Spider Programming Manual

© Steim Foundation

PREFACE

Welcome to SPIDER, a language & development system used to define and program electronic musical instruments (midi controllers) using the STEIM SensorLab. The SensorLab is a hardware device capable of converting real-world data (sensors) into midi data. The relation between the sensor inputs and the midi data produced by the SensorLab is defined by a 'program' or 'configuration file' written in the SPIDER language. This program is written in and translated by the SPIDER development system and sent to the SensorLab through midi *System Exclusive*. Once in the SensorLab the program will completely determine what the Sensorlab's response to the sensor data will be. When you're satisfied the configuration will stay in the SensorLab and the host computer (apple macintosh) will no longer be needed.

This manual will guide you through the system, describing the development environment and the SPIDER language proper. Since the system is made for performing artists, many of whom will have no or very little programming experience, I will go into much detail where the SPIDER programming language is concerned, using many explicit examples. More experienced programmers could find this a bit tiring and may want to skip sections. Users completely new to computer programming may want to read an introductory book on computer programming in general. SPIDER is a very specific language, but many of the general programming concepts are similar to other languages, most of all to the programming language 'C'.

SPIDER is a very dangerous tool, it is very easy to do an awful lot of things, including making serious mistakes which might completely spoil your performance or piece. Finding errors in computer programs, even programs written is a relatively simple language like SPIDER, can be a long and tedious business. The SPIDER development environment helps a bit, but only careful work and a lot of testing will get you to where you want to be.

The SPIDER / SensorLab combination is a system in developement, suggestions and remarks are welcome. Please feed back.

Tom Demeyer.

System requirements & installation

SPIDER runs on all apple powermacintosh computers running system 7.6 or later. It requires a modest amount of memory, and the speed on the older mac's is perfectly acceptable. In order to be able to test your code and do something useful with it you will need to have OMS installed on your system. When not installed, SPIDER will give you a message telling you so, but will allow you to continue editing you code. You will not be able to test it, however. SPIDER will allow you to compile your code and warn you of any syntactical errors, but logical errors in the code will only be detected once you've sent the compiled code to the SensorLab and have run it through its paces. To install SPIDER on your mac simply copy it to to directory of your choice.

Hardware installation

All communication between SPIDER and the SensorLab is done through midi. A fairly typical development setup would have a midi cable running from the mac midi-out to the SensorLab midi-in, from the SensorLab midi-out to your midi equipment (the midi machines you'll be using in your actual performance) and from the midi-thru port of your midi-equipment back to the midi-in port of your macintosh. This will allow you to write code, compile and send it to the SensorLab, and while testing your code on the actual instrument (or whatever setup you've connected to the SensorLab), check the midi data coming out of the SensorLab on SPIDER's built in midi-monitor.

If you have a completely virginal SensorLab the SensorLab's error leds (the two yellow leds on the front panel) will indicate an invalid configuration by keeping the left led steady while blinking the right. To test connections and setup you can start up SPIDER, which will open with a blank 'Untitled' file. When you now choose 'Send' from the SensorLab menu an empty (but valid) configuration will be sent to the SensorLab. This should, if all is connected well, cause the error indicators to switch off. If your SensorLab is not empty and your error leds are not blinking, choose 'Stop SensorLab' from the monitor menu. This should start the right-hand led blinking, indicating a 'stopped' state (more about this later).

The Spider editor and development environment

SPIDER's built in programming editor is a basic, some frills, editor without much sophistication. It is not strictly necessary to use SPIDER's built in editor, as long as you produce 'TEXT' (a word processor in 'text' or 'ascii' mode) files you can use any program you like. SPIDER will be able to read them in and compile them.

Once properly hooked up to a SensorLab a typical development cycle would go from a bit of editing through to the 'send ' command, then a bit of testing with the instrument under development, checking SensorLab output on the attached midi instruments and the SPIDER monitor and back to a bit of editing.

Let's tour the menus in order:

File

New _N This will open a new, untitled window, ready for immediate input, you will be asked for a name when you save this file, or add it to a project (see **Project**) through the standard macintosh file selector dialog box.

Open _O Brings up the standard file selector, allowing you to open a TEXT or SPIDER file of your choice for editing.

Close _W Closes the top window, if the file has been modified since the last save you will be prompted whether to save the file before closing.

Save _S Saves the file in the top window to disk.

Save as... Brings up the standard file selector for you to choose a new filename for the file in the top window and saves it to disk.

Revert Replaces, after prompting for confirmation if the file has been changed since the last save, the file in the top window with the last version saved to disk.

Page Setup... Gets the standard mac page setup dialog up on the screen to allow you to change some printing parameters.

Print... _P Prints, after displaying the standard print

dialog, a basic, no frills, version of the file in the top window.

Quit _Q Quits SPIDER, asking to save any unsaved work.

Edit

Cut _X

Copy _C

Paste _V

Clear Standard mac editing commands, see macintosh user guide if you are unsure.

Find... _F Brings up the Find dialog. Enter the text you want to search for in the top edit box and press *Ok* to find the first occurrence of the text. The search commences towards the end of the file and continues at the beginning until the whole file is scanned. When the 'ignore case' button is checked the search will be case insensitive. Text can optionally be replaced by something else by entering the replacement text into the second edit box in the find dialog (see next entries).

Find again _G
Searches for the next occurrence of the last text entered in the find dialog.

Replace & find again _H
Replaces the current selection (may or may not be selected by a **Find** command) with the text in the replace box of the find dialog, then proceeds to find the next occurrence of the text in the find box of the find dialog.

Match braces _B
This will select the block of text between the innermost pair of braces - '{' and '}' - around the insertion point.

SensorLab

Compile _K This function, only available when a text window is topmost, will compile the text in the front window, bringing an error window to the front if necessary. If there have been no changes since the last compile command nothing will happen. In the error window, clicking on an error line

will bring the corresponding text file to the front with the offending line selected. If a project file is open the compile command will start compiling all the files in the project in the order in which they appear in the project (see the **Project** menu entry).

Send _T This command will first perform a compile step (see above) and then, if no errors are found, it will send all information which has changed since the last ‘send’ or ‘send all’ command to the connected SensorLab. Care should be taken if there has been any mucking around with midi cables between ‘send’ commands, because SPIDER may in that case have a wrong picture of what exactly is already in the SensorLab, and thus might not send enough information to the SensorLab after a successful compile. As a rule of thumb, after changing cables or in any other way changing the setup, use the ‘send all’ command instead of the ‘send’ to make sure that both SPIDER and the SensorLab agree on the state both are in. **Send** will automatically revert to ‘send all’ the first time it is used. Also, after sending a syntactically correct but otherwise disastrous program to the SensorLab, the SensorLab may be in a state where its memory is corrupted to such an extend that a **Send all** command is necessary to restore the SensorLab to a healthy state.

Send all _A See **Send**, the difference is that all information is sent to the SensorLab, taking more time than the send command, but making sure SPIDER and the SensorLab are synchronized with respect to the compiled image of the SPIDER code in their memory.

Monitor

Open monitor _M This will open the midi monitor window, allowing you to see whatever comes in on the mac’s midi in port, and to send out any midi command using the

narrow text input box at the bottom. Incoming midi data is sorted left to right according to the status of the message; from left to right: note on and note off, controllers, pitch bend, channel pressure, program change, poly key pressure and at the right system exclusive messages. The monitor comes up in hexadecimal display mode, allowing you to see at a glance which midi channel a particular message came in on. When sending out midi using the monitor you are expected to type hexadecimal bytes into the text box too. The text box keeps a couple of lines in a buffer, saving you a lot of typing in some cases. To retrieve a previous line just use the cursor up (arrow up) key on your keyboard; the previously typed lines will come scrolling by, ready for editing or immediate transmission by hitting <return>. Typing the word "version" will give you the version date of the SensorLab rom software (if a SensorLab is attached of course).

Clear monitor <esc> Clears the midi input buffers and the monitor window, bringing it to the front if necessary. This command can also be executed by hitting the escape key at any time.

Start lab If the SensorLab is in a stopped state (see below) this command will send a start message to the SensorLab.

Stop lab Sometimes the SensorLab is sending midi in such quantities that the mac is so busy updating the monitor window that response is becoming to slow. In this situation it is sometimes very convenient to be able to shut the SensorLab up (especially when you have some noisy equipment attached as well). This command will send a system exclusive message to the SensorLab which will cause it to stop processing the attached sensors. The rightmost yellow led on the SensorLab will start blinking, indicating that it is in a stopped state. To start the

SensorLab again you have several options. Use the **start lab** command, reset the SensorLab or use the **send** or **send all** commands.

Decimal output or **Hex output**

Will toggle the monitor's display and send midi number system. The current number system will be indicated in the lower left hand corner of the monitor window.

Spider sensor subsystems

Following will be a short description of the various SPIDER sensor subsystems, a more detailed description of each of these will be given in the language reference section of this manual where the specific sensor declarations are discussed. The keywords **dgroup, analog, usound, timer** and, for midi inputs, **midi_non, midi_nof, midi_ctr, midi_pbd, midi_pkp, midi_pgc** and **midi_prs** all define a sensor subsystem, an *input* from the programmer's point of view. This *input* will, when defined, generate an *event* which usually needs to be dealt with. You 'deal' with an event by writing a -generally small- section of SPIDER code, to be executed by the SensorLab when this *event* occurs. See SPIDER language reference further down in this manual for information on how to actually implement all this.

The dgroup system:

Digital keys (simple make/break or pushbutton switches) are defined in SPIDER through the **dgroup** command. Before simply defining all the keys used, it is necessary to think of the functions the keys will have and split these keys into functional groups. It makes sense, for instance, to have keys controlling the function of other keys separate from keys that just generate midi notes. Through the concept of keygroups SPIDER allows keys which have a similar function to have the same name, by which they are programmed. Also, these named groups provide the possibility to **swap** the functions of whole bunches of keys simultaneously.

In order to be able to assign a function (or action) to a key, to be executed when the up- or down status of the key changes, the following naming scheme has been adopted:

KeyGroupName.keynumber.mode.dir:

is the label to which a particular piece of code is assigned. Every time the key concerned is pressed or released the code starting at this label is executed up to the first **end** statement.

KeyGroupName

Is the name of the group the key belongs to, a name best chosen to reflect the function of the key.

keynumber

Is the ranking number of the key within the definition of this particular group (see example in DGROUP)

mode

Is the mode a key is in, preceded by the letter 'm'. It is not necessary to explicitly give the mode - number '1', for **dgroups**, this is the default mode and the ‘m1’ can just be left out (‘m1’ must be supplied for the other *inputs*, though.

dir

Is the direction, the letter '**u**' or the letter '**d**', meaning the up event or the down- event of the key. A typical example would be a midi note-on at the down-event and the corresponding note off at the up event. See the **dgroup** command for more information and examples.

The analog system

There are 32 analog input channels available, with 8 bit resolution. The user has available a programmable attenuator/gain, programmable offset voltage and polarity, and a programmable inverter. These parameters are given their default values in the declaration of the analog channel, but can also be changed dynamically for use in more sophisticated designs. Experience shows that 'tuning' the analog channels for a given instrument design can be one of the most time consuming parts of the programming phase. After a rough approximation it is usually a matter of trial and error before a satisfactory response is obtained. The use of dynamically changing gain, and offset voltages allows a theoretical resolution of more than eight bits, the user changing the response of the a/d converter according to its output. It should be noted, however, that this will be a time consuming process in terms of development and ‘tuning’ time. See the **analog** command description for further information.

The ultrasound system:

Provided for are two ultrasound transmitters and three ultrasound receivers, giving a total of 6 ultrasound channels. The ultrasound system is used to measure the distance between transmitter and receiver. In controlled situations transmitter and receiver may be mounted side by side, in order to measure the distance (times two) to a sound-reflecting surface. The system offers an accuracy of better than 1/3rd of a millimeter on short distance scales. The maximum practical distance over which the system may still

work reliably will be around 10 meters, maybe as much as 15. With increasing distance the response will suffer, though, because the decision that the sound pulse is ‘lost’ can only be taken after a longer time. The user has, for each channel, control over where (in space) the start point is located, meaning that the variables will only be updated when the distance is larger than this value, and the length of the active area. These ‘active areas’ can, for each channel, be given more than once, resulting (provided the start point and length define areas that don't overlap) in several distinct ‘responsive areas’ within the trajectory from transmitter to receiver. One may in this way define, for instance, continuous response between 5 cm and 1 meter, a trigger between 150 cm and 152 cm, and again a continuous area from 2 to 4 meters. The order in which the channels are defined is the same as the physical order on the connectors. The first three channels defined will react to transmitter one, the second three to transmitter two. If one receiver and two transmitters are to be used one should (apart from considering to use one transmitter and two receivers - amounting to the same thing) define four ultrasound channels of which the second and third dump their data into dummy variables. See also **usound** command.

The midi-in system

The SensorLab allows for 32 midi-input ‘sensors’, triggering a section of SPIDER code whenever one of the specified midi messages is seen on the lab’s midi input port. These could just send the message back out, (providing a ‘programmed thru’ for which there is a faster generic **thru** command) or do something different altogether. The midi parameters of the incoming message constitute the ‘sensor’s’ data and are available to the SPIDER code.

The timer system

The SensorLab has available 16 timers, which can be set to trigger a piece of program at regular intervals. These intervals can be changed dynamically. The timers are not meant for sequencing or other jobs which require accurate timing; they are not good enough. Depending on the load of the SensorLab the ticks may be delayed a bit. Resolution of the timers is in 100th’s of seconds @16Mhz (the standard setting, see hardware manual).

Spider language reference

General structure of a spider program

A SPIDER program starts with a series of *declarations*, here the various analog, ultrasound, switches, midi inputs and timers are declared and configured, **vars** and **tables** are declared here as well, usually. These declarations do not necessarily have to be at the beginning of a file, although it will do a lot towards general legibility when all this stuff is grouped together. A restriction is that things will need to be declared before they are used (with the exeption of *labels*). If the configuration (SPIDER file) is going to be big it makes sense to put all declarations in a separate file, open a **project** for your work and put the declarations file at the top.

All declarations except the **var** and **table** declarations set up *events*; (a key closure, a change in analog voltage or ultrasound distance, a timer expiring or a midi message coming in). These events make the SensorLab jump to one of eight predefined sections of code - one for each of the *modes* (see below) a sensor subsystem can be in.

This means that for each event generating declaration you will have to write a minimum of one and a maximum of eight sections of SPIDER code.

Modes:

All the sensor subsystems can be in one of eight **modes**, they all start out, after a reset, in mode 1. A different **mode** simply means that a different section of code is associated with the same sensor. A key, for instance, may send out a note in mode one, and a program change in mode two. **Modes** are selected for each defined *input* using the **swap** command. **Dgroup** keys are a bit special; when they are **swapped** to a different mode all the keys that are currently in a 'closed' or 'down' situation and are in the keygroup which is **swapped,** receive a simulated key-up event before being swapped to the new mode. After the **swap** a 'down' event is simulated again. This feature is designed specially to avoid 'hanging notes' on connected synthesizers when swapping midi-channel, or changing, for instance, a global transposition offset, but can be made use of in different situations as well.

The SensorLab associates different sections of code with

the different event generating systems in their various possible *modes* through a system of *labels*.

Each *input* declaration statement supplies an *identifier*, the name by which that particular *input* will be know in the rest of the file. It makes for readable code when this name reflects the purpose of the *input* it is referring to. This name will be used to specify the sections of code belonging to that *input*. The name will also be used to address the *input value* (for **analog**, **usound** and **midi_in** *inputs*) and can in some cases be used to acces the *input's* control data structure. To address one of the eight possible sections of event handling code -for all *inputs* except the **dgroup**- the name is used as follows:

```
name.mx:   // the x to be replaced by one of 1 - 8 for the 8 modes
      (code
        here ...)
```

To acces the value of an *input* the name is used as if it were declared as a **var**, with the difference that writing to these '**vars**' is not permitted. In the case of a **timer** *input* writing *is* permitted, to change the resolution of the timer and to restart, when necessary.

```
var newNote;
midi_non inNote,omni;       // define input 'inNote'

inNote.m1:                  // label for midi_non input 'inNote'
                            // when in mode 1.
      newNote = inNote;     // get the 'value' of the input inNote
      end;                  // in this case the midi note number
```

Dgroup labeling is more elaborate, see the section on **droup** above, and the **dgroup** keyword entry. Examples will be found in the sections dealing with the declarative keywords themselves.

Labels can also be declared independent from events, then they are just an **identifier** followed by a colon (**:**).

```
sustain_on:
panic:
_switch_channel:
WeIrD_cAsInG:
```

are all valid labels. They are used as targets for the **goto** and **call** keywords. A special label **reset:** is used to define

a reset routine, which is executed when the SensorLab is switched on, or when the SensorLab is manually reset. It can also be jumped to, it acts as a normal **label**.

The sections of SPIDER code themselves are basically a series of statements terminated by an **end** statement. Statements are either single statements or compound statements, the latter being a means of grouping statements so that they can be referred to as a single entity for use in **if** **, else** and **while** statements. Compound statements are a set of single statements enclosed by curly braces (**{** and **}**); See the sections on the **if** or the **while** keywords for examples. A single statement is always terminated by a semicolon, just like in ‘C’.

The double forward slash (**//**) is a comment, this will cause the compiler to ignore everything following it until the end of the line on which it appears.

Variables and tables

SPIDER has basically two types of **var**iable, the 16 bit integer and the **table** of 8 bit integers. Variables and tables are used through an **identifier** - a name, if you like. **Identifiers** always start with a letter or an underscore (_) and are followed by any length of alphanumerical symbols (letters, digits or the underscore). **Identifiers** are **case insensitive**; `myVar`, `MYVAR` and `myvar` all refer to the same variable. All assignment and arithmetical operations on variables work with 16 bits, except in the following cases, where only the *least significant* (low) byte is used:

- In **table** addressing mode; this is used whenever addressing an element of a **table** by using a **table** name with an offset as in `a = convert[b];` (see the **table** keyword). This mode can also be used to get pieces of a variable declared through a normal **var** statement; see **var**.
- When addressing the *input* systems; analog, ultrasound, and midi-in input variables always have an 8 bit result, the *high byte* (the most significant byte) is zeroed. In other respects they behave just as normal variables, although assignment to these variables clearly makes no sense.

Expressions:

At many places later in this manual you will find the word 'expr'; this stands for 'expression' and should be read as:

- a *number*, which can be either ordinary decimal 1, 2, -765, etc, or hexadecimal: $1, $2, $15, $77f, etc. One can also use character *literals* of the form ‘a’ or ‘7’; note the apostrophes, they are part of the syntax. These character literals are immediately converted into their ascii number equivalent and act just as numbers. So ‘A’ and 65 are identical ‘numbers’ as far as SPIDER is concerned. All numbers are 16 bit integers, running from -32768 to 32767 inclusive. With midi commands only the least significant 4 bits (for the channel) or least significant 7 bits (for other parameters) are used. This can be seen as only using the remainder after division by 16 (4 bits) or 128 (7 bits). With negative numbers it might get a bit confusing, -1 becoming 127 or 15; -2 will be 14 or 126 etc.
- an *identifier*, that is, a variable name which has to have been declared before it is used in any expression.
- Any combination of the above strung together with any of the following operators. Operators are given in order of precedence, i.e. if you see something like '`a + b * c`', '`b*c`' will be evaluated first, then `a + (b*c)`. In this case one says that the '*' operator takes precedence over the '+' operator. In the table below operators with the highest precedence are given first. Operators in the same section have equal precedence and are 'left binding'. Left binding means '`a * b / c`' will be evaluated as `(a*b) / c`; i.e. is evaluated from the left. An expression can also contain a **`random`**`(max)` statement, which will produce a random number from 0 to max-1 inclusive. Some of the operators are denoted as *boolean*. This means is that they will only have two distinct values, *true* and *false*. Typically they are used in 'if' statements, their actual values are 0 for *false* and -1 for *true*, although in a boolean context (in an if statement for example) any value other than 0 will be treated as *true* and can be used as a boolean;

```
VAR a = 20;
if(a) { do something }
```

  is legal and often very useful;

(...) Parens group subexpressions, and are used to force a

different order of precedence as needed; ' (a+b)*c ' makes sure the addition is performed before the multiplication.

The following four operators are 'unary', they only affect the subexpression immediately to the right of them.

? : This operator leaves a *true* (-1 numerically) if the variable to the right of it has changed since the last time it was checked at this place. On a number it would always leave a *false*, because the number doesn't change over time.
For example:

```
distance = programchange[rawdist];
if(?distance) pgc(0,distance);
```

would read the supposed ultrasound distance variable rawdist, convert it into a suitable program change number through a previously defined table and then sent it out as a midi program change command (on midi channel one) only if the resulting value is actually different from the last sent program change **from this location.** SPIDER does not keep track of program changes sent from other locations in the code, the ‘changed’ operator works locally only.

! : Logical negation, if the subexpression immediately to the right evaluates to *true*, it would be changed to *false*, and vice versa. For example:

```
if(!sent) {
     non(Mchannel,theNote, Velocity);
     sent = TRUE;
}
```

Where sent, Mchannel, theNote and Velocity are all previously declared variables.

~ : bitwise negation; every bit in the 16-bit result of the subexpression to the right of the ~ will be flipped, 0 to 1 and 1 to 0; you will need this only very rarely.

-: negates the subexpression to the right, 1 becomes -1; -20 becomes 20 etc.

The following five are the regular arithmetical operators, they behave just as expected. It should be noted that the division and modulus operators are

expensive in terms of execution speed. Whenever possible, that is whenever 'modding' with a number of the form 2^n-1 (so with numbers like 3,7,15,31 etc) use the **&** (and) operator , this will execute much faster. Division by a number of the form 2^n (2,4,8,16,32 etc) should be done using the shift right (>>) operator.

expr **/** expr divide (`11 / 3` evaluates to `3`)
expr **%** expr take modulus (remainder after division)
expr * expr multiply
expr + expr addition
expr **–** expr subtraction

The next three are bit operators, these operators work on the bit level and are useful to mask out midi channels and such things. Bit manipulation needs a working knowledge of the binary representation of numbers in a computer .

expr **&** expr bitwise and
expr **|** expr bitwise or
expr **^** expr bitwise exclusive or

```
channel = channel & 15;
```

forces the variable `channel` into the range 0 - 15. The SensorLab will actually take care of this when the var should be out of range, but this is just an example, right? Be aware that we're talking bits here;

```
myVar = myVar & 16;
```

will **not** force myVar to be in the range 0 - 16, rather, it will set it to either 16 or zero, depending on its previous value.
The 'exclusive or' operator is very convenient to 'toggle' the state of a boolean variable;

```
var sustain = 0;          // declaration of 'sustain'
controlKeys.1.u:          // entry point for a particular key
    sustain = sustain ^ 127; // 127 becomes 0 and
    ctr(0,64,sustain);        // 0 becomes 127
    end;
```

This small program will alternate between switching sustain on midi channel one on and off.

The next two are also operators on the bit level; ‘ a >> b’ shifts ‘a’ ‘b’ places to the right.

expr >> expr shift right
expr << expr shift left

The first of these provides an efficient way of dividing by multiples of two; the latter of multiplying by powers of two:

```
channel = 1;
while(channel & 15) {
     non(channel,40,127);
     channel = channel << 1;
}
```

will send note messages on midi channels 2, 3, 5, and 9 (remember that channels start counting at zero).

The next two operators compare the two subexpressions on either side and are replaced by either the maximum or the minimum of the two.

expr <> expr maximum
expr >< expr minimum

The remaining operators are all boolean, they evaluate to a *true* or *false*, a -1 or a 0.

expr > expr greater than
expr < expr smaller than
expr >= expr greater than or equal to
expr <= expr smaller than or equal to

expr == expr equal
expr != expr not equal

expr1 && expr2 logical and
expr1 || expr2 logical or

The last two operators leave a *true* if both expr1 and expr2 are *true* or in the or case if either expr1 or expr2 is *true*.

Lets give just one example for the last bunch:

```
if( a && distance > 150 && ( pressure <= 10 || always == true))
     a = false;
```

will set `a` to *`false`* when a is *true*, distance is bigger then 150 and either pressure is smaller than or equal to 10 or always is nonzero.

Again in order of precedence:

( )			
?	!	~	-
*	/	%	
+	-		
&	|	^	
>>	<<		
<>	><		
>	<	>=	<=
==	!=		
&&	||		

Spider keywords

var <identifier> (= <number)(, <identifier> (= number), ...);

```
var a;
var x,y,z;
var sustain = false;
var breath = 2, modulation = 1, pan = 10;
```

This keyword defines a basic 16 bit variable, it can later be referred to by its name (the identifier). The optional initialization can set the value the variable starts out with, **only** after sending it to the SensorLab from SPIDER. Since memory in the SensorLab is backed up by a battery, changes to the contents of variables will survive power down and reset situations. To set a variable to a know state after a reset or a power up, you would have to include the initialization statements in the **reset** routine:

```
reset:
    sustain = false;
    breath = 2;
    modulation = 1;
    pan = 10;
    end;
```

In this declaration stage the right hand side of the assignment can only be a simple number, no expressions at this stage, the assignment is evaluated in SPIDER on your mac, not in the SensorLab. The two reserved words **true** and **false** count as simple numbers too; they are immediately converted into -1 and 0 and are thus completely equivalent to these numbers. In the SPIDER code proper variables are assigned values using the **assignment operator** , the = sign. Assignments take the form:

```
a = <expr>;
```

See the section on expressions for examples.

csclock

This is a predefined variable, the centisecond clock. It is simple 16 bit counter set to zero on a reset and thus reflects the number of centiseconds since the last reset. Writing a value to it will have it count upwards from that value.

table <identifier>;
table <identifier> [<numeric>,(<numeric>, ...)];

```
table convert [1,3,5,7,9,11,13,15,17,19,21,23,25];
table convert [lin,256,0,127];
table convert [ 1,2,3,4,5,6,7,8,9,ran,20,10,30,22];
table convertP;
```

Tables are data structures in the memory of the SensorLab consisting of a predefined number of consecutive bytes (8 bit numbers). The length of these tables is fixed when they are defined and cannot be altered dynamically. The first three of the examples initialize the tables. In cases where you don't need something specific in the table at the start, as in defining a midi input buffer, use a `lin,256,0,0` (see below). To access an element in a table you use a 16 bit offset between square brackets, no bounds checking is performed, you should see to it that you don't address anything beyond the defined length of the table, you may be overwriting program code for all you know! Negative offsets are allowed, if you can find a use for them. A special table **start_of_ram** has been predefined, allowing you access to the sensorlab's 32k or ram. See the appendix for the lab's memory map and internal data structures.

The last example defines a **table** *pointer*, this is a variable *only for table addresses*, be they literal 16 bit addresses you happen to know could be useful, or the name of a previously defined table. After assigning an address of a table or a 16 bit number to one of these **table** *pointers* it can be used just as if it were an ordinary **table**. Make sure to initialize a **table** *pointer* before you use it, otherwise very unpredictable things will happen.

<numeric> in the syntax definition stands for one of five things:

- A simple 8 bit number, decimal or hexadecimal ($ - prefix)
- a **lin** keyword; this word stands for **linear** and defines a linear stretch of numbers. It takes the next three numbers as parameters, first a length, then a start value and lastly an end value

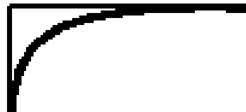
  `lin,256,0,127`

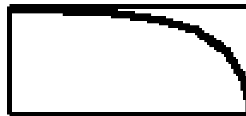
  defines 256 bytes of which the first is a 0, the last is a 127 and the 254 bytes between these two run linearly between these two.

- a **log** keyword; this word stands for **logarithmic** and defines a logarithmic stretch of numbers. It takes the next three numbers as parameters, first a length, then a start value and lastly an end value

  ```
  log,100,0,127
  ```

  defines 100 bytes of which the first is a 0, the last is a 127 and the 98 bytes between these two run steeply up at first and step up to the end number in increasingly small steps. This is useful when you're dealing with a sensor which has a logarithmic response.

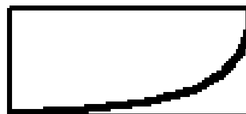
  lo to hi (log,256,0,127): 

  hi to lo (log,256,127,0): 

- an **exp** keyword; this word stands for **exponential** and defines an exponential stretch of numbers. It takes the next three numbers as parameters, first a length, then a start value and lastly an end value

  ```
  exp,1000,10,60
  ```

  defines 1000 bytes of which the first is a 10, the last is a 60 and the 998 bytes between these two start off increasing with small steps, but taking increasingly bigger steps towards the 60.

  lo to hi (exp,256,0,127): 

  hi to lo (exp,256,127,0):

- a **ran** keyword; this word stands for **random** and defines a random stretch of numbers. It takes the next three numbers as parameters, first a length, then a minimum value and lastly a maximum value;

  ```
  ran,20,10,30
  ```

  defines 20 random bytes, never lower than 10 and never higher than 30.

Some examples:

```
analog LeftHandPressure,0,0,255,2,0,0,0;
table convert [ lin,256,127,0];
LeftHandPressure.m1: // mode 1 of the analog channel entry
      prs(0,convert[LeftHandPressure]);
      end;
// send out midi pressure after molding the raw 8 bit
// sensor output into a range suitable for midi.
```

```
midi_non incomingNotes,omni;
table in_buffer[lin,100,0,0];
var index=0;

incomingNotes.m1:      // entry point for note on events
      in_buffer[index] = incomingNotes; // store note number
      index = (index + 1) % 100;        // keep index below 100
      end;
// store the last 100 notes which have come in for later
```

```
var FirstTimerCurrentMode, TimeRemaining;
table FirstTimer;
reset:
      FirstTimer = $8180;    // address of first timer struct
      end;
. . .
      FirstTimerCurrentMode = 1 + FirstTimer[1];
      TimeRemaining = FirstTimer[2] + (FirstTimer[3] << 8);
// time remaining until the first timer fires in 1/100 seconds
// remember table addressing mode only reads 8 bits, 16 bit values
// are stored low byte - high byte.
// It is a lot easier in the above example to just sa
//    TimeRemaining = <timername>;
```

```
dgroup tableSelect[0/0,0/1,0/2];
table log_tab[log,100,0,0];
table lin_tab[lin,100,0,0];
table exp_tab[exp,100,0,0];
table TheTable;

tableSelect.1.u:
      TheTable = log_tab;
      end;
tableSelect.2.u:
      TheTable = lin_tab;
      end;
```

```
tableSelect.3.u:
      TheTable = exp_tab;
      end;
// the keys select (through their up- events) one of three tables
// for use in other sections of code
```

---

analog <ident>, <channel>, <lo>,<hi>,<minimal change>,
<offset>,<gain>,<inv>;

ident

Is the name to be used in defining associated code sections and accessing the sensor output values;

ident.m1:

defines the program code, the m1 being anything from m1 to m8;

ident (and ident[0])

accesses the latest available sample; ident [1] accesses the previous sample, ident[2] ... ident[8] give power users access to analog parameters to change them on the fly:

0	latest data byte (current value)
1	previous data byte
2	current mode
3	low threshold value
4	high threshold value
5	min difference threshold
6	attenuation (see appendix)
7	offset (see appendix)
8	aux (see appendix)

<channel>

Is the physical SensorLab analog in pin this particular sensor is connected to. This runs from 0 - 31.

<lo>
<hi>
<minchange>

threshold parameters, they control when an actual ‘event’ is generated for this particular sensor and the associated code is executed. When the difference between the previously triggering sample and the current one exceeds the **minchange** value and the current value is within the lo and hi range, then an event is generated and code is executed.

The next three parameters condition the signal

into something the SensorLab can deal with. The allowed voltages on the analog input pins of the SensorLab can range from -8 to 8 volts. The analog to digital converter on the SensorLab, however, deals with signals from 0 to 5 volts only. So, for the best performance you should knead the raw input values into the 0 - 5 volts range. The

<offset>

runs from -60 to 60, in 10th's of volts, so - 6.0 to 6.0 volts; this value is added to your signal to move the signal up or down the scale. If, for example, your sensor gives you values from 2 volts at the low end and 7 volts at the top you would be able to move it into the desired 0 - 5 volts range by specifying an offset from -20 (-2 volts). The

<gain>

runs from -7 (maximum attenuation, roughly 0.17 times) to 7 (maximum amplification, 10 times); this parameter stretches or shrinks the range of your input. A positive gain expands the sensor's output signal, a negative gain contracts it. Suppose for example your sensor's output (as measured by a multimeter) is 2 volts at the low end and 3 volts at the top. This is a range of only 1 volt, to be converted into a range of 5. If you check the list below you'll see that a gain of 4 would very nearly do the trick. This will give you a range of 2 to 6.7 volts. You'll need a negative offset again to move it down the scale toward the 0 - 5 volts range.

gain:	result:
-7	0.17 x
-6	0.25 x
-5	0.32 x
-4	0.40 x
-3	0.50 x
-2	0.63 x
-1	0.78 x
0	1.00 x
1	1.50 x
2	2.20 x
3	3.30 x
4	4.70 x

5	6.80 x
6	8.20 x
7	10.00 x

<inv>

is either zero or non-zero and inverts the sensor's output when non-zero, a convenient way of changing the up/down direction of a sensor. Also it is sometimes necessary to bring a signal within range: no offset is able to bring a signal running from -3 to -8 volts to the desired 0 - 5 volt range; inversion and a negative offset of 3 volts will do the trick, however.

Every physical input channel can have several logical channels (as above) associated with it; you can for instance apply two separate triggers to on analog input by defining two analog sensors on the same channel with different **lo** and **hi** parameters:

```
analog Pressure,0,0,200,2,0,0,0;
analog Trigger,0,220,230,7,0,0,0;
table convert [lin,200,0,127];
var VERYLOUD = 127;
Pressure.ml:
    prs(0,convert[Pressure]);
    end;
Trigger.ml:
    non(0,40,VERYLOUD);
    non(0,40,0);          // switch off again immediately
    end;                  // using non to switch off notes
                          // makes use of the sensor lab's
                          // running status capability
```

This would give you a continuous pressure reading when you apply pressure to a pressure sensitive device, and also give you a trigger when pressing the device all the way home. Both are, as you can see, attached to physical channel 0, thus reading the same sensor. The minimum change of the second declaration is set at a rather high setting of 7 since we're only interested in the trigger in this example. In this example the gain, offset and inv parameters are set to zero, implying a perfect device, outputting a beautiful 0 to 5 volts across the pressure range.

usound <name>,<start>,<length>,<min_change> (, <name2>,<start2>, ...);

name.m1: (a label)
is the entry label for the SPIDER code (mode 1);
name (a variable)
accesses the latest available sample

Defines an ultrasound channel, measuring the distance between an US transmitter and an US receiver. Parameters between brackets are optional and define further 'active zones' on the same physical ultrasound channel. The order in which the ultrasound channels are defined corresponds to their physical connection to the SensorLab; the first three correspond to the three receivers listening to the first transmitter, the following three use the three receivers with the second transmitter.

The **start** point is the distance in centimeters where the ultrasound channel starts generating events, the **length** determines the length of the stretch in centimeters over which the result variable will vary from 0 to 255;

```
USOUND distance, 10,210,2;
```

This will set up an active range of two meters starting at 10 centimeters from the receiver, an event will be triggered when the difference between the current sample and the previous sample exceeds a value of 2.

This is how you would set up an ultrasound channel to control the velocity of notes generated somewhere else in the program:

```
usound rawDistance,10,110,2;
var Velocity;
rawDistance.m1:
     Velocity = rawDistance >> 1; // bring into 0 - 127 range
     end;
```

Here we set up three ‘triggers’ for program changes:

```
usound Trig1,100,110,5, Trig2,150,160,5, Trig3,200,210,5;
Trig1.m1:  // active from 100 cm to 110 cm ..
     pgc(0,10);
```

```
     end;
Trig2.ml:  // active from 150 cm to 160 cm ..
     pgc(0,20);
     end;
Trig3.ml:  // active from 200 cm to 210 cm ..
     pgc(0,30);
     end;
```

---

dgroup <groupname> [sl/sr(,sl/sr, sl/sr, ...)];

Defines a group of (digital) key sensors into a logical unit. The **<groupname>** will be used in the rest of the file in swap and scratch commands, and in the key program labels. The scanline**(sl)**/scanread**(sr)** combinations can be inferred from the physical connections to the SensorLab, where **sl** ranges from 0 to 15 and **sr** ranges from 0 to 7 (See SensorLab hardware manual).

Example:

```
DGROUP NoteKeys[1/0,1/1,1/2,1/3,1/4];
var Velocity=80;

      NoteKeys.1.d:
            NON(1,64,Velocity);
            end;
      NoteKeys.1.u:
            NOF(1,64,64);
            end;
      NoteKeys.1.m2.d:
            NON(1,76,Velocity);
            end;
      NoteKeys.1.m2.u:
            NOF(1,76,64);
            end;
```

Valid key program labels will be:

```
NoteKeys.1.u through NoteKeys.5.u  and
NoteKeys.1.d through NoteKeys.5.d
```

And, for mode 2 through 8, if used:

```
NoteKeys.1.m2.u through NoteKeys.5.m2.u  and
NoteKeys.1.m2.d through NoteKeys.5.m2.d
```

To use a key as a ‘toggle’:

```
DGROUP SustKey[3/7];
var SusTain = false;
```

```
SustKey.u:
      SusTain = SusTain ^ 127;
      ctr(0,64,SusTain);
      end;
```

When using keys as a toggle you need to define a code section only for one direction (either the **up** or the **down** event of the key); when possible use the **up** event in this situation, this reduces the chance of you making serious errors when programming **mode** changes or **execute** commands where the group **swapped** is the group which does the swapping. This is just a piece of advice, when you know what you are doing there is no reason not to use the down event for toggle functions.

timer <ident>,<resolution>;

This statement defines one of the 16 possible timers in the SensorLab, setting it up initially to fire every (**resolution** / 100) seconds. **Ident** is used, as before, as the entry label for the code which is to be regularly executed (with the **mode** extension m1 to m8).

Ident is at the same time defined as a special kind of variable, assigning a 16 bit number to it will set the resolution of the timer to the value assigned, reading from this variable will give you the time remaining before the timer will fire (in centiseconds).

As mentioned before, the timers are only as accurate as you let them be, the system only checks to see if a timer has timed out when it is not busy executing other code. So an extremely long key program can easily hold up the execution of a timer program. It is not easy to say what a ‘long’ program is, try things out to find the limits of the SensorLab.

To kill a timer, your only option is to swap it to an unused mode; all mode programs not defined by the programmer consist of a lonely **end** statement, so that nothing will happen when you **swap** something to a mode which is not defined.

```
timer Update,50;
Update.m1:
      display(0,"    ");               // clear the display
      if(Sustain) display(0,"S");
      if(Transpose) display(1,"T");
      if(Control) display(2,"C");
      if(MultiChannel) display(3,"M");
      end;
// this short program will update an attached 4 character
// display every half second to reflect the state of four
// variables
```

```
dgroup hit [5/6];
timer roll,1000;
var rolltime;

hit.1.d:
      rolltime = 128;        // used to keep track of current interval
      roll = 128;            // initialize timer to 128 centiseconds
```

```
     swap roll,1;          // make timer active
     end;

roll.m1:
     non(9,43,100);  // send one note trigger,
     non(9,43,0);    // immediately switching it off (ok for drum
sounds)
     rolltime = rolltime >> 1;   // divide time interval by two
     if(!rolltime) swap roll,8; // if interval is zero kill the timer
     roll = rolltime;            // make interval the new timer interval
     end;

reset:
     swap roll,8;    // swap the timer to an unused mode,
     end;            // effectively shutting it up
     // everything starts out in mode 1, after a reset that is why
     // you have to manually swap to a 'silent' mode if so desired
```

---

midi_non
<ident>,<channel>;
midi_nof
<ident>,<channel>;
midi_pbd
<ident>,<channel>;
midi_pgc
<ident>,<channel>;
midi_prs
<ident>,<channel>;

midi_ctr
<ident>,<ctr>,<channel>;
midi_pkp
<ident>,<key>,<channel>;

These commands define midi inputs as events, they set the SensorLab up to jump to a section of user-written code whenever the specified command is seen on the midi input port. **Ident**, as before, specifies the entry label to jump to when the event occurs, it also gives access to the data of the midi message that generated the event. The channel specifies the midi channel on which the event should occur before it triggers you program. The channels run from 0 through to 15, as everywhere in the SensorLab. In addition you can specify the word '**omni**' to indicate that every channel is of interest. The midi continuous controller and the poly key pressure definitions require you to give an extra parameter, either the controller you're interested in or the specific key for the poly key pressure messages. After the definitions the following scheme gives you access to the midi parameters:

```
ident (or ident[0])   data byte #1 (key number f.i.)
ident[1]              data byte #2 (velocity)
ident[2]              status byte  (for channel info in omni mode)
```

For midi_pgc and midi_prs the data byte #2 is not defined, will read nonsense. The data bytes are reversed in the case of ctr and pkp messages, Giving you easier access to information you don't already know. See the **thru()** command.

```
midi_non notes_in,omni;
midi_nof off_notes_in,omni;
```

```
notes_in.ml:
     non(notes_in[2] & 15, 127 - notes_in, notes_in[1]);
     end;
off_notes_in.ml:
     non(off_notes_in[2] & 15, 127 - off_notes_in,0);
     end;
```

The above is a simple program to reverse the keyboard, high notes on the left, low ones on the right. The channel is preserved, it being retrieved from the status byte at offset 2, as is the velocity. Note on events with velocity zero are used as note off events to make use of running status and reduce the consumption of midi bandwidth. It is perfectly legitimate to use the normal note- off midi message. Your choice. The use of **'& 15'** in the channel position is not necessary, but clarifies the 4 bit output limit for channel data.

```
midi_ctr breath,1,omni;

breath.ml:
     prs(breath[2] & 15,breath);
     end;
```

Change midi breath control messages into channel pressure messages, note that the value of the breath message is found at offset zero and can thus be addressed without any brackets.

```
dgroup keys[0/0,0/1,0/2,0/3];
table buffer[lin,256,0,0];
var note,index=0;

midi_non note_in,0;        // only listen to midi chan 1

note_in.ml:
     buffer[index] = notes_in;    // store incoming notes
     index = index + 1 & 255;     // keep index in bounds
     end;

keys.1.d:
     note = buffer[ random(256) ];    // select from buffer
     non(0,note,100);
     end;
```

This is a very simple minded example of how you could play notes using somebody else's material, as found on midi channel 1.

call <program label>;

Calls subroutine program at <program label>. Execution continues at <program label>, until a **return** statement is executed. Then the execution of the program will continue with the statement right after the **call** statement. Extra care should be taken that a 'subroutine' i.e. a piece of code starting with a label and ending with a return statement is only executed through a **call** statement. A **return** statement without matching **call** will definitely cause the program to crash. **Call**'s can be nested (calls to routines from within a routine which was 'called' itself, or calls to the executing routine itself - recursion -) to a depth of 128.

Example:

```
MyLabel:
      ...
      call do_noton;
      ...
      end;

do_noton:
      non(1,NoteNr,Velocity);
      return;
```

The section of code at 'MyLabel' somewhere calls the subroutine 'do_noton'. When the 'do_noton' is finished (by the **return** command), the rest of the code at ' MyLabel' will be executed.

return;

See above.

display(expr1,expr2);
display(expr1, <string>);
displayr(expr1,expr2);
displayx(expr1,expr2);
displayl(expr1,expr2);

```
display(0,"string");
display(1,myVar);
displayx(myPos,30 + myVar);
displayl(0,myVar * yourVar);
```

Display a "string" or a variable at the position expr1 on the *ascii* display. Positions will vary with the implementation; 0-31 are defined, but need to be physically connected to the SensorLab of course. **Displayx** displays the variable in hexadecimal format at position pos. Both **display** and **displayx** only display the least significant byte of a variable; since most implementations only use a four character display a decimal representation of a 16 bit value would not fit, it being a maximum of five characters long. Therefore only a *hexadecimal* 16 bit display command is available, the **displayl** command. The **displayr** command is mainly intended for users who use the display interface for other i/o purposes and need to be able to send raw values to the display interface. The 8 bit value supplied by expr2 is sent out of the display port without any interpretation, you only need it when you do.

A decimal number will always reserve three characters on the display, a hexadecimal number two. Characters will be overwritten only when that character position is accessed, so displaying ‘10’ after displaying ‘100’ would not erase the ‘1’ from the number ‘100’ and leave the display displaying ‘110’. This is intentional, smaller numbers can in such a manner be displayed without reserving 3 display positions and more efficient use can be made of the display.

Examples:

```
display(0,"OKEE");          // The display will show: 'OKEE'

Value = 56;
display(0,Value);           // The display will show: ' 56 '
displayx(2,Value);          // The display will show: '  38'
     // to allways reliable display a decimal number:
```

```
display(0,"   "); // three spaces
display(0,myNum);
```

end;

The statement to end all statements. It is the statement which terminates a user *event program*, and signals the SensorLab to stop executing the current sequence of code and look for something else to do, wait for another event for instance. If an **end** statement is omitted the SensorLab will just continue executing until it does find one. Sometimes one can make use of this (see **goto**) but in general it is an error.

execute <keygroupname>, <programlabel>;

Executes the key - up program for all the keys in <**keygroupname**> which are currently in the -down- state, as if the user actually *released* the keys. Then it executes the code at label up to the first **end** statement, and subsequently executes the key down programs for the previously 'released' keys, making the SPIDER program believe that the user pressed the keys again.

This command allows for a change in transposition or midi channel, for instance, without having to think about hanging notes because notes are switched off before the channel or transposition is changed, assuming that note off commands are programmed at the key up events of the keys in <**keygroupname**>.

Example:

```
dgroup Shift [0/0,0/1];
dgroup Keys [2/0,2/1];
var Transposition = 60, NoteNr,Velocity=100;

_transposup:                // of course this label may also
      Transposition = 72;   // have been named Transposup
      end;                  // or transup or _hopsakee, but as
_transposdown:              // a convention all labels to be
      Transposition = 60;   // jumped to from an EXECUTE, start
      end;                  // with '_' (underscore)
Shift.1.d:
      execute Keys,_transposup;
      end;
Shift.1.u:
      execute Keys,_transposdown;
      end;
Keys.4.d:
      NoteNr = Transposition + 3;
      non(1,NoteNr,Velocity);
      end;
Keys.4.u:
      NoteNr = Transposition + 3;
      non(1,NoteNr,0); end;
```

If key # 4 gets a down event, it will calculate the corresponding note number and play the note. If the first key

of group 'Shift' is down, it executes the **execute** command for the group 'Keys' to do a transpose up.

What happens is that the SensorLab will simulate key up events for the keys in the group 'Keys' that are currently in the -down- state, thus sending midi note off messages with the old value of the variable 'Transposition', then execute the code starting at the label '_Transposup', setting the new transposition value and finally simulate the key down events again, sending midi note on messages with the newly calculated transposition.

goto <label>;

Continues program execution at **<label>**. Labels can be defined anywhere in the file without restrictions and have the form **'<identifier>:'** .

Example:

```
Keys.1.d:                        // Calculate the variable 'NoteNr'
     NoteNr = 48;                // and jump to label 'note_on'
     goto note_on;               // notice that no 'end' command is
                                 // necessary now.
Keys.2.d:
     NoteNr = 49;                // 'fall through' to note_on
note_on:                         // Send midi note on messages on
     non (1,NoteNr,Velocity);    // the first three midi channels.
     non (2,NoteNr,Velocity);
     non (3,NoteNr,Velocity);
     end;
```

if (expr) <comp-stmnt1>
if (expr) <comp-stmnt1> else <comp-stmnt2>

Conditional execution; <comp-stmnt> here means any one semicolon terminated statement or a series of such between braces '{' and '}'. If the expression evaluates to a non-zero value comp_stmnt1 will be executed, if it is *false* (zero) comp-stmnt2.

Examples:

```
if(sendModulation)    // only send control information when
     ctr(0,1,CurrentModulation);  // 'sendModulation' is nonzero
end;
```

```
if(MyVal > 25)
     non(1,60,MyVal);  // If variable 'MyVal' is bigger than 25
else                   // send the midi note on event, if less than
     non(1,60,0);      // or equal 25, send the midi note off
                       // event.
```

```
if(program == 10) {
     program = 11;
     pgc(0,11);
}
else if (program == 11) {
     program = 12;
     pgc(0,12);
}
else {
     program = 13;
     pgc(0,13);
}
```

In the last example, once a condition is met, no other checks are made and execution continues after the last curly brace.

led(expr,expr);

This switches one of the four the led outputs of the SensorLab, the first expression evaluates to the led number, and should thus be in the range 0 - 3, and the second expr is read as a boolean, non-zero switches the led on, zero switches the led off.

Midi output commands

non (expr,expr,expr);

Sends a midi note on message, expressions are channel, note number and velocity. The channel is limited to the numbers 0 - 15, corresponding to midi channels 1 to 16. The other two expressions are limited to the range 0 - 127.

nof (expr,expr,expr);

Sends a midi note off message, expressions are channel, note number and release velocity. See **non**.

ctr (expr,expr,expr);

Sends a midi continuous controller message, expressions are channel, controller number and controller value. The channel is limited to the numbers 0 - 15, corresponding to midi channels 1 to 16. The other two expressions are limited to the range 0 - 127.

pbd (expr,expr,expr);

Sends a midi pitch bend message, expressions are channel, low value and high value. The low value is ignored by most midi devices and should generally be set to 64 (the middle value), the high value runs from 0 - 127 and has a ‘no pitch bend’, a centre value of 64.

pkp (expr,expr,expr);

Sends a midi poly key pressure message, not much used, but there if you need it. Parameters are channel, key number and value.

pgc (expr,expr);

midi program change. Parameters are channel 0-15 and program 0-127.

prs (expr,expr);

midi channel pressure, channel and pressure value are the parameters.

sysex (expr,expr,...);

Sends out a system exclusive message, all values except the first and last need to be in the 0 to 127 range. The sysex header (`$f0`) and tail (`$f7`) are **not** added by the SensorLab, and should be included by the user.

thru(expr);

This command switches between *midi thru* mode (when expr evaluates to a non-zero value) and *midi interpret* mode (when expr is zero). In the *thru* mode everything the SensorLab finds on its input will be sent out unchanged. In *interpret* mode the SensorLab will, on seeing a midi message coming in, check to see if there are any declarations for this midi message. If there are they will be dealt with, otherwise the message finds its end in the SensorLab and nothing happens.

random(expr)

Evaluates the expr and then returns a random number in the range 0 - (expr - 1).

The random number generator has a sequence length of 65536, this means that after calling random 65536 times it will start generating the same sequence again. See also rseed.

Example:

```
MyValue = random(43);        // Variable 'MyValue' now will be
                             // in the range 0 to 42.
// to obtain a random value between lo and  hi:

MyValue = lo + random(hi - lo);
```

rseed(expr);

The rseed command primes the random generator, to get a sequence of random numbers several times (that is: the same sequence) use rseed with the same value; what this value is is not important, it only matters that the value is the same as the last time. At reset the SensorLab seeds the random generator with the value $AAAA.

```
var value;
label:
      rseed(1001);
      value = random(10);   // result, say, 7
      value = random(10);   // result, say, 2
      rseed(1001);
      value = random(10);   // result again 7
      value = random(10);   // result again 2
      value = random(10);   // result, say, 3
      rseed(1001);
      value = random(10);   // result again 7
```

scratch <keygroup>;

```
dgroup keys[0/0,0/1,0/2,0/3];
scratch keys;
```

Scratch is a command for **dgroups** only. It is fully equivalent to a **swap** to the same mode as the group is already in. Because of the simulated key up and -down events this can be useful. It would, if the keygroup **scratched** was actually a keygroup sending notes, have the effect of retriggering all notes for which the corresponding keys have been depressed. Coupled to a change in distance from an ultrasound event or a change in pressure from an analog event, one can do the retriggering very fast, resulting in sometimes interesting effects. This gets an extra dimension when one changes something like the velocity of the notes one is retriggering.

```
dgroup NoteKeys[0/0,0/1,0/2,0/3];
usound UltraSound,2,100,2;
table VelocityConversion[lin,256,1,127];
var Velocity=1;

UltraSound.m1:
      Velocity = VelocityConversion[UltraSound]; // keep between 1 and 127
      scratch NoteKeys;
      end;
NoteKeys.1.d:    // note on key 1
      non(0,40,Velocity);
      end;
NoteKeys.1.u:    // note off key 1
      non(0,40,0);
      end;
 --  etc --
```

The above switches note on and off in response to an ultrasound event, all the time adjusting velocity.

swap <identifier>,<expr>;

This command swaps any of the defined ‘inputs’ to one of the eight *modes*, see the section on *modes* earlier. The swappable things are:

usound,timer,analog,dgroup and midi_in

They are swapped by giving their identifier to the swap command and specifying the mode, ranging from 1 to 8. **Dgroup** swaps perform the release- and close simulation as discussed under **execute**.

```
dgroup channel[0/0];
dgroup notes[1/0,1/1];

shift.1.d:
      swap notes,2;
      end;
shift.1.u:
      swap notes,1;
      end;

notes.1.d:             // for keys mode 1 doesn't need to be supplied
                       // but notes.1.m1.d would be fine
      non(0,40,100); end;   // channel 1 note on
notes.1.u:
      non(0,40,0); end;     // channel 1 note off
notes.2.d:
      non(0,43,100); end;   // channel 1 note on
notes.2.u:
      non(0,43,0); end;     // channel 1 note off

notes.1.m2.d:
      non(1,40,100); end;   // channel 2 note on
notes.1.m2.u:
      non(1,40,0); end;     // channel 2 note off
```

The shift key, when pressed makes the note keys transmit on channel 2, rather than channel 1. When you press the shift key with one or two note keys pressed as well, the corresponding note off events will be sent on the proper channels, before switching midi channel.

while(expr) <comp-stmnt1>

Executes **<comp-stmnt1>** (see above under **if**)

repeatedly until expr becomes **false**. This implies that somewhere in **<comp-stmnt1>** something should happen that influences the **expr**, otherwise you would end up with a loop which would never exit and the SensorLab would never execute anything else until a reset (it would ‘hang’).

```
var index;
table buffer[lin,256,1,1]; // fill table with 1's
reset:
     index = 0;
     while(index < 256) {  // clear buffer
          buffer[index] = 0;
          index = index + 1;
     }
     end;
```

Memory map & data structures

The SensorLab has 32k of rom, which is not accessible by the user, and 32k of ram, running from address $8000 to address $FFFF, in which system parameters, data structures and the user code are held. The ram as a whole is addressable through the predefined **table** 'start_of_ram'. Use assignments to elements of this **table** only if you know exactly what you're doing. In the monitor window there are two commands which allow you to inspect the memory; the first, **inspect**, allows you to check the SPIDER internal image as it has been built by the last compile command. Just type **inspect**, a space and a hexadecimal address. You will be presented with 256 bytes from the internal image in the monitor window. The second command, **dump**, requires a two way midi connection between the SensorLab and your mac, has the same syntax as the **inspect** command, but shows you a page (256 bytes) from the SensorLab's internal ram. This is continuously updated to reflect changes in the ram. To exit this mode just click the mouse. Try looking at address $8000 (the 'system page') to see the centisecond clock running.

Memory map

$8000 system page, system parameters, address offsets to data structures and memory validity check area.
$8100 analog to digital data structure offsets, 64 addresses of analog data structures, maximum.
$8180 timer data structures, a maximum of 16 timer data structures.
$8200 midi in definition data structures, a maximum of 32 data structures describing the midi in subsystem.
$8300 keygroup current modes, 128 numbers reflecting the modes the keygroups are currenly in, the keygroup offsets correspond to the order in which they are defined.
$8500 subroutine & swap stack
$8600 SensorLab 'virtual machine' stack.
$8700 keybits - reflect state of the digital keys
$8A00 midi - in message flags
$8B00 user area (later than july 1993)
$9B00 user area (before august 1993)
$F700 midi buffers

Data structures

analog data (pointed to by addresses at page $8100)

offset	content
0	physical channel (0-31)
1	latest data byte
2	previous data byte
3	current mode
4	low threshold value
5	high threshold value
6	event minimal change threshold
7	attenuation
8	offset
9	aux
10	mode 1 routine address
12	mode 2 routine address
14	mode 3 routine address
16	mode 4 routine address
18	mode 5 routine address
20	mode 6 routine address
22	mode 7 routine address
24	mode 8 routine address

The analog data structure is compiled in the user code area, the analog *identifier,* as defined in the **analog** command, point to offset no. Attenuation/Gain is calculated by SPIDER through the following scheme:
Negative numbers are an attenuation, multiply by -1 and use the result as an index in the following array :
255,228,203,181,162,144,128,114
This gives you the attenuation byte the lowest three bits of the *aux* are cleared..
If the gain/att is positive it is simply copied into the lowest three bits of the *aux* byte, the attenuation byte is set to 255 in this case.

If the offset is negative bit 6 of *aux* is cleared, if it is positive bit 6 of *aux* is set. The sign is then dropped. If the resulting number is greater than 30 bit 5 of *aux* is set and the (by now unsigned) offset is divided by two. If the unsigned offset was smaller than or equal to 30 bit 5 of *aux* is cleared. This leaves us with a possible 31 different numbers which are then translated through the next array to arrive at the number which is finally stored in the *offset* byte of the above data structure.

0,48,65,81,93,104,114,124,132,141,148,
154,162,173,175,181,185,192,200,204,208,
220,239,242,243,246,248,250,252,255

timer data

The **timer** data structures are found from address $8180, in the order in which they were defined. A write to the **timer** *identifier* actually fills two consecutive addresses with the value written: those at offset 2 and at offset 4.

offset	content
0	subcount
1	current mode
2	16 bit current count value
4	16 bit reload value
6	address of jump block

The jump block address points to eight addresses of the eight routines for the **timer** in the eight different *modes*.

midi in data

These data structures are found at address $8200, again in the order in which they were defined.

offset	content
0	data valid
1	data to wait for (pkp & ctr)
2	current mode
3	data byte no 1
4	data byte no 2
5	status byte
6	address of jump block

The jump block is the same as for the **timer** data structure. At address $8A00 the first 112 bytes flag the existence of a defined *midi-input*. If the highest bit of the byte at <status / 16> is set, the remaining seven bits form the number of the midi input definition structure at page $82.

If bit seven is not set, no *input* is defined for this particular status byte.

usound data

The ultrasound data is compiled into the code section, the six possible data structures are pointed to by addresses which can be found from address $8020; first three for transmitter 1 and then three for transmitter 2.

offset	content
0	new distance high/low
2	previous distance high/low
4	command byte
5	zero cross point high/low
7	divide/multiply factor high/low
9	previously triggering value (for filter)
11	filter (minchange)
12	mode
13	address of jump block
15	current value high/low (pointed to by name)

If there is more than one definition for this ultrsound channel then bytes 4 through 15 are repeated. The high bit of the command byte is set if this is the last definition for this ultrasound channel. The values are determined as follows:

zero = 120 + 30 * start;
div/mul = divmul = (10 * length) / 85; (if length > 250)
divmul = 65280 / (30 * cmm); (if length <= 250)
if length > 250 bit 1 of the command byte is set.

SensorLab opcodes

Internally the SensorLab works with tokens, implementing a sort of virtual machine. The ‘machine works with a stack and two ‘registers’. The register names are **a** and **b**. All SPIDER code is translated into these tokens which are then interpreted by a SensorLab ‘virtual machine’. Follow the opcodes, where Ctl and Cth are the low respectively high bytes of a constant; Hg and Lw are the high and low bytes of an address and b16 and b8 indicate the witdh of a data transfer. All numbers are hexadecimal.

Move commands..	coding
• move constant to a	10 Ctl Cth b16
• move constant to b	11 Ctl Cth b16
• move indirect to a	12 Hg Lw b16
• move indirect to b	13 Hg Lw b16
• move a to address	14 Hg Lw b16
• move b to address	15 Hg Lw b16
• move b^th element at address to a	16 Hg Lw b8
• move a to b^th element at address	17 Hg Lw b8
• swap a and b	18 b16
• push a	19 b16
• push b	1A b16
• pop a	1B b16
• pop b	1C b16
• move b^th from address at address to a	1D Hg Lw b8
• move a to b^th at address at address	1E Hg Lw b8

- push a 1F b8 (for i/o)

Arithmetical commands

- multiply a and b, result in a 20
- divide a by b, result in a 21
- take remainder of a div b, result in a 22

- sub b from a 23
- add b to a 24

- bitwise and a & b, result in a 25
- bitwise or a & b, result in a 26
- bitwise xor a & b, result in a 27

- shift left a, b places 28
- shift right a, b places 29

- maximum from a and b in a 2A
- minimum from a and b in a 2B
- random 0 -- <a> 2C
- leaves 1's cpl from a in a 2D
- leaves 2's cpl from a in a 2E

Conditional & logic commands

These commands all leave a boolean result in a; if the condition holds *true*, -1 ($FF) will be left in accu a, if the proposition is *false* a will be cleared.

- a is smaller than b 30
- a is smaller than or equal to b 31
- a is equal to b 32
- a is not equal to b 33

- a and b are true (nonzero) 34
- a or b or both are true 35

- negate a (-1 -> 0; 0 -> -1) 36

- a changed? 38

Program flow and control commands

- jump if a false 41 Hg Lw
- jump 43 Hg Lw
- jsr 44 Hg Lw
- rts 45

- swap keygroup #b to a 4A
- swap keygroup #a & execute code 4B Hg Lw
- scratch keygroup #a 4C

Midi commands

midi commands take their parameters off the stack in the following order: second, first and channel when it is a three byte message; data byte and channel in the case of a two byte message.

- send note off 50
- send note on 51
- send poly key pressure 52
- send controller message 53
- send program change 54
- send channel pressure 55
- send pitch bend message 56
- send single byte out the midi port 57
- midi_thru on/off (state in a) 5F

Display commands

Display commands take a parameter from a, then a display position from b.

- display in hex format 60
- display in decimal format 61
- display 8 bit raw format 62
- display boolean value 63
- switch on led (led # in b, state in a) 64
- display string 66 Ct Ct Ct ... FF
- display 16 bit hex pos in b val in a 67

Example:

The Midi Conductor, description and program

Description:

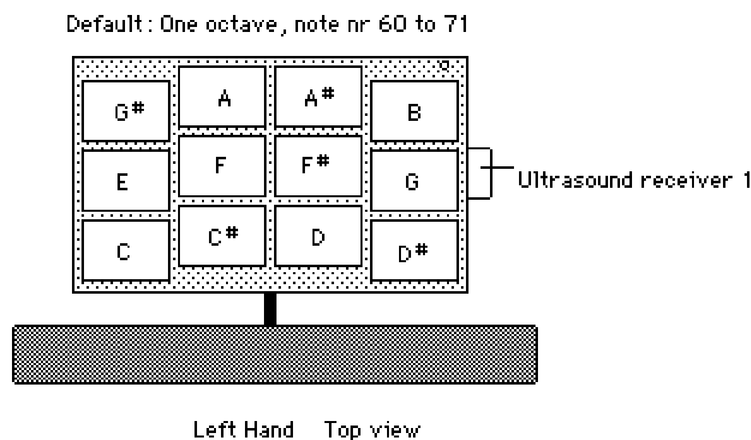
The Midi Conductor (TMC) is an electronic instrument, designed by Michel Waisvisz, which is capable of transmitting midi to enable control of music synthesizers. TMC consists of three parts, the instrument itself comprising a left hand and right hand part, and the SensorLab.

The midi implementation of TMC:
- more than ten octaves pitch range
- velocity sensitive
- modulation control
- breath control
- volume control
- panning control
- sustain control
- aftertouch (channel pressure)
- pitch bend
- program changes

In the standard setup all midi is sent on midi channel 1, and in a special mode it will send on midi channel 16.

The Instrument:

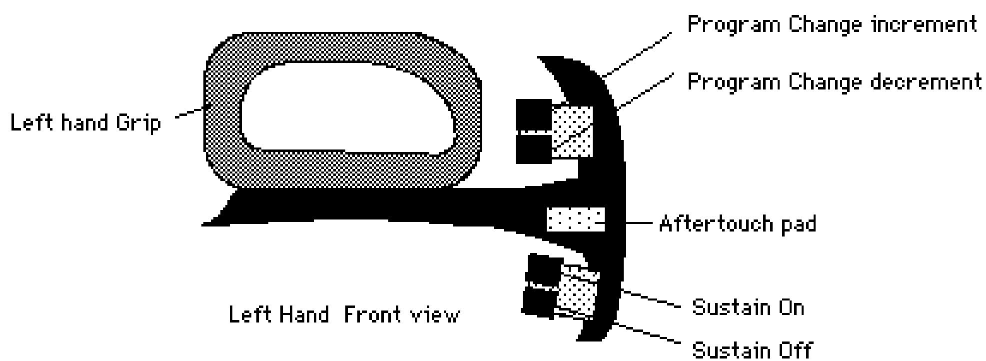
Left hand part:



Left Hand Top view

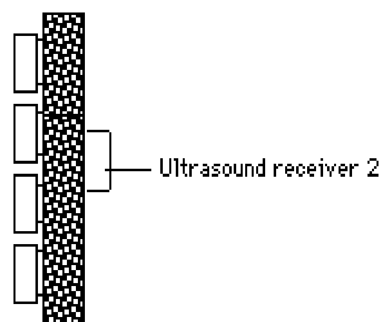
12 switches, representing one octave. Pressing a switch generates a midi note-on event, releasing the switch generates a midi note-off event. The velocity information is

derived from the ultrasound receiver 1 (horizontal), which measures the distance between the left and right hand. The greater the distance, the higher the velocity.



The performer's left hand thumb can generate midi aftertouch events by pressing the *Aftertouch pad*, send midi program change events by clicking the *Program Change* switches and control midi sustain by clicking either the *Sustain On*, or the *Sustain Off* switch.

The *Left hand Grip* is removable, so individual grips can be made to fit the player's hand size.

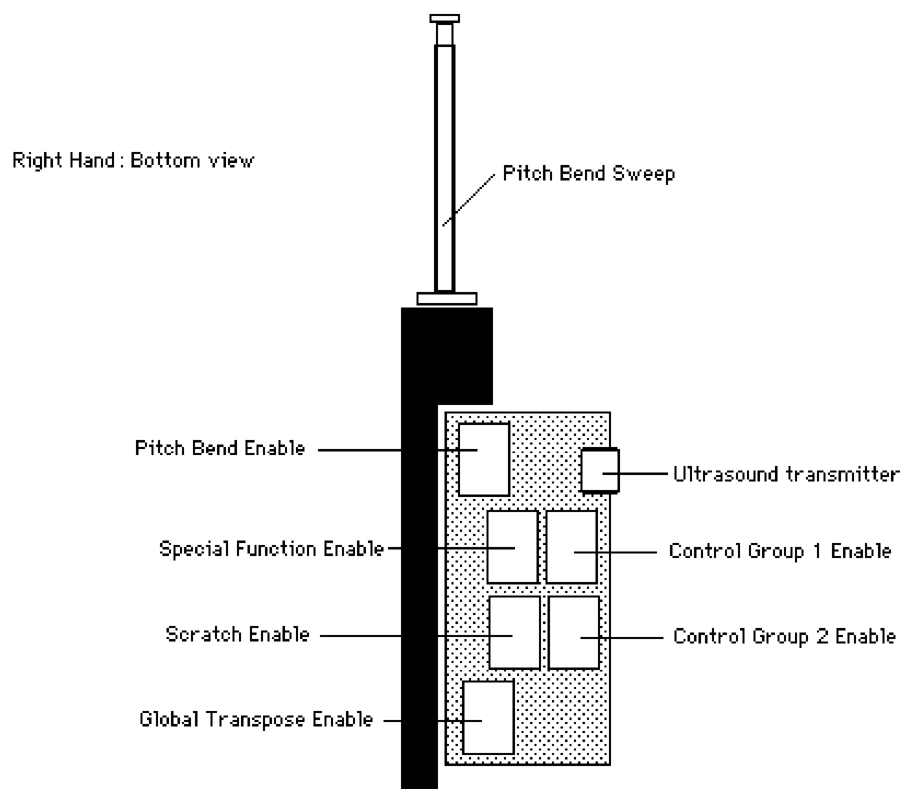


Left hand keypad, vertical position

The left hand part of TMC also contains a second ultrasound receiver, which is active when you hold this hand in vertical

position. Its functions depend upon which switch is pressed on the right hand part of TMC.

Right hand part:



All the switches on the right hand part act as shift

switches: as long as a switch is held down, its function is asserted.

When no right hand switch is pressed, the only function of the right hand part of TMC is to provide the ultrasound signal for the ultrasound receiver 1 on the left hand part. The transmitter and the receiver must be directed towards each other, and the distance between them determines the midi velocity value to be used whenever a midi note event is generated.

The *Pitch Bend Enable* switch transmits midi pitch bend messages. The amount of pitch bend is controlled by the angle the *Sweep* makes with the right hand grip. The greater the angle, the higher the pitch bend value. The *Sweep* is actually a telescoping antenna which can be changed in length and thus will change the sweep frequency.

The *Special Function Enable* switch will transpose the midi note events generated by the left hand keypad by one octave (so note numbers 72 to 83) and send those events on midi channel 16. These events can be used as special events to control a dedicated computer program for example.

The *Scratch Enable* switch activates a function not found on most other midi transmitting instruments. Whenever one or more left hand keypad switches are pressed (so midi note-on events are generated), and the distance between the ultrasound transmitter and the ultrasound receiver 1 changes, those notes will be retriggered with their new velocity values. This allows you to generate notes very rapidly by changing continuously the distance between left and right hand.

The *Global Transpose Enable* switch allows you to select a new range for the midi note events as generated by the left hand keypad. It actually functions as a transposition controller. When the ultrasound transmitter and the ultrasound receiver 2 (vertical receiver) are directed to each other, the distance between them sets the midi note number that the left hand keypad switch marked 'C' will generate. The smaller the distance, the lower the note number. The note number ranges from 0 to 127. The switches in the keypad will still maintain their chromatic order, so if the switch marked 'C' generates note number 23, then 'C#' = 24, 'D' = 25, etc.

When the left hand part is in horizontal position again, the new octave range will still be active as long as the *Global*

Transpose Enable switch is pressed.

The *Control Group 1 Enable* will allow midi modulation control messages to be sent by the ultrasound receiver 1 and midi breath control messages sent by ultrasound receiver 2. The smaller the distance, the smaller the value. The ultrasound receiver 1 still functions to determine the note event's velocity values.

The *Control Group 2 Enable* will allow midi panning control messages to be sent by the ultrasound receiver 1 and midi volume control messages via ultrasound receiver 2.

The smaller the distance, the smaller the value. The ultrasound receiver 1 still functions to determine the note event's velocity values.

If more than one of the enable switches is pressed, the corresponding functions will still work, so you may scratch while sending modulation and panning control messages all at the same time.

This covers the functional description of The Midi Conductor, following now is the SPIDER program text.

```
// 'Spider' file to be used with the Midi Conductor
// (c) 1991 by Frank Balde'

dgroup Notes[0/0,0/1,0/2,0/3,0/4,0/5,       // the twelve note keys
             0/6,0/7,1/0,1/1,1/2,1/3];
dgroup ThumbLeft[1/7,1/6,1/5,1/4];          // the left hand thumb keys
dgroup Shift[2/0,2/1,2/2,2/3,2/4,2/5];      // the right hand shift keys

analog Rawpress,0,0,255,3,10,6,0;           // the pressure pad sensor
analog Rawpb,1,0,255,3,-20,3,1;             // the right hand sweep sensor

usound VerUS,15,75,3;                       // vertical ultrasound sensor
usound HorUS,15,75,3;                       // horizontal usound sensor

table lintab [lin,256,0,127];               // 7-bits table for cont. control
table veltab [lin,256,1,127];               // velocity table (no zero!)
table pbdtab [lin,256,64,127];              // pitch bend, only upwards
table trstab [lin,256,0,116];               // pitch transpose offset table
table invtab [lin,256,127,0];               // inverse table for prs. sensor

var Pressure,Velocity = 1;                  // some variables
var NoteNr;
var Preset;
var PitchBend;
var Control,VerControl;
var NewTrans,Transposition = 60;
var ScratchFlag = FALSE;                    // some flags
var ModBrtFLag = TRUE;
var PanVolFlag = FALSE;
var TransPoseFlag = FALSE;
var PitchBendFlag = FALSE;
var Ch1 = TRUE;
```

```
// --------------------------- cont. sensors --------------------------------

RawPress.ml:                          // handle raw pressure pad data
     if(Ch1)                          // if flag 'Ch1' is TRUE, send it
          prs(0,invtab[RawPress]);    // as aftertouch on midi chan. 1,
     else                             // else send it on chan. 16
          prs(15,invtab[RawPress]);
     end;



HorUs.ml:                             // handle horizontal ultrasound
     Velocity = veltab[HorUS];        // calculate the var 'Velocity',
                                      // to be used by the 'Notes' keys
     if(ScratchFlag && Ch1)           // if this flag and the Ch1 flag
          scratch Notes;              // is TRUE, scratch the notes

     if(ModBrtFlag)                   // if the right hand shift key
for
          {                           // modulation and breath control
          if(Ch1)                     // is down, the flag 'ModBrtFlag'
               ctr(0,1,lintab[HorUS]);     // is TRUE, so also send midi
mod.
          else                        // events on either chan. 1 or 16
               ctr(15,1,lintab[HorUS]);
          }

     if(PanVolFlag)                   // same as above, but now for
          {                           // panning/volume control
          if(Ch1)
               ctr(0,10,lintab[HorUS]);
          else
               ctr(15,10,lintab[HorUS]);
           }

     end;
```

```
 VerUS.m1:                                  // handle the vertical ultrasound
     if(ModBrtFlag)                         // if necessary, send midi breath
          {                                 // control on either chan. 1 or
16
          if(Ch1)
               ctr(0,2,lintab[VerUS]);
          else
               ctr(15,2,lintab[VerUS]);
          }

     if(PanVolFlag)                         // same for midi volume
          {
          if(Ch1)
               ctr(0,7,lintab[VerUS]);
          else
               ctr(15,7,lintab[VerUS]);
          }

     if(TransPoseFlag && Ch1)               // if 'TransposeFlag' is TRUE,
and
          {                                 // if playing on chan. 1, calc.
          NewTrans = trstab[VerUS];         // the transpose offset, and do
an
                                            // 'execute', to make sure we
have
          execute Notes,_transpose;         // no hanging notes. _transpose
is
          }                                 // the label of the code that
does
                                            // the actual transposing, see
     end;                                   // below...

RawPB.m1:
     {
     if(PitchBendFlag)
          {
          if(Ch1)
               pbd(0,64,pbdtab[RawPB]);
          else
               pbd(16,64,pbdtab[RawPB]);
          }

     end;
```

```
// --------------------- 'execute' routines ----------------------------------

_transpose:                                  // the actual transposition is
      Transposition = NewTrans;              // done here, because now all the
      end;                                   // notes are 'off', and right
                                             // after this 'on' again, so no
                                             // hanging notes

// ----------------------------- dgroup Notes----------------------------------
// ----------------------------- mode 1, down ---------------------------------

noton:                                       // the general midi note-on
      non(0,NoteNr,Velocity);                // command, 'Velocity' comes
      end;                                   // from the hor. usound

Notes.1.d:                                   // for each of the keys of group
      NoteNr = Transposition;                // 'Notes', calculate the pitch
      goto noton;                            // it should play when down...
Notes.2.d:                                   // the 'Transposition' is either
      NoteNr = Transposition + 1;            // 60, or comes from the vert.
      goto noton;                            // ultrasound
Notes.3.d:
      NoteNr = Transposition + 2;
      goto noton;
Notes.4.d:
      NoteNr = Transposition + 3;
      goto noton;
Notes.5.d:
      NoteNr = Transposition + 4;
      goto noton;
Notes.6.d:
      NoteNr = Transposition + 5;
      goto noton;
Notes.7.d:
      NoteNr = Transposition + 6;
      goto noton;
Notes.8.d:
      NoteNr = Transposition + 7;
      goto noton;

Notes.9.d:
      NoteNr = Transposition + 8;
```

```
      goto noton;
Notes.10.d:
      NoteNr = Transposition + 9;
      goto noton;
Notes.11.d:
      NoteNr = Transposition + 10;
      goto noton;
Notes.12.d:
      NoteNr = Transposition + 11;
      goto noton;

// ------------------------------- mode 1, up -------------------------------

notoff:                                  // the general midi chan. 1
      nof(0,NoteNr,Velocity);            // note-off command
      end;

Notes.1.u:                               // when a key from group 'Notes'
      NoteNr = Transposition;            // changes from 'down' to 'up',
      goto notoff;                       // send the corresponding note-
off
Notes.2.u:                               // event
      NoteNr = Transposition + 1;
      goto notoff;
Notes.3.u:
      NoteNr = Transposition + 2;
      goto notoff;
Notes.4.u:
      NoteNr = Transposition + 3;
      goto notoff;
Notes.5.u:
      NoteNr = Transposition + 4;
      goto notoff;
Notes.6.u:
      NoteNr = Transposition + 5;
      goto notoff;
Notes.7.u:
      NoteNr = Transposition + 6;
      goto notoff;

Notes.8.u:
      NoteNr = Transposition + 7;
      goto notoff;
Notes.9.u:
```

```
      NoteNr = Transposition + 8;
      goto notoff;
Notes.10.u:
      NoteNr = Transposition + 9;
      goto notoff;
Notes.11.u:
      NoteNr = Transposition + 10;
      goto notoff;
Notes.12.u:
      NoteNr = Transposition + 11;
      goto notoff;

// ---------------------------- dgroup Notes ----------------------------------
// ---------------------------- mode 2, down ----------------------------------

Notes.1.m2.d:                               // in 'Mode' 2, all the keys from
      non(15,72,Velocity);                  // group 'Notes' send their note
      end;                                  // events on chan. 16...
Notes.2.m2.d:                               // since in this mode, no global
      non(15,73,Velocity);                  // transpose is done, we can
enter
      end;                                  // the 'hard' note numbers
Notes.3.m2.d:
      non(15,74,Velocity);
      end;
Notes.4.m2.d:
      non(15,75,Velocity);
      end;
Notes.5.m2.d:
      non(15,76,Velocity);
      end;
Notes.6.m2.d:
      non(15,77,Velocity);
      end;


Notes.7.m2.d:
      non(15,78,Velocity);
      end;
Notes.8.m2.d:
      non(15,79,Velocity);
      end;
Notes.9.m2.d:
      non(15,80,Velocity);
```

```
     end;
Notes.10.m2.d:
     non(15,81,Velocity);
     end;
Notes.11.m2.d:
     non(15,82,Velocity);
     end;
Notes.12.m2.d:
     non(15,83,Velocity);
     end;

// ------------------------- Mode 1, up -------------------------------------

Notes.1.m2.u:                          // the up events for 'Notes' in
     nof(15,72,Velocity);              // 'Mode' 2, note-off events on
     end;                              // chan. 16
Notes.2.m2.u:
     nof(15,73,Velocity);
     end;
Notes.3.m2.u:
     nof(15,74,Velocity);
     end;
Notes.4.m2.u:
     nof(15,75,Velocity);
     end;
Notes.5.m2.u:
     nof(15,76,Velocity);
     end;
Notes.6.m2.u:
     nof(15,77,Velocity);
     end;

Notes.7.m2.u:
     nof(15,78,Velocity);
     end;
Notes.8.m2.u:
     nof(15,79,Velocity);
     end;
Notes.9.m2.u:
     nof(15,80,Velocity);
     end;
Notes.10.m2.u:
     nof(15,81,Velocity);
     end;
```

```
Notes.11.m2.u:
      nof(15,82,Velocity);
      end;
Notes.12.m2.u:
      nof(15,83,Velocity);
      end;

// ----------------------------- dgroup ThumbLeft -----------------------------
// ----------------------------- Mode 1, down ---------------------------------

// these keys have in common, that they don't use the 'up' event part

ThumbLeft.1.d:                          // when this key is down, it
      Preset = (Preset + 1) & 127;      // increments the 'Preset'
                                        // number and sends it on the
                                        // active midi channel (1 or 16)
                                        // if 'Preset' is bigger than
                                        // 127, it becomes 0 again
      if(Ch1)
            pgc(0,Preset);              // send the midi program change
      else
            pgc(15,Preset);

      end;




ThumbLeft.2.d:                          // same as above, but now it
      Preset = (Preset - 1) & 127;      // decrements the preset

      if(Ch1)
            pgc(0,Preset);
      else
            pgc(15,Preset);
      end;

ThumbLeft.3.d:                          // send a midi damper pedal-down
      if(Ch1)                           // event (sustain on)
            ctr(0,64,127);
      else
            ctr(15,64,127);
      end;
```

```
ThumbLeft.4.d:                              // send a midi damper pedal-up
      if(Ch1)                               // event (sustain off)
            ctr(0,64,0);
      else
            ctr(15,64,0);
      end;

// ------------------------------ DGroup Shift --------------------------------
// ------------------------------ Mode 1, down --------------------------------

// these keys have in common that they activate a Midi Conductor function
// only as long as they are held down, so they act like shift keys

Shift.1.d:                                  // allow pitch bend by the
'sweep'
      PitchBendFlag = TRUE;                 // sensor
      end;

Shift.2.d:                                  // hor. usound changes will now
      ModBrtFlag = TRUE;                    // also send midi mod. , vert.
      end;                                  // usound sends midi breath




Shift.3.d:                                  // from now on all midi events
      Ch1 = FALSE;                          // will be send on chan. 16.
      swap Notes,2;                         // use the 'swap' command to
      end;                                  // avoid hanging notes

Shift.4.d:                                  // allow hor. and vert. usounds
      PanVolFlag = TRUE;                    // to send midi panning and
      end;                                  // volume respectively

Shift.5.d:                                  // allow notes to be 'scratched'
      ScratchFlag = TRUE;
      end;

Shift.6.d:                                  // vert. usound can be used now
      TransPoseFlag = TRUE;                 // for global pitch transpose
      end;

// ------------------------------ Mode 1, up ----------------------------------
```

```
Shift.1.u:                                  // disable pitch bend
      PitchBendFlag = FALSE;
      end;

Shift.2.u:                                  // disable modulation/breath
      ModBrtFlag = FALSE;
      end;

Shift.3.u:                                  // back to midi chan. 1 again,
use
      Ch1 = TRUE;                           // 'swap' for 'Notes' to avoid
      swap Notes,1;                         // hanging notes on chan. 16
      end;

Shift.4.u:                                  // disable panning/volume
      PanVolFlag = FALSE;
      end;

Shift.5.u:                                  // no more scratching
      ScratchFlag = FALSE;
      end;

Shift.6.u:                                  // restore default transposition,
      TransPoseFlag = FALSE;                // 'Notes' generate midi note
      NewTrans = 60;                        // events in the range 60-71
      execute Notes,_transpose;             // (one octave)
      end;


// ------------------------- reset routine ------------------------------------


reset:                                      // make sure that when the Sensor
      Preset = 0;                           // Lab is switched on, the Midi
      Ch1 = TRUE;                           // Conductor enters a defined
      Velocity = 1;                         // state by setting and resetting
      TransPosition = 60;                   // some flags and initializing
      ModBrtFlag = FALSE;                   // some variables
      PanVolFlag = FALSE;
      ScratchFlag = FALSE;
      TransPoseFlag = FALSE;
      PitchBendFlag = FALSE;

      swap Notes,1;                         // make sure we're in 'Mode' 1
      end;
```