU-specific header files, but they're not always complete and they use the Keil syntax.

This is why I set out to write header files for each tested MCU family. To facilitate the writing and checking of these files, I described the SFR in CSV files, and wrote a header file generator to produce the final .h files.

While I was at it, I added definitions for interrupt numbers and vectors, SFR bank switching inline functions, comments describing the supported MCU, and a few useful macros such as F_CPU.

Because of a few inconsistencies in SFR and bit naming, I had to change some names to resolve conflicts. If you have an "undefined symbol" error when using the names found in the reference manual, just check the generated header to find the non-conflicting name I used.

Because there are many variants in each STC MCU family, I also decided to describe the most versatile member of each family and all their less capable variants.

For instance, in the STC8H family, I decided to describe the STC8H8K64U (which has USB!) as the most capable, and the STC8H3K64S4, STC8H3K64S2, STC8H1K28 and STC8H1K08 in decreasing order of feature set.

However, as they target more specific use cases, I didn't describe the STC8H1K08TR, STC8H2K64T, STC8H4K64TLR, STC8H4K64TLCD and STC8H4K64LCD. Should you need them, just update the definitions in STC8H.csv as needed and regenerate the .h.

The header generator and a few CSV files are available in the **./header-generator/** directory. Note the header generator is written in Ruby, so you need the ruby package installed on your system.

Producing a C header file from a CSV description is as easy as:

./generate-header STC8H.csv stc8h.h

I've already done this for all the supplied CSV files, so you likely just need to pick the header file matching your MCU and copy it to your project directory.

Dual-DPTR support

If the MCU you're using has a dual-DPTR and your project uses XDATA RAM, you must copy the /usr/share/sdcc/lib/src/mcs51/crtxinit.asm file to your project directory and change its "DUAL_DPTR = 0" line to "DUAL_DPTR = 1".

This file must then be assembled using the command "sdas8051 -plosgff crtxinit.asm" and linked with the other files of your project.


Don't worry too much about this, it's already covered in the examples provided in the **./makefile-examples/** directory.

Also note that SDCC expects an MCU with a dual-DPTR to define an SFR named DPS, whose least significant bit determines which DPTR is used. Instead of modifying crtxinit.asm for each MCU family, I just added an aliased SFR definition in the CSV files whenever the STC documentation doesn't use the name expected by SDCC.

Extended SFR support

All STC8 and newer MCU families (as well as a some STC12 and STC15 chips) provide more functionalities than the 0x80-0xff address range has room for, so advanced functionalities use extended SFR, accessed through DTPR after switching from the extended RAM bank to the extended SFR bank.


Inline function are provided so you can just call **enableExtendedSFR()** before using extended SFR and **disableExtendedSFR()** right after, and be done with bank switching.

Compiler options

I use the following options when compiling C source files:

- **-mmcs51** specifies the target architecture. MCS-51 is the default one, but as I use the same Makefile skeleton for different MCU architectures, I prefer to specify it explicitly.
- **--model-medium** or **--model-large** depending on the available amount of xdata RAM (medium memory model for 256 bytes, large memory model otherwise).
- **--opt-code-speed** to optimise for speed. Because STC MCU don't support JTAG debugging, producing a debug executable is not relevant.
- And of course, -c and -o.

I also pass 2 macro definitions to the compiler:

- **-DF_CPU=**nnnnnnnnnUL to specify the system clock frequency in Herz as an unsigned long constant. This is very useful to program timers or PWM generators without hard-coded numbers, or to implement delay loops.
- STC89 / STC90 only: -DT_CPU=nn (with nn = 6 or 12) to specify the MCU mode (6T/12T).

Linker options

Like the compiler, the linker needs to know the MCU architecture and memory model, but it also needs the sizes of the different memory segments:

- **-mmcs51** (same as compiler).
- **--model-medium** or **--model-large** (same as compiler).
- **--stack-size** 128 (adjust if needed)
- **--xram-size** 8192 (must match your MCU's)
- **--code-size** 65536 (must match your MCU's)
- And of course, -o.

Output files

The compiler's object files have a **.rel** suffix instead of the usual .o.

For each C source file, the compiler also produces an assembly source file (**.asm**) and assembly listing file (**.lst**, with addresses and machine code).

The linker produces several files:

- The **.lk** file contains linker options.
- The **.ihx** (Intel hex) file contains your binary firmware. You'll pass it to STCGAL when you'll program your chip.
- The **.mem** file contains a summary of data memory usage. It is very useful when you run out of memory in the default segment corresponding to your memory model: it helps you decide if you can move some variables to idata, or if you have to use another more capable MCU, possibly not an 8051.
- The **.map** file lists the addresses of each symbol in your code. Like the .lk and .asm files, the .map file is very interesting to understand how the compiler and linker work.

Using .mem and .map files

Let's suppose you're using an STC15W408AS for a simple application. This MCU has a total of 512 bytes on-chip RAM split between 256 bytes scratch-pad RAM and 256 bytes extended RAM, so you chose the medium memory model for your application.

At some point in the development, the linker will complain it doesn't have enough extended RAM for all your variables. So you open the **.mem** file and see this:

```
Internal RAM layout:
      0 1 2 3 4 5 6 7 8 9 A B C D E F
0x00:|0|0|0|0|0|0|0|0|1|1|1|1|1|1|1|1|
0x10:|a|a|a|a|a|a|b|b|b|b|b|b|b|b|b|b|
0x20:|T|c|c|c|c|c|c|c|c|c|c|c|d|d|d|d|
0x30:|d|d|d|d|S|S|S|S|S|S|S|S|S|S|S|S|
0x40:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x50:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x60:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x70:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x80:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x90:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xa0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xb0:|S|S|S|S| | | | | | | | | | | | |
0xc0:| | | | | | | | | | | | | | | | |
0xd0:| | | | | | | | | | | | | | | | |
0xe0:| | | | | | | | | | | | | | | | |
0xf0:| | | | | | | | | | | | | | | | |
0-3:Reg Banks, T:Bit regs, a-z:Data, B:Bits, Q:Overlay, I:iData, S:Stack, A:Absolute

Stack starts at: 0x34 (sp set to 0x33) with 128 bytes available.
The largest spare internal RAM space starts at 0xb4 with 76 bytes available.

Other memory:
   Name             Start    End      Size     Max
   ---------------- -------- -------- -------- --------
   PAGED EXT. RAM   0x0001   0x0107     263      256
   EXTERNAL RAM                          0      256
   ROM/EPROM/FLASH  0x0000   0x1f3e    7999     8192
*** ERROR: Insufficient EXTERNAL RAM memory.
```

The error message on the last line is crystal clear, but the linker also gives you a few other useful pieces of information: you would need **263** bytes for your variables but have only **256**; however, **76** bytes are still available in the scratch-pad RAM.

Fortunately, 263 – 256 = 7, which is < 76, so if you move at least 7 bytes from PDATA to IDATA, your build will succeed.

[By the way, note the linker inaccurately uses the adjectives INTERNAL and EXTERNAL for memory segments that are both located in on-chip RAM.]

Now, let's suppose you have 2 unsigned long variables in your application, initially declared like this:

```
static unsigned long v1;
static unsigned long v2;
```

Because you use the medium memory model, they're using a total of 8 bytes in PDATA, so storing them in IDATA would solve your problem:

```
static unsigned long __idata v1;
static unsigned long __idata v2;
```

So your **.mem** file now shows:

[Note: the variables I moved to IDATA were not unsigned long, so values are a little different from theory.]

```
Internal RAM layout:
      0 1 2 3 4 5 6 7 8 9 A B C D E F
0x00:|0|0|0|0|0|0|0|0|1|1|1|1|1|1|1|1|
0x10:|a|a|a|a|a|a|b|b|b|b|b|b|b|b|b|b|
0x20:|T|c|c|c|c|c|c|c|c|c|c|c|d|d|d|d|
0x30:|d|d|d|d|I|I|I|I|I|I|I|I|I|I|I|I|
0x40:|I|I|I|I|I|I|S|S|S|S|S|S|S|S|S|S|
0x50:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x60:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x70:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x80:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x90:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xa0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xb0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xc0:|S|S|S|S|S|S| | | | | | | | | | |
0xd0:| | | | | | | | | | | | | | | | |
0xe0:| | | | | | | | | | | | | | | | |
0xf0:| | | | | | | | | | | | | | | | |
0-3:Reg Banks, T:Bit regs, a-z:Data, B:Bits, Q:Overlay, I:iData, S:Stack, A:Absolute

Stack starts at: 0x46 (sp set to 0x45) with 128 bytes available.
The largest spare internal RAM space starts at 0xc6 with 58 bytes available.

Other memory:
   Name             Start    End      Size     Max
   ---------------- -------- -------- -------- --------
   PAGED EXT. RAM   0x0001   0x00f5     245      256
   EXTERNAL RAM                          0      256
   ROM/EPROM/FLASH  0x0000   0x1f35    7990     8192
```

The 'I' letters in lines 0x30 and 0x40 represent the bytes used by the variables moved to IDATA. IDATA? But wait, 0x30 and 0x40 are in DATA, aren't they?

Of course they are, but remember I told the compiler to **optimise for speed**, and access to DATA is faster (direct addressing is allowed in DATA, not in IDATA), so my variables were moved to DATA instead of IDATA.

The **stack** still spans across DATA and IDATA, but it doesn't matter as stack access is always indirect (through the stack pointer).

Now, if the space needed in excess of PDATA's 256 bytes was **much higher** than the available space in scratch-pad RAM, you would have no other choice than using another, more capable MCU.

However, if the space needed was only **a little higher** than what's available, another strategy might help. If you look at the **.map** file, you'll see that function parameters can also use RAM in the default segment of the selected memory model.

This means that if you have a lot of functions in your application, you can also reduce their RAM usage, though at the expense of readability and maintainability, either by using global variables, and/or by using inline functions when possible.

The use of inline functions considerably slows down compilation, but it might – quite understandably – be preferred over using global variables.

Finally, the **.map** file may also help you spot large variables in order to minimise the number of allocation specifier changes, though you can probably figure it out yourself by looking at structure and/or array definitions.

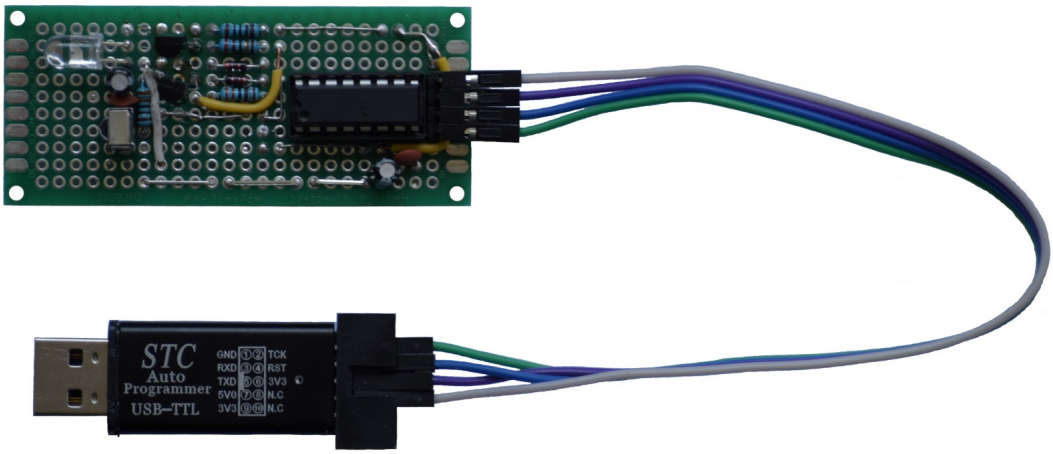
Programming an MCU with STCGAL

Connecting the programmer to the MCU

Connecting the programmer to the MCU works identically with all STC MCU.

If your MCU controls power circuits (e.g. a motor), just make sure they have a separate power supply.

Programmer	MCU
GND	GND
RXD	TXD (P3.1)
TXD	RXD (P3.0)
5V0 (or 3V3)	VCC



STC Auto Programmer connected to a prototype using an STC15W408AS (DIP16)

Uploading firmware

We'll assume your firmware's binary output file is called myproject.ihx. In order to upload it to the MCU, we'll use a command line similar to:

```
stcgal -a -p /dev/ttyUSB0 -t 24000 /path/to/myproject.ihx
```

Here's what each option means:

- **-a** tells STCGAL we're using an automatic programmer, which will power cycle the MCU when $\overline{\text{DTR}}$ is asserted low.
- **-p** indicates the serial port to use (here /dev/ttyUSB0).
- **-t** specifies the system clock frequency of the MCU in kHz (here, 24000 means the MCU will operate at 24 MHz).
  - When you MCU uses an external crystal, you must indicate the crystal's frequency.
  - When your MCU runs on its internal RC oscillator, STCGAL will program the MCU so it runs at the specified frequency.

And here's an example of a successful firmware upload:

```
Cycling power: done
Waiting for MCU: done
Protocol detected: stc15
Target model:
  Name: STC15W408AS
  Magic: F51F
  Code flash: 8.0 KB
  EEPROM flash: 5.0 KB
Target frequency: 34.073 MHz
Target BSL version: 7.2.5T
Target wakeup frequency: 38.050 KHz
Target options:
  reset_pin_enabled=False
  clock_source=internal
  clock_gain=high
  watchdog_por_enabled=False
  watchdog_stop_idle=True
  watchdog_prescale=64
  low_voltage_reset=False
  low_voltage_threshold=3
  eeprom_lvd_inhibit=False
  eeprom_erase_enabled=True
  bsl_pindetect_enabled=False
  por_reset_delay=long
  rstout_por_state=high
  uart2_passthrough=False
  uart2_pin_mode=normal
  cpu_core_voltage=unknown
Loading flash: 7990 bytes (Intel HEX)
Trimming frequency: 34.079 MHz
Switching to 19200 baud: done
Erasing flash: done
Writing flash: 8256 Bytes [00:07, 1134.91 Bytes/s]
Finishing write: done
Setting options: done
Target UID: F51FC38A00F6EA
Disconnected!
```

Debugging

STC MCU don't support JTAG debugging, which is not a problem when dealing with bugs related to timing or interrupts, as a debugger doesn't help much in such situations – a logic analyser and/or an oscilloscope are much more appropriate.

In less critical situations, you'll have to use console output, which is less comfortable than a debugger, yet quite acceptable.

I provide a few sources in the **./reusable-source-code/** directory, including console support, so the work is already done.

I strongly recommend to choose an MCU with at least **2 UART for development**, so you can use the first one for firmware upload, and the second one for debugging.

For instance, you can use an STC8G1K08 in 16- or 20-pin package for development, as those have 2 UART, and use an STC8G1K08A (8-pin package, only one UART) for production.

Conclusion

When I started playing with my first 8051 (an STC8A8K64S4A12), my initial reaction was: "Ouch..." Finding how to properly use SDCC was also a painful process, but I couldn't give up. Over time, it turned out my insistence made me become quite familiar with STC's parts, and beyond that, with their way of thinking and their values.

Besides STC MCU, I also have a bunch of RISC-V, C-Sky, ARM, Xtensa, AVR, PIC, MSP430 and STM8 to play with. However, the complexity of the project permitting, I pick an STC MCU to do it without even thinking...

When I use anything else, I feel like an embedded software developer. When I use an STC MCU, I feel like a craftsman, connected through time and distance to other people with the same inclination. And that's what I enjoy so much!

This document and the associated material have no other ambition than to help others experimenting with STC MCU, and maybe enjoy it as much as I do.

I welcome ideas, comments, and suggestions, so feel free to contact me if you have any (vincent *dot* defert *at* posteo *dot* net).

Appendix: Tales of memory and 8051

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. I'd seriously contemplate resigning if I was required to use it, so let's move on.
- **medium**: allocate variables in PDATA by default, for small applications.
- **large**: allocate variables in XDATA by default, for decent development.
- huge: allocate variables in XDATA by default and use bank switching. As explained earlier, forget it.

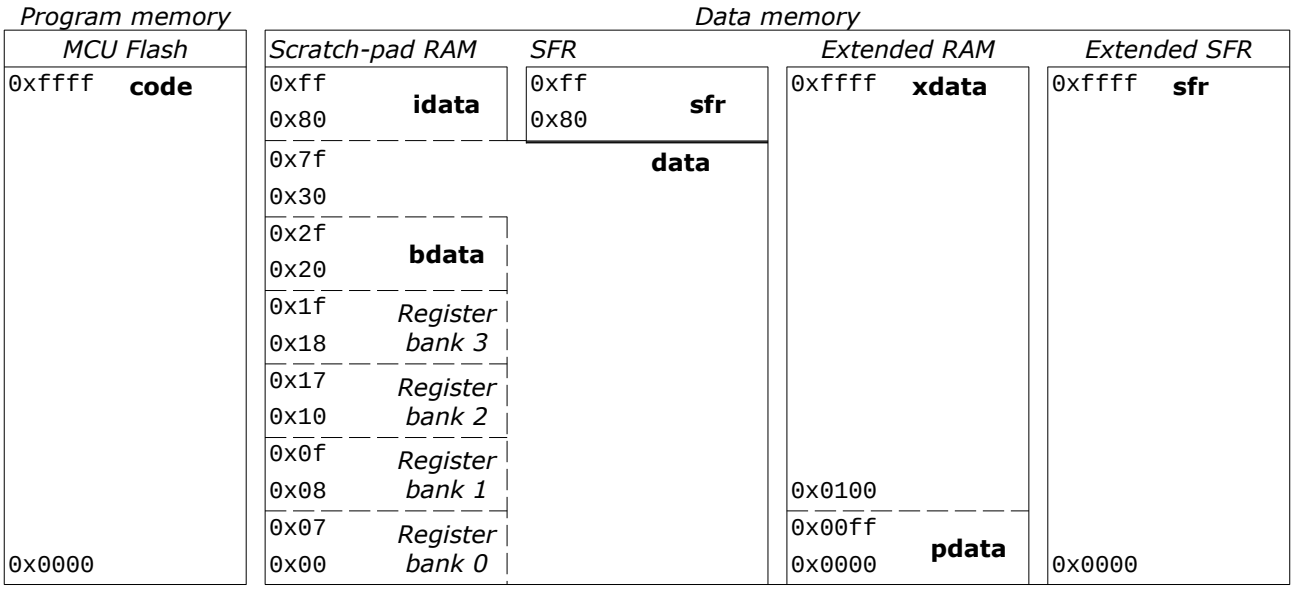
Specifying a memory model as a compiler option will save you the tedium of manually declaring the allocation class of each variable, making your code much more readable and maintainable.

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

DATA access

```
; Store immediate value 0x55 at direct DATA address 0x30.
; Can also be used to access *SFR* in the 0x80-0xff range.
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.
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.
MOV DPTR, #0x0180
MOV A, #0x55
MOVX @DPTR, A
```

PDATA access

```
; Store immediate value 0x55 at PDATA address 0x0084.
MOV R0, #0x84
MOV A, #0x55
MOVX @R0, A
```

CODE access

```
; Read byte at address 0x0240 in program memory
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 $\overline{\text{RD}}$ (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 $\overline{\text{RD}}$.

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 $\overline{\text{WR}}$ (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 $\overline{\text{EA}}$ (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 $\overline{\text{PSEN}}$ (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 $\overline{\text{PSEN}}$.

  By the way, have you noticed that the only difference between program memory read and data memory read is the use of $\overline{\text{PSEN}}$ instead of $\overline{\text{RD}}$? 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.