The technology behind avr_sim

A brief description of the avr_asm software

by

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

Version 2.8 as of August 2022

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 to the needs here. E.g.: I designed and wrote the editor highlighter for AVR assembler code simply from copying a very simple example, by adding my very special additional code (recognition of the AVR mnemonics, of directives and functions, of def.inc's and user-defined constants) step-by-step. So the design work concentrates 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, graphic interfaces, etc..

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 solely windows-oriented (the short swing to Kylix was too short for my taste). 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 WRITELNs 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 17 years old and has riped over the 46++ released versions most of the invested work paid out in avr_sim.

4. Finally, the very main reason was that I program continuously in Pascal since 1985 (when Borland released Turbo-Pascal 1.0 for DOS), but programming has 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 that it needs for a really complex software project like avr_sim.

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, .device and .include def.inc directives in the .asm file. If there are simple .include directives, the related file names are analyzed and the file is read (to detect nested includes). If a .device or a .include def.inc directive is detected, avr_sim extracts the type of device from that. If the .include def.inc directive's avr-type is not an already known def.inc, then it adds the included file to its file list and reads it.

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 only 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 information for 169 different device type groups (with 457 avr devices), including the multiple pin names of those devices.

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 form exists, 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 different 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 four ADC channels. Similarly the internal timers can be identified by searching for the symbols TCCRn (in older devices with only one TC: TCCR). If the symbols 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 are assigned to 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,
- the device's three-byte-wide signature.

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. the available Sleep modes,
4. I/O ports (PORTn, DDRn, PINn) including port pins for max. 12 ports from A to L,
5. the INTn pin locations,
6. max. 40 PCINTn pin locations in five ports,
7. all timer relevant properties such as
    - the timer's OC pin locations and their respective port locations,
    - the timer's input pins T0 and T1 and their location,
    - all interrupt abilities of timers,
    - the control port registers of the timers,
8. the watchdog timer, and
9. 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.

This collected information is stored in the respective unit. E. g. the timer/counter unit TCU holds all information concerning all timers. These are the port register addresses, generally starting with an "n" in the variable's name. The bits such as CS00 or COM1A are stored with their bit number (variable names starting with a small b) as well as with a binary mask (variable names starting with a small m). The mask byte is used to determine if a bit in a port is cleared or set (with binary AND) or to clear or set this bit in the port register (with binary ANDs or ORs).

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.).

From version 2.6 on another unit **avr_hardw** allows to use and store up to ten different AVR types dynamically. Internally stored are their def.inc-number and their package number. The index starts with 0, where the currently used AVR type is written (a copy of the variables in the indexed type). If the project file is written, all used hardware is written to the project file, starting with the last used AVR type first. The information written to the project file are the AVR-type's names and the package names (if already selected by the user). Not the numbers are stored, because this would be incompatible with changed def.inc- and dev-unit rows in later versions of gavrasm and avr_sim.

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**. The hex 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 instructions, where the upper two bits of the constant modify the lower two bits of the six bits:
        - 12..15 stand for **CPI Rd,K**,
        - 16..19 are for **SBCI Rd,K**,
        - 20..23 stand for **SUBI Rd,K**,
        - 24..27 are for **ANDI Rd,k**, and
        - 28..31 stand for **ORI Rd,K**.
    - 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 get aware of any changes made.

As most of the AVR instructions are single clock cycle, 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 as currently executed (in the procedure **MarkIntExec**) and points the 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.

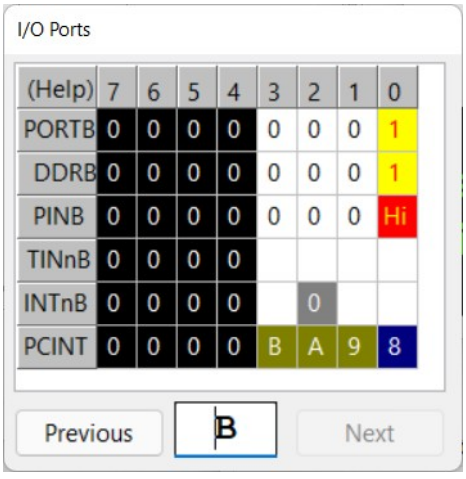
4 Single hardware components

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

Please note that I removed the standard Close, Maximize and Minimize buttons from the hardware display windows. This was intended only for the close button (to urge the user to use the selection boxes that are provided on the simulation window for that). But disabling the close button in Lazarus unintentionally also removes the whole set. I did not find a way to only remove the close button. As the other two buttons do not really provide a user-friendly re-sizing (the scope display, where this would make sense, can be re-sized by pulling on the window's edges), so I can live with that limitation.

4.1 I/O-ports

All I/O port registers are handled in the unit **avr_ports**.



This unit has been changed in many versions to improve its functioning, because the display of the I/O ports is rather sophisticated. Two cases have to be handled:

1. Program execution changes I/O port registers, so those changes have to be reflected in the display.
2. A user click onto I/O register bits changes I/O registers, which has to be reflected in the display.

The main changes to I/O registers all might affect the PIN bits. If the DDRn bit in a port is set, the PINn bit has to follow the

respective PORTn bit. If the DDRn bit in a port is clear and if the PORTn bit is set, the PINn bit is in an active Pull-Up state and reads as one. The user has then to be able to click on the PIN bit to make the input pin low. This can also trigger INTn and PCINTn interrupts, if so selected.

The ports are displayed using a TStringGrid component. This allows to comfortably change text and background colors of the displayed strings by applying the TStringGrid.PrepareCanvas and .Draw-Cell methods. That was exactly what was needed for this kind of display.

In the first version of avr_sim the user action of clicking into a cell of TStringGrid was programmed as a reaction to the TStringGrid.SelectCell event. This turned out to be instable, because the component itself sometimes initiates this event (e.g. when drawing the component for the first time), so I had to add diverse flags to control that and to get this stable and correct. In version 2.6 of avr_sim I changed that to the TStringGrid.MouseUp event, after I learned how to translate the mouse position to the column and row positions of the StringGrid. Much better and much more stable than the SelectCell method.

A lot of work changed this unit in version 2.6: all user actions to the string-grid component did not work correct. Note that the PCINT-programming has changed considerably to get this corrected. Minor changes were also made to the TC unit to get user clicks on TINn bits correct.

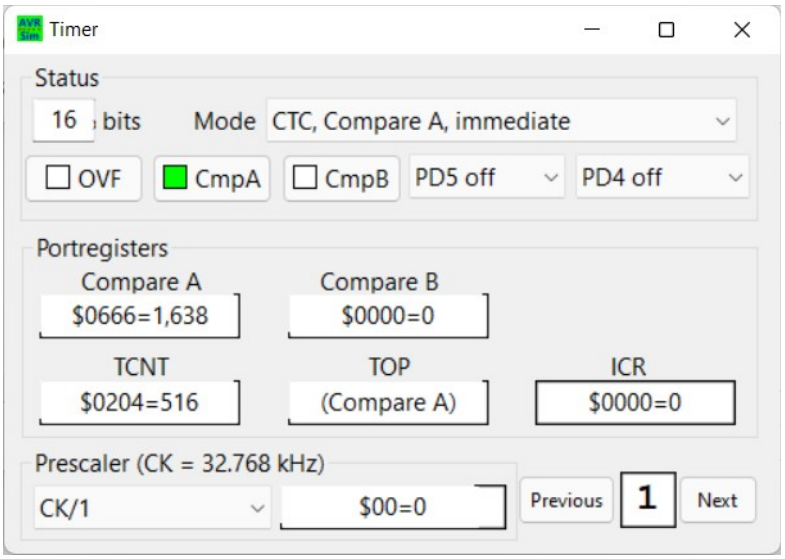
This I/O port unit provides the following published procedures:

- PortReset: This sets all port variables (namely their addresses) to their default value (addresses: $FFFF) and is called on every start-up (on form initiation, on start of the project file reading). It also calls PortRestart to clear the complete port register array as well as the port name associations.
- PortRestart: This clears all port registers to their default and is called on every restart of the simulation.
- UnmarkAll: This sets all previous port values to the current values.
- UnmarkLast: The last change to a port’s value is copied to the previous port value (called on every manual step to avoid lengthy copy operations).
- ToggleIOPort: If a write access occurs on a PIN port register (by program execution only, not by user action), the bits that are one toggle the bits in the respective PORT register, if the device provides this toggle feature. This procedure is called by the ChangePort procedure in the **avr_sim_u1** unit.
- ChangeIOPort: This procedure handles all write operations to the PORTn, DDRn and PINn port registers. The major function of the procedure is to reflect changes to the PORTn and DDRn port registers, which change the PINn register accordingly.
- CountPortsPinsInts: This counts the number of available ports, pins and ints/pcints. It is called by the main unit when preparing the project window.
- DisplayPort: The currently selected port is displayed in this procedure.
- DisplayActivePort: If the port is not fixed, this procedure switches the port to be displayed and displays this port.

- MarkInt, UnMarkInt: This procedure marks/unmarks the currently executed INTn pin and displays the background color of INTn.
- MarkPcInt, UnmarkPcInt: This procedure marks/unmarks the currently executed PCINTn pin.
- SetIsc: This procedure sets the ISC bits of the INTn and displays their background color.
- SetIntCR: Changes to INTn's interrupt control register are processed in this procedure.
- SetPcMsk: Changes to the PCINT's interrupt mask register are handled here.
- SetPciCR: Changes to the PCINT's interrupt control register are processed here.
- StartInt: An INTn has been initiated and the clock cycle counter is started.
- StartPcInt: A PCINTn is started here by setting the clock cycle counter.
- TickInt, TickPcInt: The clock cycle counters of INTs and PCINTs are down-counted and it is checked whether the INT or PCINT condition is still valid.

4.2 Timers/Counters

TBD



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

TBD

4.5 EEPROM

When starting simulation or when restarting simulation, the content of the .eep file is read to the respective byte storage space. When EEPROM content is read it is taken from this byte storage. If new EEPROM content is written, the respective procedure with setting the address in EEARH:EEARL and the data register in port EEDR and finally setting the EEPROM-write-enable bit is followed. The timing of the EEPROM write process uses a clock-specific counter, which enables to display the write process in a progress bar display.

The interrupt on completion, when enabled, is displayed like any other of the interrupts (green if enabled, yellow if requested, red if executed).

In single steps, the complete EEPROM content is compared with the content one step before. If there are any changes in content, the display window is updated. In run/go mode, this comparison and update is only made when the run/go mode is finished.

4.6 SRAM

The complete SRAM content is set to 0xFF when simulation starts or when a simulation restart is selected. Content changes are recognized by comparing the complete data before and after each step, in run/go mode only after the simulation is stopped.

5 Memory organization in unit avr_alert

The memories of the applied AVR were, in the versions up to 2.2, organized as simple byte arrarys in the unit **avr_mem**. From version 2.3 on those were moved into the newly added unit **avr_alert**. The reason was the following: the added alert feature has to check any accesses to registers, portregisters and SRAM locations if the user has activated the alert feature for this location. In order to not having to change all code lines that access those locations I

- defined them as classes (TReg, TPortreg, TSramreg),
- moved those byte fields as PUBLIC arrays into those classes (in order to allow unchecked accesses to those locations like before),

- wrote code for their checked write- (SetReg, SetPortreg, SetSramreg) and read- (GetReg, GetPortreg, GetSramreg) access during simulation that additionally checks if an alert has been set active on this location.

That ensures that all accesses to those locations can be checked without major changes to the pre-existing code. In this way,

- write- and read-accesses to those locations are checked separately to allow the user to be alerted when those accesses happen (in the modes: read, write and read/write),
- if the location (register, register-pair, portregister, portregister-pair, Sram and Sram-pair) has reached a certain value, or
- simply changes its value.

The events can be

- counted,
- counted and a break can be performed (when simulation is running),
- counted and a sound is played via the PC's/laptop's speaker,
- counted and a break is made and a sound is played.

The reason why all components, variables, procedures and functions have a "1" at the end is that I'd like to add the same structure again to check for a second condition in later versions, if this feature proves as useful and well-functioning.

6 Scope display with units scopectrl and scopeunit

The scope display is located in two units: **scopectrl** and **scopeunit**. The control unit provides the user interface to configure the display, the scopeunit collects and displays the configured entries.

The first four channels are configurable: those can follow digital signals, either from a single port bit or from an OC pin of one of the timers, or they allow following analog signals from an external R/2R network attached to one of the ports of the device.

With version 2.3 it is also possible to follow the analog values that are produced with an internal Digital-to-Analog converter in devices that have the hardware for that, e. g. in an ATtiny212. This has been attached as **Beam 5**. This feature is rather seldom and required some very special changes to the software.

The scope display uses arrays of a limited size to store change events. This limit is by default set to 1,000. When increasing this size in the source code, the respective limit check has also to be adjusted. As the scope picture is re-painted whenever a change to the configured sources happens, be aware that increasing the size requires some additional time.

7 Tracing in unit avr_trace

The unit **avr_trace** was introduced in version 2.5. It allows to control the collection of data during the simulation process and to save this data regularly (after a certain number of data lines have been collected) and at a defined end. Prepared lines in the source code enable the tracing process (see the

handbook), a control window opens when simulation is started that allows to configure the collection process.

The defined terms that control the data to be traced allow to switch registers, port registers and SRAM locations on. Fixed addresses as well as symbols can be used. Up to eight 8-bit components can be combined to yield 63-bit integer values. A maximum of 33 numbers can be combined on one line. The number format in the output file can be decimal or hexadecimal.

The results of the tracing process are written as .csv files, separated by ASCII-Tab characters.

8 Debugging complicated .if sections

Sometimes you have complicated constructions of nested .if/.ifdef/.ifndef/.else/.elif/.endif. These can get very complicated to follow and debug. Up to version 2.8 the .if processing only handled two levels correct and failed when using more complex .if conditions. So the complete .if processing was improved in version 2.8.

The fIfAsm flag in the unit “gavrif” is used by the unit “gavrasm” to determine if the source code is currently to be assembled or not. This includes not only the assembling of instructions but also any directives. If assembling is disabled by fIfAsm = False then only the directives

.if, .ifdef, .ifndef, .else, .elif and .endif

are to be executed.

All these directives are called by gavrasm in procedures named “Process[name]” (ProcessIf, ProcessIfdef” etc.), where all the checks are made (e. g. if the condition in .if is true or false). These procedures then call functions within the unit “gavrif”, that provide results and do all handling of the conditions on the stack.

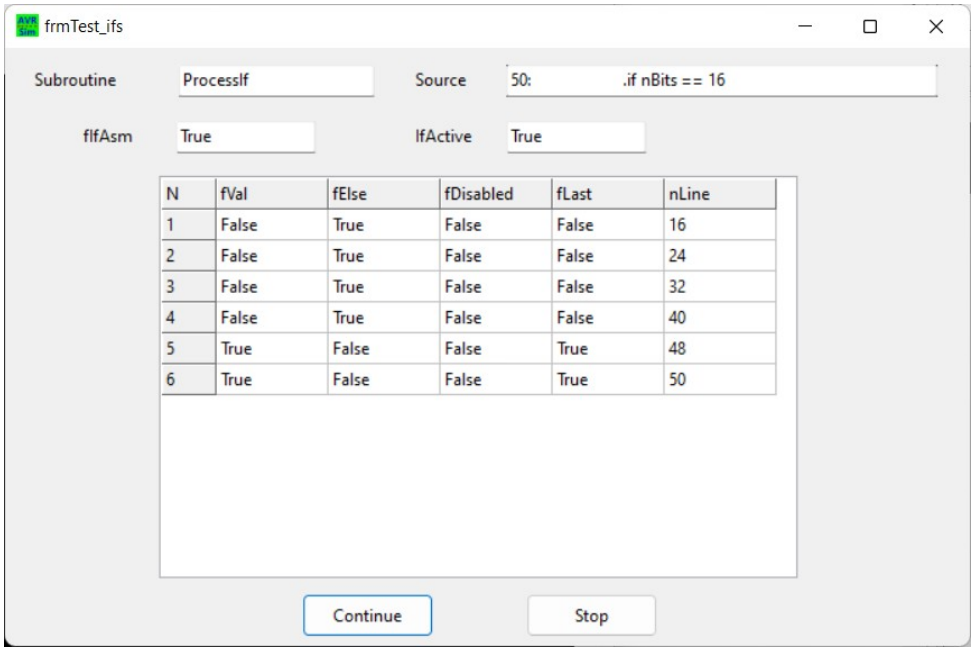
The processing is not a simple task, because any types of conditions and any nesting depth has to be processed correct. The following potential conflicts had to be avoided:

1. each .if/.ifdef/.ifndef needs an entry in the list, even if the condition is not true, because later on an .else or .elif can reverse the condition to true,
2. only if each of those has an entry, the .endif directives can be handled correct: no matter if active or not, each .if/.ifdef/.ifndef neads an entry and each .endif has to close this condition,
3. if an .if/.ifdef/.ifndef occurs in a section which is currently disabled, all following flag settings are to be disabled as well, which means that their condition is not to be evaluated (avoiding e. g. any missing symbols in equations) and that the following .else directives cannot change the fIfAsm flag any more, those entries therefore need a fDisabled flag to switch these off.

As complicated the nested .if constructions get for assembling, the nesting can get complicated for the programmer as well. For these I have added the unit “test_ifs”, which allows to follow the assembling process step-by-step and to check all conditions during assembly.

To enable this feature edit the form of the unit “gavasmu” and replace the options in edOptions.Text by the string “smewzi”: the option “i” enables debugging of all .if directives. Re-compile, start avr_sim and open your complicated project source.

When assembling, each time a directive of such a type is encountered, a window opens.



This window shows all subroutines that are called, the source line from which this is called, the current flag “fIfAsm” and the “IfActive” status.

The list entries below show all nested .if with their

- initial value “fVal” (normally the condition result, false if the .if came when fIfAsm was false), is needed when an .else or .elif is coming in,
- their .else status “fElse” (gets true if an .else has been processed), is needed to determine if already in the .else state, an additional .else produces an error message,
- enabled/disabled status “fDisabled”, which is true if the .if/.ifdef/.ifndef came when fIfAsm was currently inactive, it is needed to determine if an .else can alter fIfAsm,
- the last status “fLast”, which is the status of the entry when the next entry was introduced, is needed to recover the previous state if .endif clears the next level,
- the line of the source code, where the subroutine is called from.

If you close the modal window with the “Continue” button, the testing is continued. If you close with the button “Stop”, the testing is disabled and is only re-enabled when re-assembling the source.

9 Lazarus source code and compiling

The Lazarus source code and the Win64 executables produced have the following properties:

avr_sim properties			Size in kB (Win64)	
Version	**Files**	**Lines**	**Exe debug**	**Exe packed**
2.8	51	42.754	51.629	7.286
2.7	50	42.384	51.326	7.277
2.6	49	41.432	51.089	7.241
2.5	48	40.933	46.317	11.010
2.3	45	39.961	44.537	10.836
2.2	45	37,772	45,025	10,633
2.1	45	37,681	43.969	10.634
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 only slightly 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.

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

The reduction in the packed exe size from version 2.6 on is an improvement made by Lazarus/FPC. It can be recommended from that to always use the latest compiler version.