Using STC MCU with open-source tools

Contents

What, why, and whatnot........2
  Copying........2
  Credits........2
  Introduction........2
  Why use an 8051 MCU in the 21st century?........3
    As a business........3
    As an individual........4
  Why use an STC MCU?........4
  Why use open-source software?........5
Prerequisites........6
  Software........6
  Hardware........7
Compiling firmware with SDCC........8
  First and foremost........8
  Header files........8
  Dual-DPTR support........8
  Extended SFR support........8
  Compiler options........9
  Linker options........9
  Output files........9
  Using .mem and .map files........10
    Meeting RAM constraints........10
    Fitting your code in flash memory........12
  Using .lst files for troubleshooting, illustrated........12
    Symptoms........12
    Diagnostic........12
Programming an MCU with STCGAL........14
  Connecting the programmer to the MCU........14
  Uploading firmware........14
Debugging........16

What, why, and whatnot

Copying

Credits

- AVR is a registered trademark of Microchip Technology Inc.
- ARM is a registered trademark of ARM Limited
- RISC-V is a registered trademark of RISC-V International
- Xtensa is a registered trademark of Cadence Design Systems, Inc.
- MSP430 is a trademark of Texas Instruments Incorporated
- Linux is a registered trademark of Linus Torvalds
- Solaris is a registered trademark of Oracle
- UNIX is a registered trademark of The Open Group
- macOS is a registered trademark of Apple, Inc.
- FreeBSD is a registered trademark of The FreeBSD Foundation
- NetBSD is a registered trademark of The NetBSD Foundation, Inc.
- Arduino (the word) is a registered trademark of Arduino (the company)
- The pink smiley is an adaptation of: https://openclipart.org/detail/203461/cartoon-smiley-with-headphones

Feel free to email me if you notice any error or omission.

Introduction

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 an STC MCU, I feel like a craftsman, a human being involved in a human activity, and not just some disposable wheel of a corporate machine – what they call a human **resource**.

This document has 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).

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 she needs, impacting their own performance.

With a simple product using a simple technology, your new hire will have gained experience in your field, 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.

The same applies to electrical engineers, of course: with a simpler technology, hardware design issues are both less likely and less serious.

The following quote illustrates the remarkable efficiency of this "small steps strategy":

> Thomson's rule for first-time telescope makers: "**It is faster to make a four-inch mirror then a six-inch mirror than to make a six-inch mirror.**"
> *in Programming Pearls, Communications of the ACM, September 1985*

In other words, using a technology matching the complexity of the problem to solve brings you flexibility and visibility in terms of planning, product development and team management.

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 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 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 or AVR) 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 their additional complexity. 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 you won't need to painfully unlearn tools such as the Arduino framework when you'll want to go beyond what they offer.

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, potentially including:

- 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, 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.

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

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.

SDCC can be downloaded from its SourceForge page, http://sdcc.sourceforge.net/, 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. However, STCGAL's developer seems to have lost interest in his product early 2021, so I recommend using the following version, patched to support recent STC MCU and slightly improved:

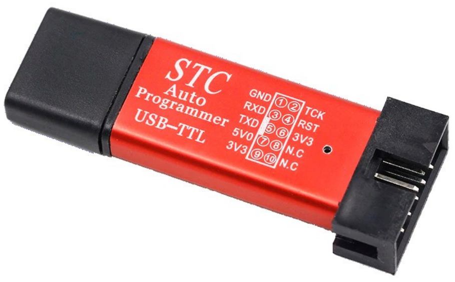
https://github.com/area-8051/stcgal-patched

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

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, described in the "**Using_an_STC_Auto_Programmer_with_STCGAL.odt**" document, available in the same repository as the present document.

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.odt**" document, available in the same repository as the present document, 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 the STC12, STC15 and STC8 MCU families. While I was at it, I also wrote a Hardware Abstraction Layer (HAL) for the peripherals I was frequently using.

All this is available as open-source software at **https://github.com/area-8051/uni-STC**

Just clone this repository and follow the instructions in its README file.

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 Makefile examples provided under the **demos/** directory of the **uni-STC** repository.

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 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.

Macros are provided, so you can just call **ENABLE_EXTENDED_SFR()** before using extended SFR and **DISABLE_EXTENDED_SFR()** 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-size** to optimise for size. Because STC MCU don't support JTAG debugging, producing a debug executable is not relevant.
- And of course, -c and -o.

I also pass the following macro definition to the compiler:

> **-DMCU_FREQ=***nnnnnnnnn*UL specifies the system clock frequency in Hertz as an unsigned long constant. This is very useful to program timers or PWM generators without hard-coded numbers, or to implement delay loops.

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

Meeting RAM constraints

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|c|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 differ a little 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|c|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 size**, and access to DATA needs less instructions (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 also use RAM in the default segment of the selected memory model. SDCC does this for arguments that don't fit in the registers usage defined by its ABI (Application Binary Interface, see SDCC's manual for details).

This means that if you have a lot of functions in your application, you can also reduce their RAM usage:

- By using global variables, at the expense of readability and maintainability.
- By using **inline** functions, which considerably slow down compilation and quite often increase code size.
- By using **__reentrant** functions, i.e. functions using the stack for their arguments. However, the stack on an MCS-51 is very small (8-bit stack

pointer), so you can't do this if your arguments take more than 4-6 bytes (SDCC will give you an error when you reach the limit).

As you can see, each alternative comes with its own trade-offs.

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

Fitting your code in flash memory

Besides RAM, the linker may also report insufficient flash memory for your firmware, despite already optimising for size.

If you're using the **large** memory model and the **.mem** file says you would have enough with 256 bytes XDATA RAM, using the **medium** memory model instead will do wonders!

If it's not enough and you have a few **inline** functions consisting of more than 1 or 2 variable assignments, you may try to replace them with **__reentrant** functions.

If you're using a HAL, you'll also want to remove any unneeded functionality from it. For instance, remove support for GPIO ports 1 and 5 if you only use port 3, or support for UART2/3/4 if you only use UART1. Doing so will preserve the benefits of the HAL while reducing the memory footprint.

You may want to list the generated **.rel** files by size to see where your efforts are the most likely to bring significant gains.

Now, these strategies allow to deal with a firmware size exceeding the available flash memory by, say 10%. With 100% in excess, of course, you'll have to select another part.

Using .lst files for troubleshooting, illustrated

Symptoms

The CS line of an LCD display was remaining high in spite of gpioWrite() being called to clear it before transmission. The C/D line was affected by the same problem, but stayed low instead of going high.

Diagnostic

Clearing the CS line with P1_6 = 0 was working, though, so it wasn't a physical problem. Moreover, gpioWrite() called elsewhere in the code was working as usual, so the problem was obviously external to gpioWrite().

Strangely, adding printf() statements in the code did make it work, evoking a buffer overflow. Removing printf() and adding an 's' local variable of type char[32] in main() produced the same effect (code working). Reducing it to a char[3] caused C/D to work, but CS to fail, confirming the hypothesis of a buffer overflow.

Because adding the char[] to main() had an impact, there were good chances the culprit was in the same source file. This is where the **.lst** file came handy.

Comparing a working main.lst file with a non-working one showed the only differences were the lines highlighted in bold red below:

```
                    00FA3B    951 _DMA_UR4R_AMT      =       0xfa3b
                    00FA3C    952 _DMA_UR4R_DONE     =       0xfa3c
                    00FA3D    953 _DMA_UR4R_TXAH     =       0xfa3d
                    00FA3E    954 _DMA_UR4R_TXAL     =       0xfa3e
  000000                      955 _lcdDeviceBuffer:
  000000                      956     .ds 480
  0001E0                      957 _main_s_65536_83:
  0001E0                      958     .ds 3
                              959 ;--------------------------------------------------------
                              960 ; absolute external ram data
                              961 ;--------------------------------------------------------
                              962     .area XABS    (ABS,XDATA)
```

The *_main_s_65536_83* symbol represents local variable *s* in function *main*, and "*.ds 3*" reserves 3 uninitialised bytes, corresponding to a char[3].

Immediately above is an uninitialised array of 480 bytes, lcdDeviceBuffer, also declared in main.c, which represents each pixel on the LCD display with one bit in the buffer.

*[Note: this specific memory layout is due to the limited 8051 stack space, causing SDCC to use the stack **only** for __reentrant functions. Everywhere else, local variables and function parameters are allocated in "normal" RAM. Such a bug wouldn't occur as-is on an MCU architecture with a decent stack.]*

Ok, but why 480? The LCD display has 48 lines of 84 pixels, and 84 isn't divisible by 8, so the buffer had room for only 80 pixels, not 84. The correct buffer size should be 48 x (84+7) / 8 = 528 bytes (+7 is for rounding).

After fixing the buffer size calculation, the program works without any char[] or printf(), and the **.lst** file shows the correct size:

```
                    00FA3B    951 _DMA_UR4R_AMT      =       0xfa3b
                    00FA3C    952 _DMA_UR4R_DONE     =       0xfa3c
                    00FA3D    953 _DMA_UR4R_TXAH     =       0xfa3d
                    00FA3E    954 _DMA_UR4R_TXAL     =       0xfa3e
  000000                      955 _lcdDeviceBuffer:
  000000                      956     .ds 528
                              957 ;--------------------------------------------------------
                              958 ; absolute external ram data
                              959 ;--------------------------------------------------------
                              960     .area XABS    (ABS,XDATA)
```

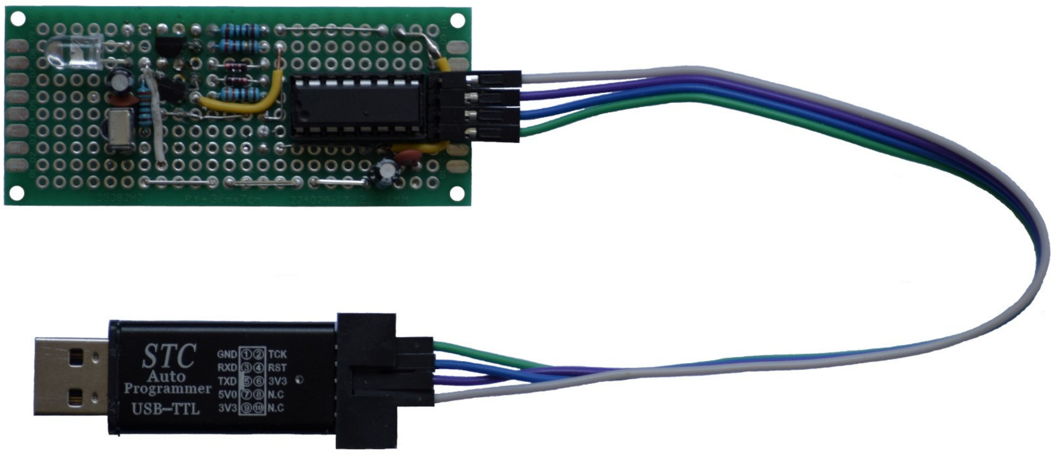
Programming an MCU with STCGAL

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.

The HAL in the **uni-STC** repository also provides serial console support, so the work is already done.

The files you need in your project are:

```
	$(HAL_DIR)/fifo-buffer.c \
	$(HAL_DIR)/timer-hal.c \
	$(HAL_DIR)/uart-hal.c \
	$(HAL_DIR)/serial-console.c \
```

Your main.c should include:

```
#include <uart-hal.h>
#include <serial-console.h>
```

and you should call, at the beginning of main():

```
	serialConsoleInitialise(
		CONSOLE_UART, // TODO: replace with
		CONSOLE_SPEED, // your own values
		CONSOLE_PIN_CONFIG // or define these symbols in project-defs.h
	);
```

Also, don't forget to enable interrupts (EA = 1;) before using printf()!

Note

Some terminal emulators (e.g. Minicom) take $\overline{\text{DTR}}$ low until the device responds which, when using an STC Auto Programmer, powers the MCU off.

If this is a problem, you'll have to use an MCU with 2 UART (at least for development) and connect the second one to the serial console.