Blaise Gassend
François-Marie Lefevere

Tuesday may 8th 2000

DUST

Developer's Universal SX Tool

Technical Manual

Introduction

Developing for micro-controllers can be a long and tedious process. The use of assembly language is often necessary to reduce memory usage to a minimum and to save precious clock cycles. Mistakes are easily made and it is important to be able to debug the code quickly and efficiently. Micro-controllers make this task difficult. When it can be done, erasing and reprogramming a micro-controller can take up to 15 minutes with EPROM, and doesn't drop below 30 seconds in the best of cases. Once the code is loaded into the micro-controller it isn't always easy to know why the code isn't working. Many sources of malfunction exist even if the code is correct. The micro-controller might have burned out, there might be a problem with the electronics that surround it, or there might just be a bug in the code. Since the in-circuit debugging capabilities are usually very limited, knowing where the problem is can be like staring at a black box and wondering why it isn't doing what you want it to do. Developing in such conditions isn't a very efficient process so various methods have been perfected.

In most cases, the micro-controllers that are used have limited capabilities which makes it possible to simulate them on a computer. Instead of being loaded into the micro-controller, the software is loaded into a program that simulates the micro-controller's operation. In this way, it is possible to get rid of the chip-reprogramming delay and the possibility of a hardware failure. The simulator can also provide a powerful debugging environment with which it is possible to pinpoint problem spots in the code. Unfortunately, most simulators only simulate the micro-controller, leaving aside the surrounding electronic circuit. This isn't a problem when the micro-controller is the brain of a simple appliance such as a coffee machine. The signals coming and going between the micro-controller and the surrounding circuit can easily be kept track of by the programmer. This ceases to be true when the micro-controller is embedded into more complex applications, where it has to analyze complex time varying signals. Finding a bug that takes place after a million impulses have been analyzed (typically this can happen in one second) is humanly impossible if the operator has to feed the impulses into the simulator manually. Various solutions exist, allowing the impulses to come from one file and to be recorded in an another, but they are only partial solutions because the input signals are independent of the output signals.

To get rid of all these difficulties at once, it is possible to simulate both the micro-controller and the surrounding circuit on the computer. This requires quite a bit of added complexity since the electronic circuit can take on a wide variety of forms. However the problem is perfectly manageable on a modern computer.

While programming for the robot we are presenting in the E=M6 robotics competition, we have met up with most of the problems described above for the Scenix SX micro-controller. Unfortunately, there doesn't seem to be any free SX simulator that handles outside circuitry. We have decided to change this situation by writing a circuit simulator with the SX as one of the possible components. Of course it would be possible to add other micro-controllers to this simulator in the future, as it has been made to be as flexible and easily extendible as possible.

Simulating an electronic circuit

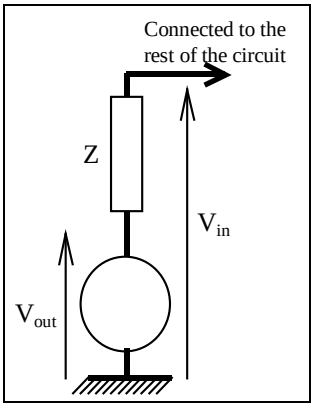
A model for the physical reality

Simulating an electronic circuit is in itself a very vast problem which is far from being solved in a general way. Specific techniques have been developed for linear circuits as well as other specialized areas. Before we start, it is thus necessary to choose a model that is close enough to the physical reality to cover all the cases that we are likely to need,

but also that is simple enough to avoid unnecessary complications that would bring no improvement in the type of use we are planning.

The simplest approach would be to consider circuits where a component has output pins and input pins and where an input pin gets a logic true or false from the output pin that it is connected to. This solution is simple and can lead to interesting optimizations. However it is not sufficient for our purposes because in typical micro-controller applications pins can be reconfigured from input to output during normal operation. When using a data bus, for instance, the same pins will be outputs for write operations and inputs for read operations.

With a little more complication we come to a slightly less simple model, but that will prove to be sufficient for our purposes. We will represent a component's pin as an ideal generator with a load resistor. The pin's output will determine the generator's voltage compared with a common ground voltage, and the pin's input will read after the load resistor, after influence from the connected components. In this case we input and output voltages instead of true or false. A typical input pin will have an infinite load resistance and will compare the input voltage with a reference voltage to determine the logical input state. A typical output pin will have a zero load resistance and will set its output voltage depending on the logical state it wants to output.

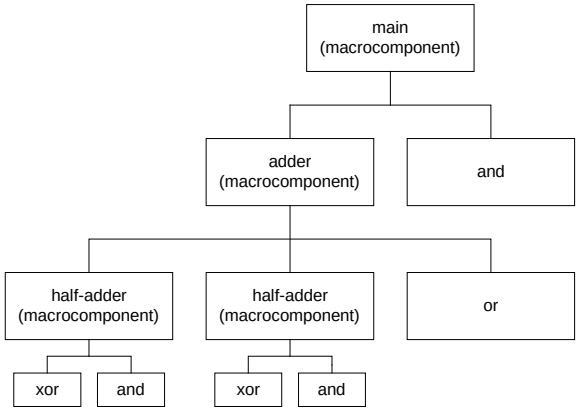


With this model a pin's function can be reconfigured in use. It is possible to simulate three state pins that are very common in micro-controller applications (they have a high level, a low level and a high impedance level where they do not influence the rest of the circuit). The use of pull-up or pull-down resistors can also be simulated (they are used to set the voltage of a « wire » that is only connected to high impedance pins).

This model is well suited to the situations that we wish to simulate. However it does have limitations that must be kept in mind. In particular it doesn't simulate propagation times. It would also be quite a clumsy model if one wanted to simulate linear circuits. From a computational point of view, it won't be possible to optimize as well as a simple logic circuit. This isn't a major problem since we aren't trying to model extremely complex circuits. Most of the time there will be a SX with only a few external components.

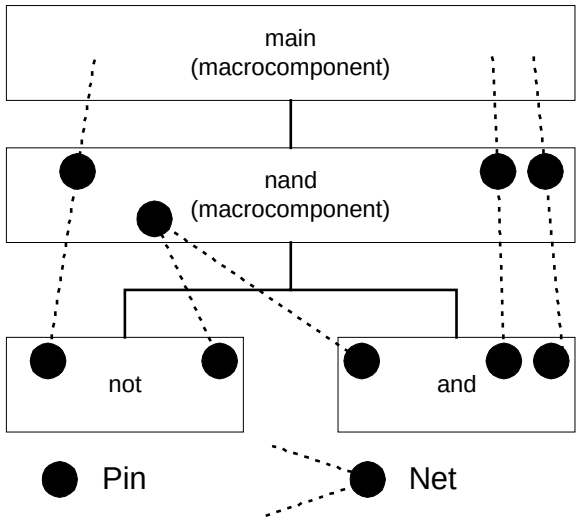
Structuring the electronic circuit

Now that we have a physical model for the pins we are going to be connecting when we create an electronic circuit, let's try to put a little bit more structure into the picture. It has been implicit in what has been said that the pins we are considering are the pins of electronic components. What are these components? Of course, the SX will be a component (that's what this is all about). There will also be components for the basic logical functions (and, or, not, xor, nand, nor, ...), for more complex logical functions (counters, multiplexes, flip-flops, memories, ...), for various generators and clocks, for data transmission circuits (parallel, asynchronous serial, synchronous serial, ...) and of course for components that we will be able to use to interact with the circuit (buttons, oscilloscopes, ...).



But what happens if we need a component that isn't included in the program? Well, we build it using the provided components. This is fine if we only use the component once, but what if we want to use it many times? Copying the definition of an elementary component dozens of times can get a little tedious. Thus it would be interesting to offer the possibility to create components from smaller components. We will call these components *macrocomponents*.

With *macrocomponents*, the circuit takes on a tree structure. There is a main component that represents the whole circuit. This components contains other components, some of which are elementary components, while others are *macrocomponents* which in turn contain other components.



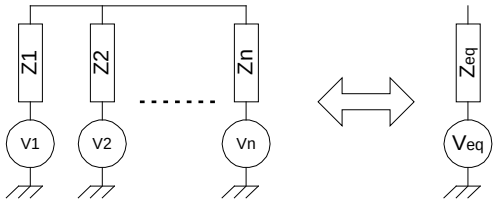
It is very nice to have all these components organized together in a tree, but we still have to connect these components together. To do this, each component will have a certain number of pins that its parent component will be able to connect to. The pins will be of the type described above. The various pins will be connected by networks of wire (that we will call nets from now on). It will often be useful for the nets of a *macrocomponent* to be considered as pins by the component that contains it, so we will consider nets to be a special kind of pin.

Making it work

Now that we have our circuit all set up, it's time to make it start working. There are two basic things that we can do :

- Combine all the output voltages and load resistances for pins connected to a net to determine their input voltage.
- Have a component look at its pins to decide what it should output.

Determining the input voltage



For a pin that is alone, the input voltage will be equal to the output voltage.

If you consider many pins connected together in a net, it is easy to obtain an equivalent pin.

The net will take on the characteristics of the equivalent pin.

The equivalent pin has the following input voltage and load resistance :

$$Z_{eq} = \frac{1}{\sum_i \frac{1}{Z_i}}$$

$$V_{eq} = Z_{eq} \sum_i \frac{V_i}{Z_i}$$

In this way, it will be possible to find the input voltage of a whole tree of nets by reducing them one at a time until we end up with a single pin that is equivalent to the whole.

The input voltage for pins on a net will be equal to the input voltage of the net.

Thus once we have the equivalent pin for a tree of nets, we know that the input voltages for all the pins in the tree will be equal to the output voltage of the equivalent pin.

It would seem that finding the input voltage for the pins on a net can be done like this :

1. *Find the pin that is equivalent to the tree of nets by reducing nets one by one. We are actually propagating information towards the root of the tree.*
2. *Send the voltage information back down the tree to the pins at its leafs.*

Simulating the behavior of a component

Since there will be a wide range of components, all we can do is plan a standard interface that will tell a component to update itself. This task is done by the *Update* function that any component must overload. Typically, this function will look at the component's pins and set new values for the pins, it might also modify the internal state of the component.

But when does the *Update* function get called? After all, a 50Mhz oscillator will have to be updated far more often than a DC power generator. Only the component knows when it should be called, thus the *Update* function has the job of setting an *updatetime* flag. Another case in which a component should be called is if one of its pins has changed state.

Putting the steps together

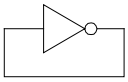
By combining what we have seen so far, we can write a simple simulation algorithm :

1. *Calculate the equivalent pins for all nets. Then set input voltages for all pins.*
2. *Update the component that has requested its update earliest.*
3. *Calculate the equivalent pins for all nets. Then set input voltages for all pins, noting which pins have had a change on their input.*
4. *If no pins have changed go to step 6.*
5. *Update all components with changed pins. Go to step 3.*
6. *Go back to 1 for the next component that wants to be updated.*

The algorithm we just put together is very nice. However, if you connect the input of a not gate to its output, you are likely to wait a very very long time before the circuit stabilizes and you get to see step 6. This is a big problem, and we have to have a look at the real world to see how to get around it.

What can I do if the circuit won't stabilize?

What would happen with a real circuit in the case where there is no stabilization?



Example of a not gate with no stabilization :

For this example, it all depends on how the gate is made. In some cases we might stabilize around an average value, in others we would have oscillations. We could try to simulate this type by making our output voltages vary slowly instead of going from one extreme to the other as soon as the component changes state. In practice, this would be very slow since we would have to do many updates of the circuit just to simulate the change of state of one pin.

Therefore, we will consider that our components will tend to oscillate between discrete states when there is an instability, and that they will do this very fast. So fast that we don't really know what state they will be in when the time comes for the next component to update. Thus, all we have to do is apply the preceding algorithm until everything that will stabilize has stabilized, and then just skip on to the next *updatetime*.

Of course there is the problem of knowing when everything that can stabilize has stabilized. We can't do a careful analysis of the interaction between the components to decide what is unstable since we don't know what is going on inside the components. (It wouldn't be a very useful analysis anyways.) What we'll do is simply wait 100 loops, or whatever is defined in *MAXITER* and give up. Since we strongly believe that any digital circuit that is usefully simulateable with Dust is without instabilities, we will also display a warning message. For very big circuits it might be useful to increase this value.

Making the algorithm local

If we think back a little, we will remember that our circuit was made up of a tree of components, and that a component is updated with an *Update* function. In the algorithm that we have been looking at we don't really use the tree structure, and the Update function only gets called for components at the leaves of the tree. It would be nicer if we could work on one node at a time and call *Update* to take care of child nodes. Here is a new algorithm that does just that (but first, we should notice that steps 1 and 2 as well as 3 and 4 are two copies of the same basic pattern, we will just show one copy of the pattern here) :

A. (At the beginning we will suppose that the input voltage is set for all nets of the current component)
B. We propagate the input voltage to all the pins connected to the nets of the current component. (I call this propagating down)
C. We update the child components. (Condition A is met for the child)
D. (Now we expect all the child nets to have had their equivalent pins evaluated)
E. We calculate equivalent pins for all nets of the current component. If a net isn't the child of another net we set its input voltage to be equal to its output voltage. (I call this propagating up)
F. (Condition D is met for the parent.)

The algorithm we have just written is exactly what we need for the *Update* function of *macrocomponent*s. Now a *macrocomponent* acts just like a normal component as far as its parents are concerned. The *macrocomponent*'s *updatetime* will be the smallest of all its children's *updatetime*s.

There is just a slight twist, we will allow nets to be connected to nets of indirect (grandchildren, etc…) children of the current component, which changes condition A slightly.

Optimizing

Up to now, we have been updating components as soon as one of their pins has changed. That's a lot of changing when you consider that a typical micro-controller has dozens of pins, that will only be sampled on changes of state of its clock pin. Therefore there we can relax the conditions in which we update a component. Instead of updating after each pin change, we will try to be a little smarter.

The method we used was for components to specify what kinds of changes they were interested in was to specify trigger voltages. If the input voltage is between *highcarevalue* and *lowcarevalue*, then the component doesn't need to hear about the change. A *macrocomponent* calculates its trigger values using its children's trigger values.

Another slight improvement can be obtained if we only propagate down (step B) when the input voltage has changed.

There is still work to be done in this field since for now simulating one second of real-time on a 66Mhz SX takes about an hour. This slow speed isn't really a problem for debugging since the code can usually be debugged in pieces that don't take too long to execute.

What's going on in my circuit?

The namespace

Great! It's working, our circuit is being simulated, the optimizations are sufficient for it to be going pretty fast. But it doesn't seem to be doing exactly what I wanted, there must be a bug. How can I see what's inside?

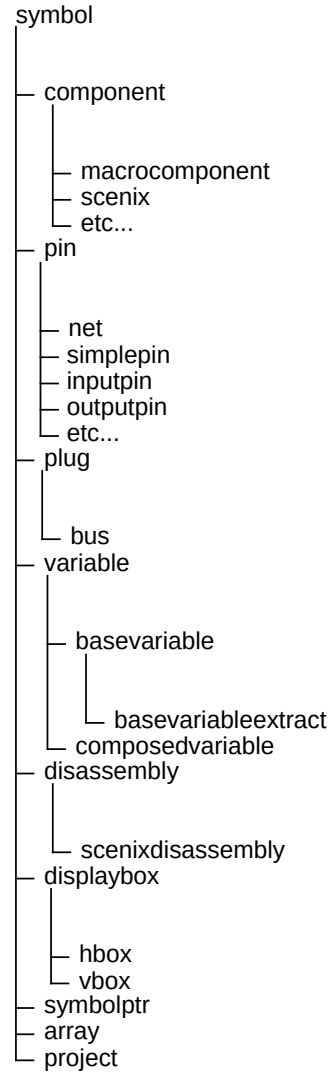
That's exactly what we are going to do now. One of the key parts of a debugger is to give the developer access to as much information as possible to help hip pinpoint where the problem is. We've set up a simulation of an electronic circuit, now it's time to give ourselves the tools to look inside a working circuit. What kinds of things can we look at?

- The most obvious thing to look at in an electronic circuit is its pins. This is traditionally what one would do with an oscilloscope with a real circuit.
- In many cases, there is more to a component than its pins, most circuits from the smallest flip-flop to megabyte memory chips, have internal states. Our big advantage over the real world, is that in our simulation all these states will be accessible.
- Keeping in mind that we want to debug circuits containing SX micro-controllers, we will have to be able to access all the internal states of the SX, preferably in an interesting form (disassembly of the code, the memory as an array, etc...).

It would seem that it is often the circuit that knows what kind of information it has to offer. This means that it would be logical for the information about a component to be accessible directly from the component tree. Thus we will end up with a tree that is much bigger than the one we had, which contains the circuit as well as all the information we might like to know about it. If we give everything in this tree a name we end up with a structure in which each element has a name. We will call this structure the namespace, and we will call the various elements it contains *symbol*s. The root of this hierarchy will be called the *project* and *main* will be one of its descendants, its other descendants will be the windows we want to open in the user interface.

Implementing the symbols

Implementing this structure comes quite naturally in object oriented programming. We will have a base class symbol, from which all the types of symbols will be derived. The result is flexible and easy to use. Here is part of the hierarchy we used :



Lets have a closer look at some of these symbols.

project

This class serves as root for the symboltree. It also keeps track of the windows of the user interface and is responsible for calling the update function of the main component.

displaybox

This symbol is used to group many symbols together for display purposes. There are two flavors that group the symbols together horizontally or vertically.

symbolptr

Simply copies a symbol from one part of the name space to another. It is handy to put symbols in a hbox or a vbox, but also if a *macrocomponent* wants to mirror the symbols of one its children for convenience.

disassembly

Dissemblers always have a window to show the code you are debugging, to set breakpoints and so on. The derived classes of the disassembly symbol do this for the components that need them (for now only the Scenix, but just wait and see!).

variable

Variables are integer values of up to 63 bits (you can try 64, but you will probably have sign related trouble). They can be based on an integer value directly (long, int, char) with the *basevariable* and *basevariableextract* classes or they can be built up from other variables with the *fusionvariable* class.

array

Arrays are made to be used for memory, they take their data directly from an array.

pin

The pin class sets a standard interface for all types of pins. It has a wide range of derived classes that simulate various types of pins (input, output, three state, etc...). Anything that can be connected to a net derives from the pin class. Thus net also derives from pin.

plug

Plugs were originally made to make it easier to connect many pins at once using a bus. It was then extended to allow grouping of any kind of symbol for naming convenience.

component

All components are derived from the component class. In particular this class has an Update function that is called when the component's state needs to be updated. The *macrocomponent* class derives from the component class, its update function takes care of propagating signals up and down the nets and calling its children's Update functions.

Attributes : aiming for flexibility

We will often want to pass parameters to components when we create them (a clock's frequency, a generator's voltage, etc…). Other symbols will also want information specified in the source file. Since each type of symbol has different kinds of attributes that have to be set we will use a general parameter passing convention that will be easily used by each symbol to get the information it needs.

Basically each symbol will receive a linked list of attributes, each attribute being an association between a name and a value that can be an integer, a real, a string or a name. The symbol just has to find the names that it wants in the list and use the corresponding values.

In practice attribute handling is one of the most delicate jobs of all. There is no difficulty in the attribute class itself, what is trickier is making the attributes easy to use. Rather than specifying an attribute 100 times for 100 children of a given symbol, the user will be pleased to know that he can specify the attribute once for the parent symbol and have it be inherited by all the children. But to make things interesting, there are other cases where we won't want inheritance. To make things more fun and to save memory we don't want to make copies of attributes that are inherited. That means that we have to be careful when manipulating attribute lists because many symbols will be using them. In practice big systems of attribute lists are set up when the source files are parsed, whereas attributes that are modified during use are sent individually.

A secondary use of attributes is for messages to be passed around in the Linux user interface, with names that couldn't have been typed by the user.

Building the symboltree

Parsing

The input files are parsed with a parser generated by bison and flex. It would have been totally unmanageable to manually create a parser for our language, and it would have seriously limited out ability to make changes to the grammar and make improvements.

The parser uses the source file to make *macrocomponenttypes* the component types that are defined by the user, as well as to prepare constructors for all the symbols.

Symbol constructors

A symbol's constructor is an object that will store information about the symbol until the symbol is ready to be created. If the symbol has references to other symbols, their names will be given as-is to the symbol's constructor. Once everything has been parsed, it is time to create the actual symbols. This gets done recursively each symbol's constructor being called when its parent gets a request for it.

The symbol table

Up until now, I have been rather vague about what a symbol's name is exactly. It is in fact a list of integers. The integers have a meaning that comes from the *symbtable* class. The *symbtable* class takes all names (letter or underscore followed by the same or numbers) and assigns them an integer value. This value is negative and even. Negative and odd is to identify unnamed symbols and positive or zero is for positive integers.

The Graphical User Interface

The graphical interface is almost totally independent of the rest of the program. This was necessary for it to be able to work with two totally independent graphical environments. The only place where the user interface appears in the simulation engine is the *symbol::getwidget* function, and a few functions in the *project* class. This function constructs a graphical interface element (a widget) for the symbol. Each user interface provides a standard set of widgets for the simulation to hook on to.

Lets have a quick look at how the user interfaces work.

Linux GUI

The Linux GUI uses GTK+ to create its interface.
It uses a number of derived classes of *widget* to work.

symbolwidget

The *symbolwidget* class is the base class for all the *widget*s that display a symbol. Each symbol is free to do what it wants here. All the standard behaviour comes from the *symbolborderwidget* that each symbol widget is inserted into.

symbolborderwidget

The *symbolborderwidget* provides the standard interface elements for the symbol that it contains. It takes care of the drag and drop interface, the context sensitive menu (that isn't context sensitive), displaying the symbol's name and border, etc…

window

The *window* class creates a top level window that can hold a *symbolborderwidget*. It also takes care of the tool bar.

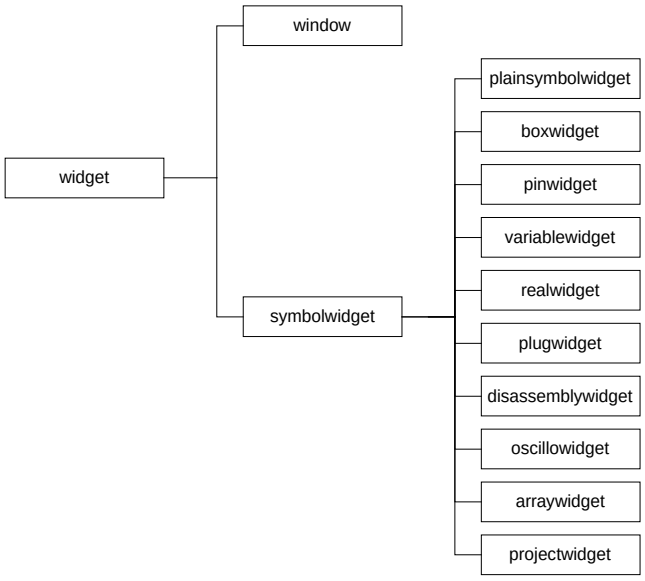
boxwidget

The *boxwidget*s are symbolwidgets, but they have a special role in the user interface as they handle the reception of the drag and drop interface. Box widgets can have their structure completely modified by the drag and drop process, thus, after a while, they don't reflect the contents of their symbol anymore.

Windows GUI

The Windows GUI uses the Borland C++Builder Visual Component Library (VCL). The VCL components are embedded in a more portable class : the widget class which is the common ancestor of each item displayed on the screen.

The hierarchy of the widgets is the following:



window class

The window class creates a top level window that can hold a *boxwidget*. This class uses the *TWindowForm* class to display a top level window. The *TWindowForm* class is a descendant of the *TForm* that is the standard class for windows in the VCL.

symbolwidget class

This class is the base class for all the widgets that display a symbol. It owns virtual functions to handle drag&drop and sizing operations.

The auto-size and auto-alignment process.

To create a smart interface, the widgets must be aligned. In vertical boxes, we want all the widgets to have the same width, in horizontal boxes we want all the widgets to have the same height. To position and size widgets correctly, the *window* widget calls the *getsizewanted* function for all its descendants.

When the *getsizewanted* function of a *symbolwidget* is called, the input parameters (x,y,ox,oy) contains the current position (ox,oy) and the maximum size (x,y) wanted by the preceding widgets in the same box. The *symbolwidget* uses (ox,oy) to position itself, then it checks if the current maximum size (x,y) is large enough. If this is not the case, it updates the values of x and y to the size it needs.

A *boxwidget* calls the *getsizewanted* function for all its descendants but with a local set of parameters that are initialized to (0,0,0,0). Then, after calling each descendant, it updates *ox* and *oy* to put the *symbolwidgets* horizontally or vertically in accordance with its type. At the end of the operation, the *boxwidget* synthesizes all the information and returns the extent it needs in *x* and *y*.

After this first pass, all the widgets are at the right position but they do not have coherent sizes.

To solve this problem, another function *setsize* is called recursively for all the *symbolwidgets* contained in the window to adjust sizes. The principle of operations is quite the same as the one mentioned above.

The SX simulation

This part of the job doesn't contain special difficulties. The simulation is only a succession of tests and switches in accordance with the SS micro-controller's documentation (it just needs patience and care). The simulation process is split into two different parts:

- the CPU simulation which handles the code execution.
- the simulation of special integrated items like watchdog timer, internal oscillators, external interrupts and reset pin.

Another function is dedicated to the disassembly of the SX opcodes.

Future improvements

- Get rid of the memory leaks and try to limit the number of referenced but unused objects.
- Thorough checks to prevent connecting a net to a pin that isn't below it.
- Detect and prevent giving identical names to symbols. For now the behavior is undefined.
- Data stream components to generate and interpret data transmission signals (SPI, RS232, parallel, etc…)
- Entering attributes interactively.
- Read symbol names in .lst file to get mnemonics for memory areas of the SX.
- A very flexible breakpoint engine is contained in this version, but the user interface for it is non-existent so far.
- Improve the linux oscilloscope widget. For now it is less that bare bones.