The technology behind avr_sim

A brief description of the avr_asm software

by

Gerhard Schmidt, Kastanienallee 20, D-64289 Darmstadt/Germany

Draft version as of April 2020

1 Why Lazarus and Pascal?

avr_sim is completely written in Lazarus-Pascal. The following reasons are behind this:

1. The Lazarus compiler is available for several operating systems. The same source code can simply be compiled to run under any of those other operating systems. Only minor changes have to be made, mainly caused by different fonts available under Win and Lin. I do the main work under Windows, copy the Pascal code files and the forms to Linux, re-design the forms to fit the different font settings under Linux, compile it, test it and release the executables for Windows and Linux. I'd like to say thank you, Lazarus developers, for that reliable piece of software.
2. The standard components for composing the user interface (buttons, edit fields, comboboxes, memos, listboxes, string grids, images, the editor SynEdit), are very simple to apply and to fit those to the needs here. I designed and wrote the editor highlighter for AVR assembler code simply from a very simple example, by adding my very special additional code (recognition of the AVR mnemonics, of directives and functions, of def.inc and user-defined constants) step-by-step. So the design work can concentrate mainly on the optimization of the user interface. And they really look-alike and function similarly under every operating system in Lazarus, without having to care for the underlying very different libraries.
3. The integrated assembler, gavrasm, has a long history: I started in 2003 to develop this as a command line application, because ATMEL did not react on serious error reports on its software. The source code for gavrasm was written in Delphi-PASCAL, I changed only later to Free Pascal (FPC) just because FPC was also available for Linux, and Delphi remained windows-oriented. As the source code for gavrasm was written in PASCAL, it was simple to adapt it to fit to the simulator. Adaptation was very simple: most of the code did not need any additional work, just replaced the WRITELN and refitted the old-style Function result settings. Mainly only the command line output operations had to be changed. Since then I do the bug-fixing work under avr_sim and simply copy the changes to the gavrasm source code. As gavrasm is now 16 years old and has riped over the 45 released versions most of the invested work paid out in avr_sim.
4. Finally, the main reason was that I program continuously in Pascal since 1985 (when Borland released Turbo-Pascal 1.0 for DOS), but programming was never been my paid profession. I learned other languages, too, but Pascal (and especially FPC and Lazarus) are, for me, simpler to program, to comfortably debug, and has all one needs for a complex software project.

2 How files and hardware work

2.1 Handling files in avr_sim

The main unit of avr_sim, **avr_sim_u1**, handles files in a record array of the type **TIncl**. An assembler project has four types of files: source code files (*.asm), include files with source code (*.inc), the assembler listing (*.lst) and a gavrasm-specific error file that lists all assembler error messages

(*.err). The array **aFile** spans from -2 (for the error message file) over -1 (for the listing) over 0 (the main source code file) to maximal 10 (for all include files). The number of files of a project is handled in the variable **nFiles**.

The **TIncl** record consists of the following components:

1. Strings: **sName** is the short name of the file, without the path, **sFile** is the complete path plus filename plus filetype extension, the string **sBreak** has a length of the file's number of lines, consists of blanks and of the character "B" for those lines that have an activated breakpoint, the same structure has the string **sLineTypes**, but it has additional characters (such as "C" for pure comments, "L" for labels, "I" for instructions, etc.).
2. Points: **pCaretPos** holds the current editor position (column, line) when the file is loaded into the editor window.
3. Bookmark positions: The line-numbers in which any bookmarks (between 0 and 9) are located in the file are stored in an array **aBookMark** of integers. Those bookmarks are restored whenever the file is loaded into the editor.
4. Boolean value **fAnyBookMarks** is true if the file has at least one bookmark.
5. An integer **iTopLine** holds the line-number that is first displayed in the editor window, so switching between edited files always show the same layout for each specific file.
6. The integer iChanged holds the file's age stamp, to be able to detect if an external editor has changed the file.

All this information is stored in project files (*.pro), is saved and restored whenever such a project file is opened.

The tab above the editor window displays the file names in the following row:

1. The main assembler source code file,
2. all included files (max. 10),
3. the assembler listing (if assembled),
4. the error listing (if assembled and errors resulted).

Note that the content of the file in the editor window is always loaded from the stored file, whenever the file is selected in the tab. If changes to its content are made, the new content has to be stored whenever you change to another tab position, otherwise your changes would be lost.

The file that is currently loaded into the editor is enumerated in the variable **nEdit**. It's value between -2 and +10 points to the respective TIncl array. Whenever the user clicks on the file tabs, or when the assembler has finished a different file is pointed to, saving the old file content, reloading the next file's content, and changing **nEdit** accordingly.

2.2 The AVR's hardware

When the user

- creates a new assembler file using "Project" and "New" in the menu, or

- opens an existing .asm file using “Project” and “New from asm”, or
- opens a project file .pro using “Project” and “Open” from the menu, or
- opens an already processed project file from “Project” and “Open previous” by selecting one of the stored projects,

then avr_sim either opens the related project file (and reads the information from there), and/or opens the related .asm file and reads it. avr_sim then searches for .include directives in the .asm file. If there are such directives, the related file names are analyzed. If it is a def.inc, that the directive points to, avr_sim extracts the type of device from that. If the include directive’s file is not a known def.inc, then it adds the included file to its file list.

All def.inc files that are known by avr_sim are in the unit gavrdev.pas. For each known device (currently 302) the following information is available in a record of the type **TDevice**:

- sn: the device’s name as a string,
- sdi: the device’s include file name as a string,
- nf, ns, ne: the device’s size of flash, SRAM and EEPROM memory in bytes,
- nr: the device’s number of available registers (usually 32, but 16 for AVR8L types),
- nss: the byte address that the SRAM starts with (normally 0x0060, but in larger devices the SRAM starts beyond that),
- iSet: a 32-bit word that encodes the device’s instruction set, see the constants **hasDoc** to **hasMathExt** defined below for the bit meanings,
- fSreg: is true if the SREG’s port address is 0x003F, otherwise the port address is encoded in one of the following manners,
- sBits, soSym, sStd and sSym: these are strings that define all symbols of the device (see the constants sBitNames and aStdSym for encoding details), all symbols are stored in a compressed form and are decompressed by respective routines: the functions **GetDeviceSymbolFirst** and **GetDeviceSymbolNext** provide functions that return symbol name and value pairs, one by one, to transfer those to the symbol storage space provided in the unit **gavrsymb.pas**.

All those information in this unit are derived from the currently 302 def.inc files that are part of the latest versions of the Studio and get updated by me twice per year. The def.inc files in the Studio are 28.5 MB in size. Due to the compressed encoding the derived PASCAL unit has only 826 kB source code size and compiles to 3.5 MB (one eights or 12% of the original def.inc files). That is why gavrasm and avr_sim are so much faster than the Studio elephant.

If the type of device has been found this way, the available hardware in this device is identified. Two different information sources are used for that:

1. the symbols from the def.inc file, as available from the **gavrsymb** unit described above,
2. the unit **avr_sim_deviceu.pas**: this unit provides for 169 different device type groups (with 457 avr devices) the pin names.

The latter unit is designed as follows. Each device type (such as ATtiny4/5/9/10) has a string in the **aDevice** array constant, that consists of the following:

1. the device group name (such as ATtiny4/5/9/10), followed by a blank, if subversions such as PB or alike differ from the base versions without PB, those have an own entry,
2. the devices group's package forms, separated by blanks if more than one identical forms, if there are different pinnings in different package forms or if the same device comes in different packages, each form gets a separate entry,
3. then all pins follow, starting with an asterisk, if the pin has more than one possible function the functions are separated by blanks.

With those two information sources the available hardware in each type can be derived. If a symbol named "ADC3" is defined in the def.inc or if a pin function "ADC3" exists in the pinning of the device, then the AVR type has at least 4 ADC channels. Similarly the internal timers can be identified by searching for the symbols TCCRn (in older devices), if TCCRnL and TCCRnH are defined then its a 16-bit TC, otherwise 8-bit. TCCRnA and TCCRnB, with n=0 to maxAvrTimer also determine the number of active TCs. If those symbols exist, the port addresses can be read and assigned with their names. Similarly the OCRnA, OCRnB and OCRnC pins can be identified from the unit **avr_sim_deviceu**. That makes it simple to identify all the available hardware and to assign their port addresses to those.

The following hardware information is additionally read from the unit **deviceclock**:

- the AVR's default clock in Hz,
- if the device has a CLKDIV8 fuse,
- if the device can toggle port pins by writing ones to the device's pin port,
- the frequency of the device's watchdog timer oscillator in Hz, and
- the base prescaler value of the device's watchdog, that can be doubled with the watchdog prescaler bits.

Hardware identification is performed in the procedure GetInternalHardware. The following hardware components are searched for:

1. the clock prescaler CLKPR,
2. max six external INTn,
3. Sleep modes,
4. I/O ports (PORTn, DDRn, PINn),
5. I/O port pins for max, 12 ports from A to L,
6. INTn pin locations,
7. max 40 PCINTn pin locations,
8. all timer relevant properties such as
    - the timer's OC pin locations and their respective port locations,

    - the timer’s input pins and their location,
    - all interrupt ports of timers,
    - control ports,
9. the watchdog timer, and
10. all max 16 ADC ports and pins.

All ports get pairs of names and addresses, all bit locations in those ports are identified to be able to simulate those bit combinations correctly.

With these information the hardware of the device used is completely known and accessible. The respective units (e.g. the TC unit) care for the timer relationships (e.g. configuration of the timer, timer ticks, hardware action on comparer events, timer interrupts, etc.).

3 Code execution

3.1 Hex code generation

The hex code to be executed is generated by the integrated gavrasm command line assembler: it assembles the source code and produces a hex code file in Intel hex format and with the extension .hex.

If assembling was successful the generated hex file is read and its content is written word-wise to the flash array **aFlash**, located in the **avr_simul_u1** unit. The code is read from that array.

3.2 Code decoding and execution

Starting from the flash address 0000, the hex code there is read from the array by the procedure **ExecuteStep**. This decodes the instruction words as follows:

1. If the code is $FFFF, an uninitiated part of the flash memory is accessed and an error message is processed.
2. The upper six bits of the code are then isolated and serve as a pre-selector for the instructions:
    - A zero stands for the multiple instructions **NOP** or for **MOVW/MULS/MULSU/FMUL/FMULSU Rd,Rr**.
    - A one stands for the instruction **CPC Rd,Rr**, a two stands for **SBC**, a three for **LSL** or **ADD**, and so on until 11.
    - The following combinations use two bits of the upper six and address 8-bit constant instrctions:
        - 12..15 stand for **CPI Rd,K**,
        - 16..19 are for **SBCI Rd,K**,
        - 20..23 stand for **SUBI Rd,K**, etc. until
        - 28..31.

- The combinations 32..35 and 40..43 address various **LD/ST** and **LDD/STD** instructions.
- The combinations in between, 36..39, address a very large variety of instructions, which have to sub-decoded.
- 44..59 encode I/O, relative jumps/calls and load instructions.
- 60 and 61 encode relative jump instructions,
- 62 and 63 stand for bit manipulations in registers.

The codes that address multiple instructions (e.g. 0) use special code to recognize the different instructions.

The flags that instructions set or clear are held are performed by the procedure **ChangeSreg**, which expects the flag's abbreviation ("C", "Z", etc.) and an "S" (for set) or a "C" (for clear) as second parameter.

Instructions that manipulate register content read and write to the 32 registers in the **aReg** byte array.

Instructions manipulating ports, such as **OUT** or **ST/STS** instructions pointing to ports are performed by the procedure **ChangePort**. That ensures that changes to the hardware are recognized and the respective hardware units are aware of any changes made.

As most of the AVR instructions are single clock cycle ones, the default of advancing the clock cycle counter is one. One time, or in case of more cycles several times, the routine **IncClock** is called and advances timers accordingly, according to the number of consumed clock cycles. The distance from the current to the next executed instruction, normally +1 but different in relative jump instructions, is held in the variable **nPcAdd**. That ensures that the jump leads to the correct flash location address.

3.3 Interrupt execution

Following each instruction execution, the procedure **ProcessIntReq** is called. This routine steps through all entries of the interrupt list of the device (top down to implement the interrupt priority rule) and checks if one of the interrupt conditions is fulfilled. If that is the case this interrupt is processed by calling **ProcessInt** and further list checks are skipped. Processing the interrupt disables the I flag, pushes the current address to the stack in SRAM, marks the int executed (in the procedure **MarkIntExec**) and points PC to the interrupt's vector.

If later on a **RETI** instruction is executed, the PC is set to the value on the stack in SRAM, the stack pointer is increased and the interrupt is unmarked (in the procedure **UnmarkLastInt**).

Both delays when starting and ending an interrupt are correctly timed by use of the **IncClock** procedure.

4 Single hardware components

The following describes some very special hardware components and their simulation.

4.1 Ports

4.2 Timer/Counter

4.3 Watchdog timer

I added the watchdog timer in version 2.0. As the watchdog timers are very different I added two additional entries in the **deviceclock** unit:

1. The oscillator frequency of the watchdog timer, **nWdOsc**.
2. The basic prescaler of the watchdog **nWdBase**, to be multiplied by the binary exponents from the [WDP3:]WDP2:WDP1:WDP0 bits.

These values vary for many device types, and even vary between different types of the same series (e.g. the PA or PB versions differ from the base and A versions in some cases). So the **deviceclock** unit was updated to reflect the current state of knowledge (which ended up in adding a few missing types to that list, too).

As the watchdog timer runs, if enabled, asynchronously to the device's clock source, I decided to program the timing separately. That means I add the time per instruction cycle (depending from the device's clock frequency) each time when an instruction has been executed. Whenever the clock frequency changes (on project load, on CLKPR operations, etc.), the new delta time **dWd** has to be updated. Through the many thousand single additions, the accumulation of time in the variable **eWdt** is prone to rounding errors in very large prescaler cases, but that should be a less important source of errors.

If the watchdog counter overruns, either a RESET happens or an interrupt is generated. The WDT- or WATCHDOG-interrupt fitted well into the existing interrupt scheme that was already working fine and reliable in avr_sim.

4.4 AD converter

4.5 EEPROM

4.6 SRAM

5 Lazarus source code and compiling

The Lazarus source code and executables produced have the following properties:

avr_sim properties			Size in MB	
Version	**Files**	**Lines**	**Exe debug**	**Exe packed**
2.0	43	36.937	44.501	10.756
1.9	42	36.469	44.303	10.710
1.8	47	39.373	44.300	10.711
1.7	44	36.066	44.300	10.711
1.6	42	34.757	43.149	10.411
1.5	42	34.331	40.893	9.901
1.4	40	34.145	40.878	9.898
…	…	…	…	…
1.0	42	32.157	39.819	9.655
…	…	…	…	…
0.5	37	26.032	38.368	9.289
0.2a	21	16.365	36.662	9.331

As can be seen, the number of source code files is not increasing since version 1.0 had been released. So does not the number of source code lines in all those source files. The same applies for the size of the generated executables and the zipped debug versions.

That means that optimizations to the code, made under each version, are highly effective and that avr_sim has reached a steady-state, where added additional features are compensated by code optimization in other areas.