Western Scale Co. Limited

Programmer's Manual

SMART1 Programmer, Version 1.10

Western Scale Co. Limited
1670 Kingsway Avenue
Port Coquitlam, B.C.
V3C 3Y9, CANADA

Phone (604) 941-3474 Fax (604) 941-4020 http://www.wescale.com

Foreword

The SMART1[(1)] is a freely programmable multi purpose device manufactured by Western Scale. It was designed to perform a number of duties not only in the weighing industry but also in many other areas of process control, data management and serial communication.

Aside from an introduction to the device's components and an overview of their respective functions, this handbook provides us with all the necessary information for installation and operation of the SMART1 Programmer, the software used to create programs for the SMART1. It also describes to us the structure of the PLAIN[(2)] programming language which was developed for the SMART1, and it gives us detailed facts for each of the individual program instructions.

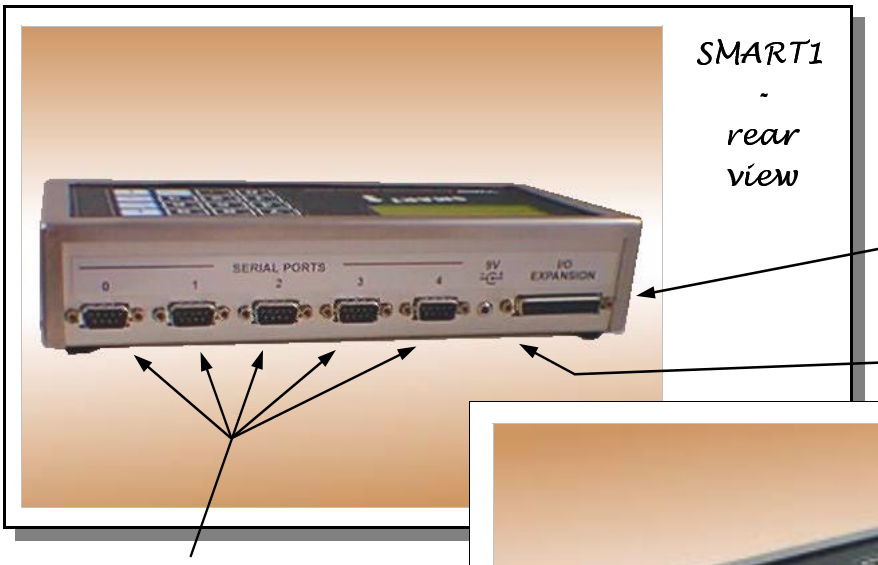
Like the programming language itself, the manual was not written with only the seasoned programmer in mind. Also the people among us who are technical personnel with little or no programming background but possess a solid knowledge of (scale)technical principles along with a good understanding of logical processes, will find this brochure a useful guide to making sophisticated programs for an unlimited variety of applications.

The SMART1 Programmer features a graphical user interface. It allows us to build complex programs in a relatively simple manner by placing and connecting images that represent commands or functions. Using the SMART1 Programmer's basic instructions set in addition to an assortment of pre-made program modules from Western Scale, we will be able to accomplish almost any task.

One of the main objectives in developing the SMART1 Programmer was to attain universal functionality. After the reading of this manual and a little bit of practice we will be able to solve most problems in a straight forward approach. Under specific circumstances, though, we might find ourselves in a situation where a round-about way seems to be the only solution to achieve what we want the SMART1 to do. That's okay, not everything in life is simple and we will be so much more proud of ourselves for making it work anyway.

The SMART1 Programmer is by no means an absolutely perfect tool. Despite of the extensive error checking capabilities built into the software, the user will still be able to make misteaks. The program might also lack one or the other slick and trendy "bell" or "whistle", but for all of us who are equipped with the proper motivation and are not easily discouraged by a little less convenience, there are virtually no limits to what we can do with the SMART1.

[(1)] **SMART1** ***S**cale **M**icroprocessor **A**nd **R**emote **T**erminal, version **1***
[(2)] **PLAIN** ***P**rogrammable **L**ogically **A**ligned **I**nterconnected **N**odes*



SMART1 - rear view

I/O Expansion Connector

Power Jack

5 Serial Communication Ports

Programming Switch

SMART1 - side view

Liquid Crystal Display

16 Buttons Membrane Keypad

SMART1 - front view

Table of Contents

CHAPTER 1 - GETTING STARTED

CHAPTER 2 - SMART1 INSTRUCTION TYPES

Instructions for Program Flow Management

Instructions for I/O Components

Instructions for System Functions

CHAPTER 3 - TO ROUND IT ALL UP

The Software Shell, second part

Tips and Tricks, Dos and Don’ts

APPENDIX

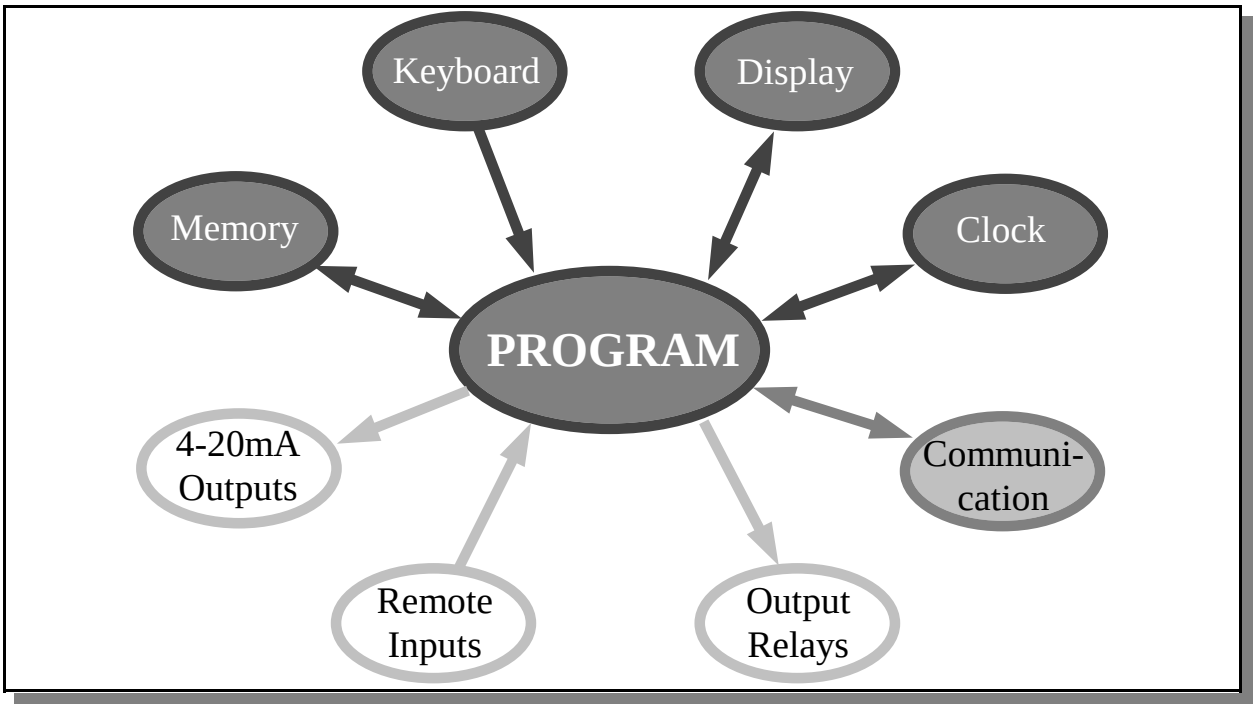
CHAPTER 1- GETTING STARTED

The SMART1’s Components

Components Overview

Before we begin to find out how to install the SMART1 Programmer and how to use it, we will stop to take a look at the SMART1's logical structure. A general overview of the device's components, capabilities and limits will give us a rough idea of the kind of tasks for which we could program it.

From a program point of view, the SMART1 consists of **Memory** for data storage and a number of different **Input/Output** components. All the things conveying information from the outside world into the program, we consider to be inputs. Everything which can be controlled or manipulated by the program, we call an output. The graphic below shows us what the particular inputs and outputs are. The elements pointing towards the program in the center are inputs. The ones with the outward pointing arrows denote the outputs. There are also objects with arrows in both directions. They can be used for input as well as output functions. We call them bi-directional. To our surprise we find the display among them. This seems like a mistake and is, indeed, only half right. So, without getting ahead of ourselves let's just say that under special circumstances the display works in conjunction with the keypad as an input device.



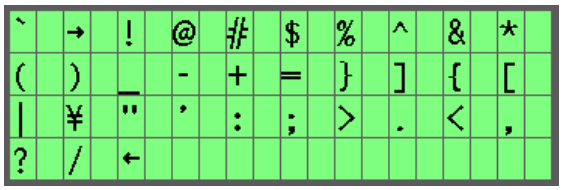
Examining the physical structure of the SMART1 we must distinguish the above components also by their location and availability. The elements shown in a dark gray with white lettering are built right into the SMART1's enclosure. They are available at all times. The communication, however, will work only when there is something to communicate with. The unit is equipped with five serial (RS232 compatible) connectors for the transmission and reception of data. The third group in the picture, shown in white with black writing, is a collection of components made available through the use of additional hardware. To access any of them, we will have to hook up one or the other of Western Scale's option boards to the big connector in the back of the SMART1. The following pages hold a brief description for each of the SMART1's components.

Keypad

Via the keypad we can input all single digit numbers from 0 to 9, multi digit numbers with or without a decimal point, all of the capitals and small letters provided by the English alphabet, words composed of such letters or even whole sentences.

Display

The display is divided into 80 individual spaces, neatly arranged in 4 rows by 20 columns. Each of these spaces may contain exactly one character, being either a letter, a (single digit) number, a blank space or any one of the signs from this picture.



Clock

Whether the SMART1 is powered up or not, the clock inside it is relentlessly ticking. We can read precise values from it ranging from year to month, day, weekday, hour, minutes and seconds all the way down to tenths and hundredths of a second. This gives us not only the potential to use the time and date in our applications, but also the chance to build some slick timer routines (a feature that might come in handy if we ever decide to create a relay control program). Since time and date could vary from one place to another, we can not only read the clock but also set it.

Communication

The SMART1's five communication channels are suitable to connect the unit to other serial communication devices for the purpose of information exchange. Some examples for such devices are: personal computers, "dumb" terminals, printers, score boards, other SMART1s and most importantly, scale indicators! All five channels can be used to both send and receive data and can be configured for various speeds and communication protocols.

Output Relays

If we are planning to do a job that requires controlling of electrical appliances such as lamps, solenoids, motors etc., we will need to extend the built in capabilities of the SMART1. A box, connected via cable to the unit's I/O expansion port will do the trick, provided it contains the appropriate equipment. One or two of Western Scale's *Setpoint Output Module Racks*, each of which outfitted with up to six output modules, will take over the task of translating the SMART1's tiny electronic signals into heavy duty currents and voltages.

Remote Inputs

For some applications the SMART1 may have to evaluate direct inputs from outside sources. We call them remote inputs. An example of a remote input could be a limit switch that is mounted to a mechanism to be monitored, let's say a gate on a hopper scale. This switch would tell the SMART1 whether the gate is open or closed. Another use for remote inputs might be a set of push buttons, pressed by the operator to start or to stop a process. Like output relays, remote inputs are connected to the I/O expansion port through additional circuitry. One SMART1 can process a maximum of twelve remote inputs.

4-20mA Outputs

The third group of hardware that we can connect to the SMART1's I/O expansion port are 4-20mA outputs. These special outputs deliver a constant current to meters, electrical gauges, PLCs and the like. Properly programmed, the value of this current can be used to represent the value of a number in the SMART1's memory, for instance a weight. Whenever the weight changes, the current at the output follows that change in a proportional characteristic. It is possible to attach two 4-20mA outputs to one SMART1.

Memory

Memory is employed to store things we want to remember. That goes for the SMART1 as well as for us. Comparable to how we humans memorize what we obtain through our senses, the SMART1 can store information it receives from its input channels and retrieve them at a later time. And there is another similarity. We think about facts we learned, link them to previous experiences, merge or change them and then memorize the result of our pondering processes. The SMART1, on the other hand, commands an assortment of system functions, allowing it to manipulate and blend the data in its memory cells and to save it in the modified form. In spite of all the parallels, there remains one fundamental difference between the SMART1 and us. We forget. The SMART1 retains the content of its entire memory throughout power-ups and -downs until the moment when it is overwritten by a program instruction.

The amount of information a SMART1 can store depends on several factors including the size of files, type of data, the complexity of the program and so on. Its overall memory capacity set aside for this purpose is approximately 28 kilobytes, allowing it to store about 7000 numbers.

We can extend the maximum numbers for all external components as well as the amount of memory to virtually any total by 'daisy chaining'. That means using two or more SMART1s for the same job and programming them in a way that they would divide the tasks among themselves.

Let's assume we needed 15 output relays. We'd take two SMART1s. The first one we would program to handle 12 relays. Whenever relays 13 to 15 need to be switched we would send a suitable command to a serial port. The second SMART1, connected to the said serial port of the first one receives the command, figures it out and switches the relays. Voilà!

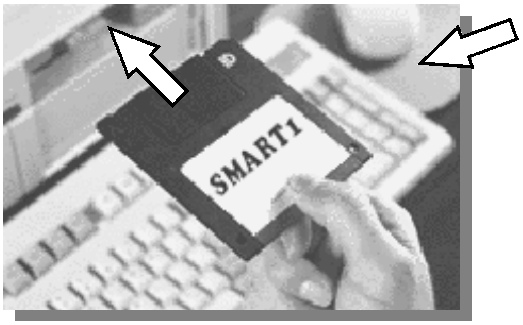
The Software Shell, first part

Installing the SMART1 Programmer Software

Reading the previous pages has given us a pretty good idea about the kind of operations the SMART1 could be utilized for. At this point we are ready to load the software that came with this manual. As a last step before we start the actual installation process, we will check whether our computer meets the minimum requirements to run the SMART1 Programmer.

- ☑ First and foremost, it must be an **IBM compatible PC**.
- ☑ Its CPU chip should be at least a **486** running at a speed of **66 MHz** or faster.
- ☑ A minimum amount of **12 Megabytes RAM** is needed.
- ☑ Approximately **2 Megabytes** of free **hard drive space** are necessary for the installation.
- ☑ To accept the installation diskette, it must have a **$3^1/_2$" floppy disk drive**.
- ☑ To program a SMART1, our PC needs to have a free **serial port**. (COM1 or COM2).
- ☑ The PC's operating system must be **Windows** (3.11 or 95 or 98 or NT).

If all conditions are met, we can begin the procedure. To install the SMART1 Programmer software on our computer we will have to:



1. insert the installation diskette into the floppy drive
2. click the **Start** button (Windows 95/98/NT) or click the **File** menu (Windows 3.11)
3. click **Run**
4. type **A:Setup** (or B:Setup, if our drive is called B)
5. press **ENTER** or click the **Ok** button
6. watch the screen and respond to the prompts in the Setup program

After restarting the computer we can run the SMART1 programmer. Again, there are slightly varying methods to do this, depending on the operating system we are using.

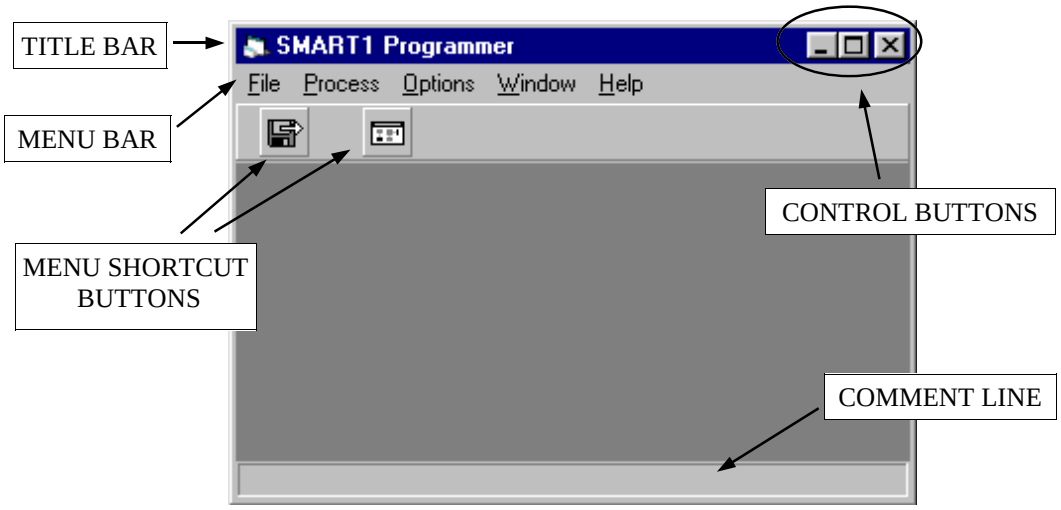
Windows 3.11 will most likely have put a new group named ***Smart1*** onto the program manager screen. After a typical installation it will contain two icons. Running the ***Uninstall*** program would remove all the software we just installed, so let's leave it alone. The icon named ***Smart1 Programmer*** is the one we double click to start the SMART1 programmer.

All of us running **Windows 95/98/NT** are facing a different scenario. First we have to click the ***Start*** button, then go to the ***Program*** menu and after that to the ***Smart1*** menu. Now we can see the two icons labeled ***Smart1 Programmer*** and ***Uninstall***.

To make the procedure of starting the SMART1 programmer under Windows 95/98/NT a little easier, we can put a shortcut on the desktop. To do this we use the right button of our mouse and click on a free spot on the screen. A menu appears. Going to ***New*** and clicking on ***Shortcut*** brings up a window in which we enter the specifications for the new shortcut. The command line must read: **\Smart1\Smart1.exe**, so that Windows can find the file. The shortcut's name is up to us. We choose to call it **SMART1 Programmer**. Using the new icon that pops up on our computer screen, we can now start our software in a very simple way. All it takes is to move the mouse cursor over it and to click the left mouse button twice. Let's do it!

The Layout of the User Interface

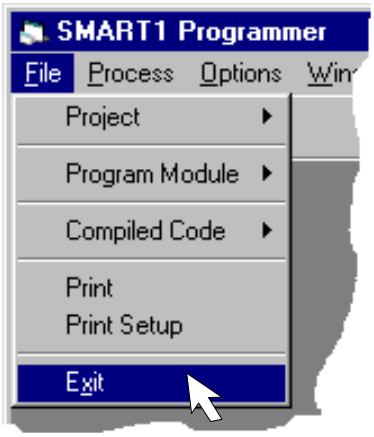
We have now started the SMART1 programmer. If our computer is running Windows 95/98/NT, then what pops up on the screen resembles the picture below. Under Windows 3.11 the program will look somewhat different but its general appearance will be the same.



The title bar on the top tells us the name of the SMART1 Programmer. It is "SMART1 Programmer". Later on there will be more (valuable) information in this spot.
The control buttons fulfill the exact same functions as they do in any other Windows application and will therefore not be explained any further in this manual.

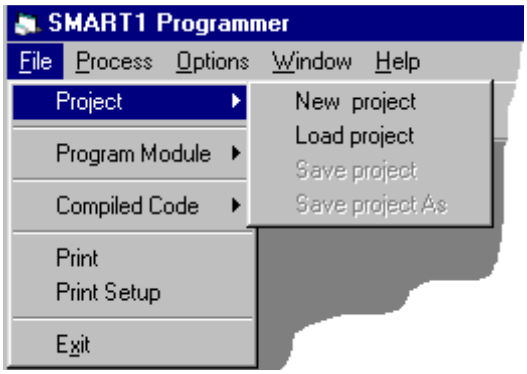
Situated on the bottom edge is a space called the *comment line*. It is used to display helpful remarks about certain objects on the screen. When we move the mouse cursor over the two menu shortcut buttons, for instance, a brief description of what they do when they are pressed is shown on the comment line. But we won't press these buttons just yet. Not before we have discussed some more important things.

The most important area currently on the screen is the menu bar. It gives us a variety of items to choose from. To start off, let's pick the one on the left which reads ***File***. To do this there are basically two different ways. The first way is to move our mouse cursor anywhere over the word ***File*** and click the left mouse button. The other way is to hold down the key labeled ***Alt*** and then press the letter ***F***. This procedure is referred to as 'using the Hot Key'. It will be available as an alternative to using the mouse at many occasions throughout the SMART1 Programmer. Whenever a letter in a menu or on a button is underlined then we can use it in conjunction with ***Alt*** as the Hot Key to invoke the menu's or button's function.



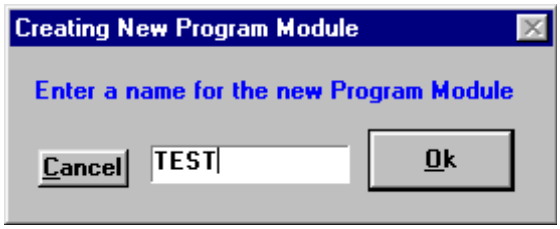
Now that we have decided on how to open the File menu, we are presented with even more choices. For all of us who have already seen enough, and for those who would like to practice what they have learned so far, the next step will be clicking on the word ***Exit*** and thereby leaving the SMART1 programmer. Everyone else may go on to the next page where the manual continues.

Projects and Modules



The first item in the File menu is **Project**. That's what we call the entirety of all the things that make up a particular program for the SMART1.

As we click on the word Project, yet another menu flies out to the side. It reveals four items, two of which are black and the other two are sort of hazy. The hazy ones are currently "disabled" and might become available at a later time. The black items, however, tell us which project functions we could use right now. The second one of them, named **Load project,** somehow looks familiar. Yes, we have read the words 'Load a project from disk' on the comment line when we moved the mouse over one of the menu shortcut buttons. Now we know how the term 'shortcut' fits in. We could have gotten away with just one mouse-click, where it took us two menu steps to get here and still one more click to load a project. But, since we've never made a project, we can't load one anyway. Well, let's choose **New project** instead and carry on.



The menu vanishes and a new window appears in the center of the screen, asking us to enter a name for the new Program Module. Even though we have no idea what a program module is, we boldly try to give it a name. After some failed attempts to enter names like *JOHN DOE*, *ME&YOU* or *TERMINATOR*, we realize that we can only put in a maximum of eight letters or numbers and that spaces and things like *!&#$%* are not accepted. So we settle for the name *TEST* and push the ***Ok*** button.

The layout changes rapidly. Along with two more buttons on the shortcut bar a whole new row of them materializes as well as a big white window that takes up almost all the space on the screen. It bears the word *Test* as title and we conclude that this must be the program module we just named. This is absolutely right, but what exactly is a program module?

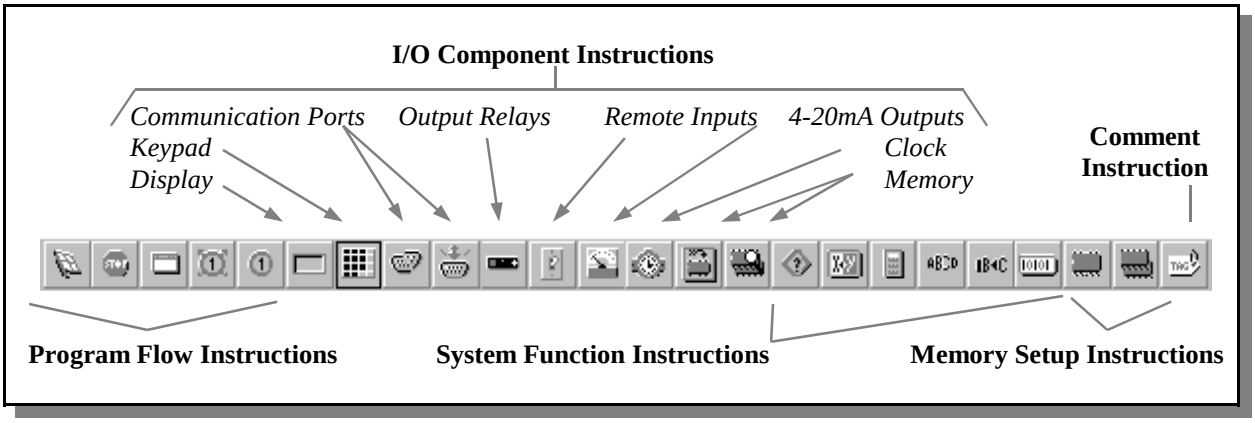
A **program module** is a container for the commands that make up a project. Like a sheet of paper we would use to jot down directions for someone who has to get from point A to point B, the program module is used for instructions telling the SMART1 what to do. If the way from A to B was very complicated or we were also giving directions on how to get from point B to point C, we would perhaps describe it on two or more pieces of paper. Much the same principle is applicable to program modules. If we have to design a project for a complex application we will create several individual procedures and place each of them in its separate program module. We can create a new module either by following the path from ***File*** menu via ***Program module*** menu to ***New program module*** or by simply clicking on the proper shortcut button. Every time we open a new module, we must give it a name which has to be different from all the other modules' names in the project.

Button Bars

Let's have a look at the other things that appeared when we opened the new program module. Well, there are two more buttons on the shortcut bar. One of them looks like another Load Project button except that the arrow points towards the disk. It is the ***Save Project*** button. Its function is to save the project that is presently in the SMART1 programmer. The other new button is called the ***Compile Button***. Pressing it will compile all instructions of a project, meaning it will translate them into "machine code", a strange language consisting exclusively of 0s and 1s. Nevertheless, the SMART1 understands it perfectly. Since we haven't made any projects yet, we have no need for this button at the present time.

The section beneath the shortcut bar holds a sheer multitude of buttons. We can push each of them to pre-select the type of the next instruction we want to put into a program module. According to their purpose these buttons are called ***Instruction selector buttons***. Every one of them stands for a different type of instructions. There are some that correspond to **system functions**, others that manage the **program flow** or the **memory setup**. One group of buttons is directly related to the SMART1's **I/O components** which we discussed at the beginning of the manual and, finally, there is a button for putting **comments** into our programs.

The graphic below points out the button groups and lists the corresponding component for each button in the I/O components group.

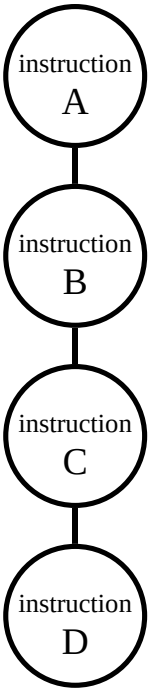


Once we click on one of the instruction selector buttons, it will stay depressed. The example shows the *Keypad selector* as being pushed, so we too click on the keypad button. This doesn't accomplish much yet, because to actually place an instruction we must tell the SMART1 programmer exactly where in the program module we want it. But before we can do this efficiently we will need a little background information about the method that is used to program the SMART1. We must learn about its structure, the internal workings of the language and the rules it is governed by.

Fortunately, it just so happens that the chapter containing the definitions for exactly these things is starting on the very next page, so let's read on.

The Programming Language

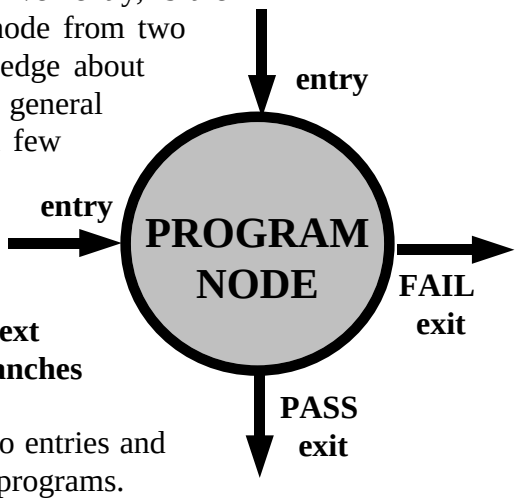
PLAIN - Programmable Logically Aligned Interconnected Nodes



The general idea behind the PLAIN programming language, developed for the SMART1, is to use **programmable** graphical symbols rather than words to represent individual instructions. We call these symbols **nodes**. Like nodes in a plant, from where the stalks and leaves grow, our program nodes are the junction points for the "growth" of the program. Contrary to plants, however, our programs expand from the top down. The nodes are **aligned** by their centers and are **interconnected** by lines called *ties* showing the path of the program **logic**. The tie leading from one instruction to the next following instruction is attached to the bottom of the node. This is the node's *exit*. The node's *entry*, located at the top, is the connection point for the tie coming from the previous instruction. In our example the first instruction to be executed is *A*. After completion of *A* the program flows down to instruction *B*. When *B* is done the flow continues to *C* and so on. For a better understanding of this example, here is a possible application for it: - Instruction *A* displays a prompt for the operator to enter a number.

- Instruction *B* is reading the operator's input from the keypad.
- In *C* the number is sent to a printer.
- Finally instruction *D* writes 'number printed' on the display.

This is a valid program and it will work just fine. But, unless the printer is ready to receive data, our little program will never get to instruction *D*. Instead it will sit at instruction *C* and wait for the printer's ready signal. Well, sometimes this may be exactly what we want the SMART1 to do. On other occasions, usually after a certain length of time has expired, we want to let the operator know that there's something wrong with the printer. If this is the case, we can not have the program stop and do nothing. It would have to choose an alternative path instead and perform steps to make the operator aware of the problem. This generates the need for an exit that would link the node to the alternative path, in case its function fails. For this very reason our node is equipped with an additional exit on its right hand side. To distinguish between the two exits we give them different names. The one used if the node passes its task we call the *pass exit*, the new one, going into action whenever the function fails is the *fail exit*. Having an exit on the right of a node suggests to also have an entry on the following node's left hand side, so that we can put a straight tie between the two. The entries of a node can be treated equally. Whichever one can be connected to conveniently, is the one that gets used. It is even possible to approach a node from two different paths, utilizing both entries. With this knowledge about entries and exits, we can now easily understand the general scheme of a PLAIN program. Let's summarize it in a few words:

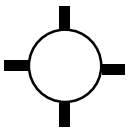
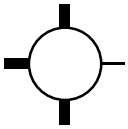
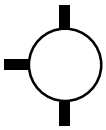
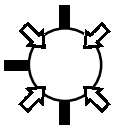
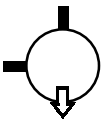
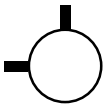
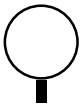


The instructions in a program are executed one by one. They are shown as nodes, tied together in straight lines. As long as nodes pass their assignments the program flow continues down to the next node below. If a node fails its task, the program branches through the node's fail exit on its right hand side.

The figure shows a normal program node with its two entries and two exits. It is the standard building block of all PLAIN programs.

Node Variations

Not all program nodes have two entries and two exits like the normal node we have seen on the previous page. They actually come in quite a variety. Some have no fail exit, others lose or acquire their fail exit when they get programmed. There are some nodes without entries and some without exits. We even have nodes that feature a remote exit or remote entries. How many entries or exits a node possesses depends on its instruction type and its internal program. Here are the symbols that are used in this manual to describe the different node types:

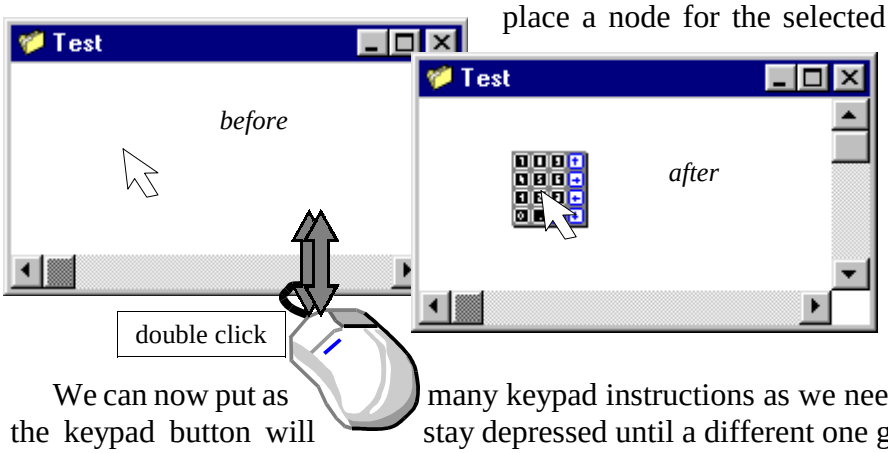
node symbol	description
	**2 entries**, **pass exit**, **fail exit**
	**2 entries**, **pass exit**, the presence of a fail exit depends on the instruction's internal program
	**2 entries**, **pass exit**, no fail exit
	**2 entries**, **pass exit,** no fail exit in addition to its two standard entries this node has an **unlimited number of remote entries**
	**2 entries**, no fail exit this node has no standard pass exit, it has a **remote pass exit** instead
	**2 entries**, no pass exit, no fail exit
	no entries, **pass exit**, no fail exit
	no entries, no exits

If a node has entries, then all of them including the remote kind are equally useable, but at least one of them must be used in order for the node to be valid. Exits, on the other hand, can never be left open. Every exit must be tied to another node's entry.

Making a Program Module

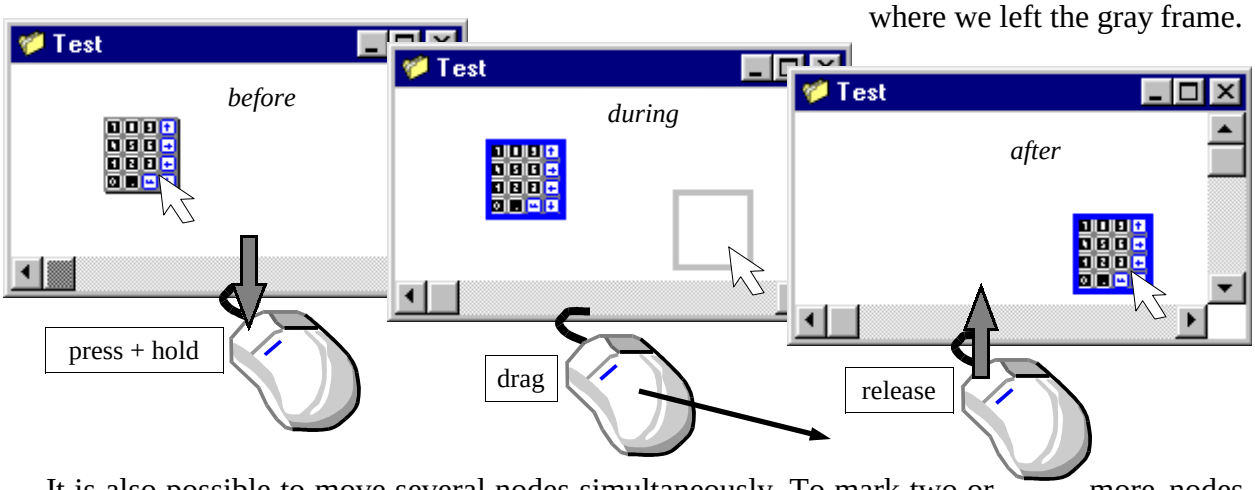
Placing and Moving of Nodes

Now that we know the basic facts about the PLAIN language, we are ready to start working on our first module. Assuming that we had pressed a button on the instruction selector bar we can place a node for the selected instruction type by double clicking on a free spot in the module. An image, more or less resembling the picture on the pushed button, appears at that very location. In our example the image looks like a miniature version of the SMART1's keypad.

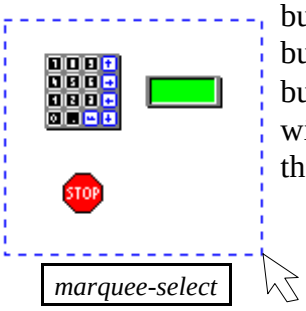


We can now put as many keypad instructions as we need into our program module, the keypad button will stay depressed until a different one gets selected.

To move a node from one place to another we have to utilize the *click-and-drag* procedure. How does this work? We move the mouse cursor over the node we want to move. We then push the left mouse button and hold it down. A blue frame, the marker for a selected node, appears around it. Now we move the mouse to the desired location, dragging a gray frame the size of the node with it. As soon as we release the mouse button, the node framed in blue jumps to the spot where we left the gray frame.



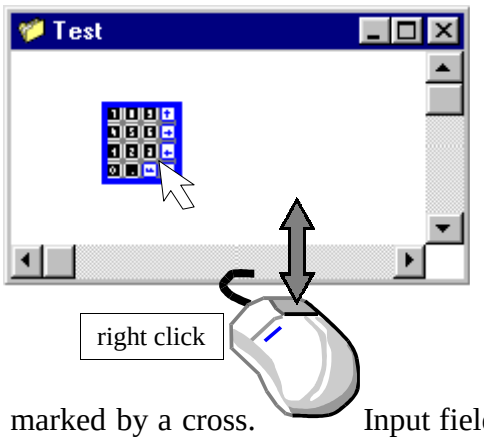
It is also possible to move several nodes simultaneously. To mark two or more nodes at the same time we have to apply a method known as *marquee-select.* Pressing the left mouse button in the module background and then moving the mouse with the button held down, we are drawing a dashed line box. Once we let go of the button, all the nodes that were **completely enclosed** by the marquee box will now have a blue selection frame. When we *click-and-drag* any one of them, all the other marked nodes will follow.



Nodes can not be moved from one program module to a different one. An attempt to move a single node or a block of them to an area outside the module will fail and the nodes will remain in their original places.

Programming of Nodes

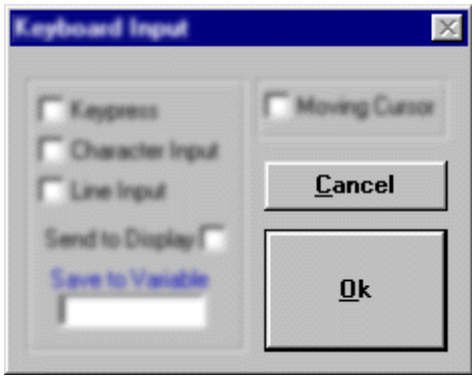
The nodes we placed in our module so far are merely pictures telling us what instruction types they belong to. To become complete instructions we must individually program most of them for their specific tasks.



By clicking the right mouse button on a node, we start the programming process. This usually means that a programming window is opened which contains an assortment of check boxes, input fields, buttons or other controls. The most widely used controls are check boxes and input fields. Check boxes are little white squares with a describing text beside them. They are used to select one or more items from a list of options. When we click on a check box, then we activate or deactivate the option associated with it. An activated check box is marked by a cross. Input fields are rectangular areas into which we can write text or the name of a variable. On some occasions the controls in the programming window are functionally interconnected. In these cases typing into an input field or activating a check box might have an effect on one or more of the other controls.

Naturally the programming window will look different from one instruction type to the next and will feature different controls. The controls in the individual instruction types' programming window are explained in the manual's *CHAPTER 2 - SMART1 INSTRUCTION TYPES.*

The only controls common to every programming window are two buttons labeled Ok and Cancel. The **Ok button** is meant to be pressed when we are finished entering all necessary data and have clicked all the appropriate check boxes. If everything we did is making sense, then clicking this button will close the programming window and our instruction is ready to be executed. If we failed to provide information that the instruction needs, then the Ok button will simply refuse to work for us. The **Cancel button**, on the other hand, gets us out of the programming window at any rate, though the instruction will remain in the same state as it was before we opened the programming window. As an alternative to clicking the Cancel button we can press the **Escape** key on our computer's keyboard whereas the **Enter** key does the same as clicking Ok.



A few instructions can be programmed without employing a programming window. For those instructions the programming window is substituted with a simple text input field that appears on top of the node and the programming process consists solely of entering a text, a name or a number.

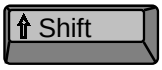
Finally some of the SMART1's instruction types do not require any programming at all and will therefore stubbornly resist our bravest attempts to force a programming procedure upon them.

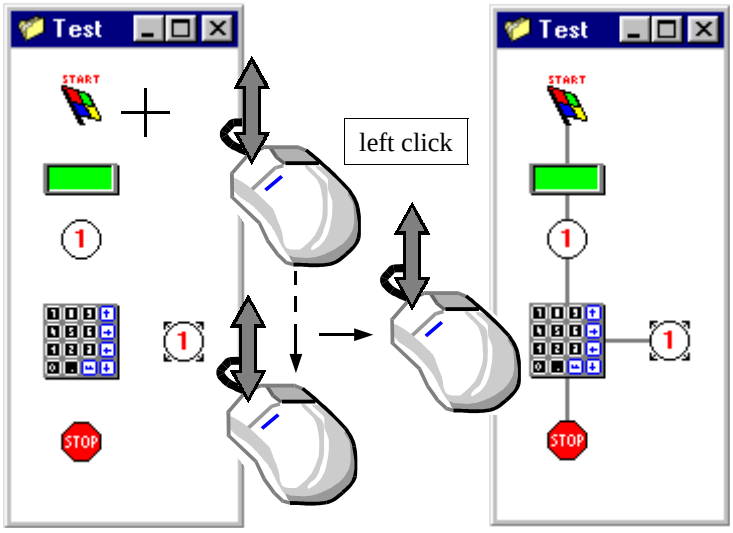
Connecting of Nodes

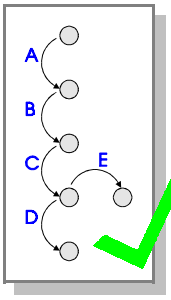
Arriving at this section of the manual we presumably have placed some nodes into our module and maybe even programmed them already. Now we are wondering how we can tie the nodes together to make them form a program path.

The keyword here is alignment! The keen ones among us will have noticed by now that our nodes can't be moved to just any spot and that they rather seem to be snapping to an invisible grid. This is indeed the case and it's done for a reason. Because, before we can put a tie between two nodes their **centers must be aligned** at either the same horizontal or the same vertical level. The snapping is supposed to assist us a little in lining them up. After we managed to arrange our images in straight lines we are ready to connect them.

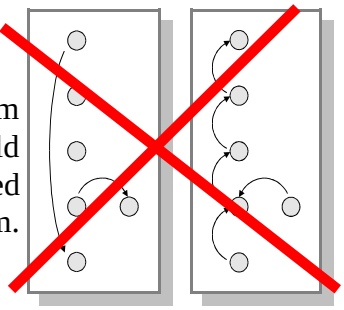
In the figure below we placed five instruction nodes in the "*Test*" module. Why don't we try to connect them! When we press the **Shift** key the mouse cursor changes from an arrow into a pair of cross hairs. Let's hold the key down and click on the first of our nodes, the start flag. Nothing happens yet. Now, without releasing the shift key, let's click on the node immediately below. There we go, we've made our first tie! The next ones are going to be just as easy. Tie number two is supposed to connect the display instruction with the 1 in the circle. Still pressing the shift key, we are drawing the tie by first clicking on the display and then on the circle. Another click on the circle followed by one on the keypad symbol gives us our third tie. The fourth tie starts where the third one ended, at the keypad. We click on it, then move the mouse cursor over the stop sign and click again. Finally, tie number five. Again, we click on the keypad and for the last time on the circle with the arrows around it and the 1 inside. We can now let go of the shift key. Actually, if we had released it during the process it would have been okay, as long as we had pressed it in time before the start of each new tie.







The sketch on the left shows us the steps we followed to tie our nodes together. They are labeled A to E in the order in which we executed them. As a matter of fact, we could have reached the same result by interchanging any of these steps in any possible order, for instance BDACE or EACDB. The only important issue here is, that we must always connect a node to the next one **immediately below** or **immediately to the right**. We can never make a tie going from a node upward or to the left. And even though a tie would appear if we were to skip a few nodes, they will be detected as unconnected nodes and we won't be able to compile our program.



Erasing of Nodes

If we need to remove one or more nodes from our program module, then we have to select them first. For a single one we just click on it to do this. To select two or more nodes we perform a *marquee selection* as described on the page on which we discussed the placing and moving of nodes. The selected nodes will be marked by that nice blue frame. To get rid of them we must now find the **Delete** key (sometimes it's labeled *"Del"*) on the PC's keyboard and press it.

Erasing of Ties

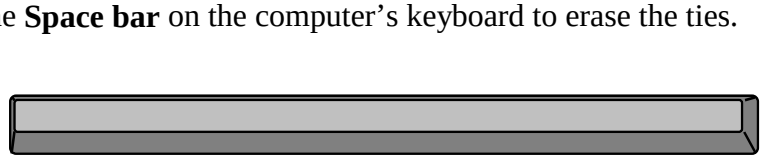
There are a number of different methods we can choose from to eliminate the ties that connect our nodes. Some of them work directly, others are results of procedures that we apply to nodes.

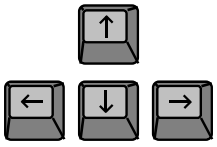
If, for example, we **delete a node that has ties attached** to it, then these ties will be erased automatically along with the node.

The process of programming a node might also have an effect on a tie. As we already know, some nodes acquire or lose their fail exit depending on the particular instruction they are being programmed with. If we are programming one of these nodes and it happens to have a fail exit with a tie, then it would lose that tie in case the new instruction causes it to **drop the fail exit**.

Another indirect method for getting rid of ties can be the moving of nodes. No matter whether we drag a single node or a whole bunch of them across our program module, as long as they stay aligned with the ones they are connected to and don't change their top to bottom or left to right order, then the only thing happening to the ties will be some stretching or compressing. A sure way of breaking ties, though, is to **move a node out of alignment**.

If we want to remove ties directly without having to modify a node, then we can do so as well. Unfortunately, we can't select a tie by clicking on it, but we can select the node which it is attached to. Just like the dragging and the erasing of nodes, the direct removing of ties allows us to select either a single node by clicking on it or a group of them utilizing the *marquee-select* process. After we are done selecting our nodes we can use the **Arrow keys** or the **Space bar** on the computer's keyboard to erase the ties.



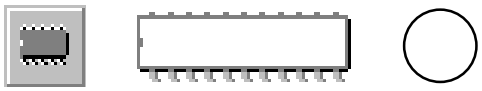


Pressing the space bar will result in all ties of all selected nodes being erased. The arrow keys, however, let us point into the direction of the tie we want to wipe out. If, for instance, the up arrow is pressed, then the tie at the top entry of all selected nodes will disappear, whereas pushing the left arrow is going to delete only the tie at the left entry of all blue framed nodes. We can probably all guess by now, what the right arrow's and the down arrow's function is. Exactly, the down arrow removes the pass exit's tie and the right arrow does the same to the tie at the fail exit of all selected nodes.

CHAPTER 2 - SMART1 INSTRUCTION TYPES

Memory Setup Declarations

Variables



selector button | node image | node type

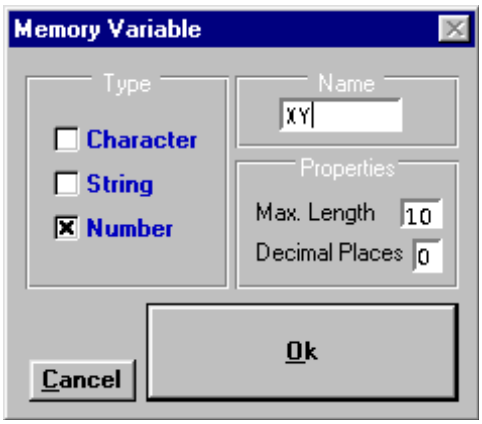
No serious programming scheme gets away without using variables. They lend flexibility to a program by enabling it to react to inputs or other changing conditions. Most of the instructions that deal with the SMART1's I/O components and system functions either have the option for a variable or absolutely require the use of one to do their task. Let's find out what variables are in a SMART1 setting and how we are supposed to use them.

We all remember the word *variables* with mixed feelings when we think back to our elementary school math classes. They were usually called X or Y, appeared mostly in formulas and could represent varying numbers, depending on other values in the formula and on our calculating skills.

Variables for the SMART1 are not much different from those, except that this time around, we leave the calculations to the electronics. Just like in math formulas, SMART1 variables are used to hold operands and results of calculations. In addition to numbers the SMART1 also allows us to store letters, whole words and similar things in variables. We use variables for all numbers, words, etc. that might or will change during the execution of the program, as well as for data of which we don't know the value or content at the time we create the project.

When we create a variable for the SMART1 we have to know what kind of data we want to store in it. Depending on this we must declare the variable to be either a *Character*, a *String* or a *Number*. These are the three variable types the SMART1 knows how to handle. A variable can only be used for the type of data it is designated for. The main reason for this is, that the precise amount of space each variable needs gets reserved in the SMART1's memory chip. Now we know what a variable essentially is. It is a pre-defined space in memory, set aside to hold a certain type of data.

Like program instructions, variables are represented in our program modules by nodes. A node that stands for a variable can be created, moved, erased and programmed just like any program instruction node. However, since a variable declaration is not an executable instruction and therefore not a part of the program flow, its node has neither entries nor exits. We can not attach any ties to it.

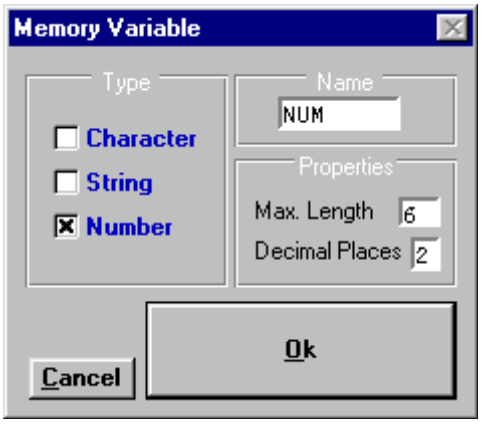


To program, or rather declare a variable, we use the *Memory Variable* window. It pops up when we right-click on a variable declaration node. Here we can give our variable a name and specify its type. For choosing the name we have to obey the same rules that apply to the naming of program modules. Depending on the type we assign to our variable, we may also have to enter values in the fields labeled *Max. Length* and *Decimal Places*. To find out what the purpose of those two parameters is, we will have to read the in-depth explanations for the three individual variable types on the following pages.

Number Variables

We declare a variable to be of the *number* type when we need it to hold numerical data for calculation purposes. In other words, whenever we want to add, subtract, multiply or divide, only a *number* variable will do.

Numbers are stored in the SMART1's memory in a way that makes it easy for the electronics to work with them. Every number, regardless of its actual value occupies four bytes of memory. Unfortunately, these numbers bear very little resemblance to the type of numbers we are used to dealing with. Therefore, if we want to look at a number, we have to tell the SMART1 in which shape we expect it to appear. To accomplish this, we use the settings for *maximum length* and *decimal places*. They are used to "format" the number when it is displayed on the SMART1's LCD screen or when it is sent out a serial port. In these cases the number gets translated back into the human-readable form so we can easily recognize it. When the number is shown, it will take up the amount of spaces specified for its maximum length and it will always be shown with the designated number of decimal places. The value for maximum length includes the places for decimal point and minus sign. If the number has more decimal places than are supposed to be displayed, then it will be rounded. If it has less decimal places, then zeroes (0) are appended to the number. In case the number has a value too big to be displayed in the desired format, an overflow error occurs and a star (*) will be shown in every place. The number will be displayed "right aligned" and "left padded with blanks". Assuming we have declared the variable NUM to be a *Number* with a *Maximum Length* of 6 and 2 *Decimal Places*, it will always appear in this format:



?	?	N	.	N	N

The meaning of the individual symbols is:

? can be a blank space, a minus sign or a number (0..9)
. the decimal point will always be in this location
N will always a be number (0..9)

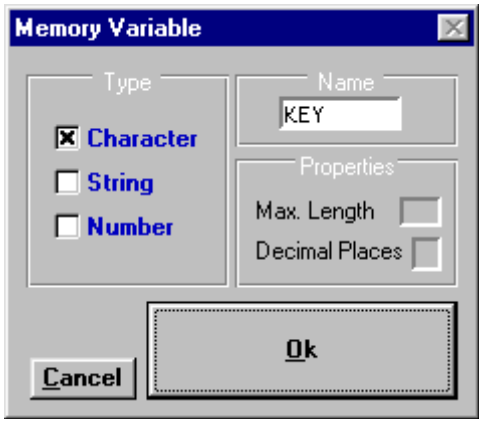
This sounds a little complicated, so let's look at a few examples to get a better understanding.

NUM = 1.25			1	.	2	5	
NUM = 100	1	0	0	.	0	0	
NUM = 1234	*	*	*	*	*	*	

NUM = 2.876			2	.	8	8	
NUM = -3.1		-	3	.	1	0	
NUM = -222.08	*	*	*	*	*	*	

Due to the format in which numbers are stored in memory, the biggest possible absolute value for any number without decimal point is **16777215**, provided the setting for maximum length allows it to be that long in the first place. To figure out the maximum for numbers with decimal point we simply ignore the decimal point. The absolute maximum for a number with 2 decimal places for instance would be **167772.15**, for 3 decimal places **16777.215** and so on.

Character Variables



Character Variables are the simplest of the SMART1's *Variable Types*. A variable of this type always occupies exactly one byte of memory. Because a character never takes up more than one space when it is displayed or printed, the parameters for maximum length and decimal places are irrelevant.

Of course, we all probably knew what numbers were before we heard about the SMART1's *Number Variables*. Characters, however, are not as common an occurrence in our everyday lives and maybe not all of us are familiar with them. Let's shed some light onto them.

For starters, a character is a letter. It can be either the capital or the small version of any letter of the alphabet. A character can also take the shape of any number from 0 to 9. Furthermore we can use *Character Variables* to store punctuation marks or a variety of signs like @ # $ % ^ & * and so on. Even a blank space qualifies as a character. Generally, we can say that each of the symbols, that show up on our PC-screen when we press a letter or number button on the keyboard, is a character. With a few exceptions, all those characters can also be shown on the SMART1's display. But that's not all there is to characters.

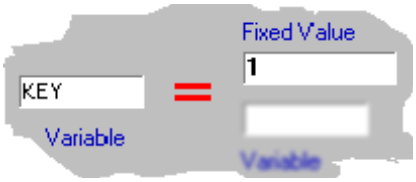
In addition to the so called *Printable Characters* or *Displayable Characters* there is a number of *Non-Printable Characters*. We can divide them into two categories. First there are the ones generally known as *Control Characters*. Sent to a display, these characters do not normally produce any symbols, but perform a function instead. Most of us have probably heard the expressions *Back Space*, *Carriage Return* or *Line Feed* before. These are the names of three typical control characters. They date back to the days of the type writer, but are still used wherever something is printed. Their function is to position the cursor. This is the little pointer which determines where the next character is displayed. The *Back Space Character*, for example, moves the cursor one step back, so that whatever happens to be in this particular location will be overwritten by the next character. When a *Carriage Return* is issued, then the cursor moves to the very first character in its present line. To move the cursor to the next line below, a *Line Feed Character* must be sent to the display. Beside those three, there is one other *Control Character* that has a special meaning for the SMART1. It's named the *Null Character*. We will learn about it in the paragraph that explains the type of the *String Variable*. Then there are 28 more *Control Characters* which are all insignificant for the SMART1 itself, but usually assume a special role when they are sent to a printer or another serial communications device.

Members of the second group of *Non-Printable Characters* are often named *Extended Characters.* To classify them as *Non-Printable Characters* is not entirely correct, but we do it anyhow because they yield nothing on the SMART1's display. Sent to a printer, on the other hand, they may or may not produce a symbol on the paper. It all depends on the kind of printer and its internal settings. If it is equipped to handle *Extended Characters*, then it is most likely going to print things such as À, œ, Ê, í, õ, Ü, µ or something similar.

All in all there are 256 different characters. On the next page we'll learn how to handle them.

Values of Characters

When we are going to create program modules for our SMART1 projects we will probably be employing C*haracter Variables* for many things. Perhaps the first task we're going to need a *Character Variable* for, is to store the value of a key that got pressed. Let's assume, for the sake of argument, that there was a message on the SMART1's display which prompted the operator to press the key labeled 1. Let's further assume that the input from the keypad is being captured in a *Character Variable*. Now, curious as we are, we want to know whether the operator had really pushed the right button. In order to accomplish this, we will check if the content of our variable is equal to the character 1. There is an instruction that lets us do exactly that. When we program it, we have to enter the name of our variable and put the character we are comparing it with in the *Fixed Value* field. Since 1 is a *Printable Character* we simply type it here. We are allowed to enclose the *Fixed Value* in single quotation marks **'** , a method that is particularly useful for entering a blank space. We would proceed in the same manner, if we were comparing the variable to any other *Printable Character*. When it comes to entering a *non-printable Character*, however, we won't be able to just type it. We will have to use a different approach. As we already know, there are 256 characters. If we keep on reading, at least as far as to the next page, then we will also find out that these characters are arranged in a particular order and that they are numbered from 0 to 255. This being the case, we now have an easy way for specifying any character as a *Fixed Value*. We just enter its number! Actually, the proper term is "decimal character value" and we must always put the number sign **#** immediately in front of it. But there are other ways to define a character. We can type a dollar symbol **$** followed by the hexadecimal character value or use a percent sign **%** with the binary character value. If we choose to enter a character's decimal, hexadecimal, or binary value, then we may omit potential leading zeroes. The characters we call *Control Characters* offer us even more choices. A combination of the caret sign ^ and one other symbol represents a control code, whereas a control character's name can consist of two or three letters and always must be enclosed by triangular brackets < >. The table on the next page is a summary of all SMART1 characters, their values, codes and names .



Any one of the discussed methods to specify a character will produce the same result, provided that it's applicable for that particular character. All methods are equally useable in every program instruction that supports entering a *Fixed Value* for a character.

printable/ displayable character	decimal character value	hexadecimal character value	binary character value	control character code	control character name
	#013 or #13	$0D or $D	%00001101 or %0001101 or %001101 or %01101 or %1101	^M	<CR>
A or 'A'	#065 or #65	$41	%01000001 or %1000001		

The above example displays all possible ways for referring to a *Carriage Return* or the capital letter A, respectively.

The Table of Characters

decimal value	hexadecimal value	binary value	control code	control code name
#000	$00	%00000000	^@	<NUL>
#001	$01	%00000001	^A	<SOH>
#002	$02	%00000010	^B	<STX>
#003	$03	%00000011	^C	<ETX>
#004	$04	%00000100	^D	<EOT>
#005	$05	%00000101	^E	<ENQ>
#006	$06	%00000110	^F	<ACK>
#007	$07	%00000111	^G	<BEL>
#008	$08	%00001000	^H	<BS> (1)
#009	$09	%00001001	^I	<HT>
#010	$0A	%00001010	^J	<LF> (2)
#011	$0B	%00001011	^K	<VT>
#012	$0C	%00001100	^L	<FF>
#013	$0D	%00001101	^M	<CR> (3)
#014	$0E	%00001110	^N	<SO>
#015	$0F	%00001111	^O	<SI>
#016	$10	%00010000	^P	<DLE>
#017	$11	%00010001	^Q	<DC1>
#018	$12	%00010010	^R	<DC2>
#019	$13	%00010011	^S	<DC3>
#020	$14	%00010100	^T	<DC4>
#021	$15	%00010101	^U	<NAK>
#022	$16	%00010110	^V	<SYN>
#023	$17	%00010111	^W	<ETB>
#024	$18	%00011000	^X	<CAN>
#025	$19	%00011001	^Y	<EM>
#026	$1A	%00011010	^Z	<EOF>
#027	$1B	%00011011	^[	<ESC>
#028	$1C	%00011100	^\	<FS>
#029	$1D	%00011101	^]	<GS>
#030	$1E	%00011110	^^	<RS>
#031	$1F	%00011111	^_	<US>

decimal value	hexadecimal value	binary value	SMART1 display
#032	$20	%00100000	(4)
#033	$21	%00100001	!
#034	$22	%00100010	"
#035	$23	%00100011	#
#036	$24	%00100100	$
#037	$25	%00100101	%
#038	$26	%00100110	&
#039	$27	%00100111	'
#040	$28	%00101000	(
#041	$29	%00101001	)
#042	$2A	%00101010	*
#043	$2B	%00101011	+
#044	$2C	%00101100	,
#045	$2D	%00101101	-
#046	$2E	%00101110	.
#047	$2F	%00101111	/
#048	$30	%00110000	0
#049	$31	%00110001	1
#050	$32	%00110010	2
#051	$33	%00110011	3
#052	$34	%00110100	4
#053	$35	%00110101	5
#054	$36	%00110110	6
#055	$37	%00110111	7
#056	$38	%00111000	8
#057	$39	%00111001	9
#058	$3A	%00111010	:
#059	$3B	%00111011	;
#060	$3C	%00111100	<
#061	$3D	%00111101	=
#062	$3E	%00111110	>
#063	$3F	%00111111	?

decimal value	hexadecimal value	binary value	SMART1 display
#064	$40	%01000000	@
#065	$41	%01000001	A
#066	$42	%01000010	B
#067	$43	%01000011	C
#068	$44	%01000100	D
#069	$45	%01000101	E
#070	$46	%01000110	F
#071	$47	%01000111	G
#072	$48	%01001000	H
#073	$49	%01001001	I
#074	$4A	%01001010	J
#075	$4B	%01001011	K
#076	$4C	%01001100	L
#077	$4D	%01001101	M
#078	$4E	%01001110	N
#079	$4F	%01001111	O
#080	$50	%01010000	P
#081	$51	%01010001	Q
#082	$52	%01010010	R
#083	$53	%01010011	S
#084	$54	%01010100	T
#085	$55	%01010101	U
#086	$56	%01010110	V
#087	$57	%01010111	W
#088	$58	%01011000	X
#089	$59	%01011001	Y
#090	$5A	%01011010	Z
#091	$5B	%01011011	[
#092	$5C	%01011100	¥ (5)
#093	$5D	%01011101	]
#094	$5E	%01011110	^
#095	$5F	%01011111	_

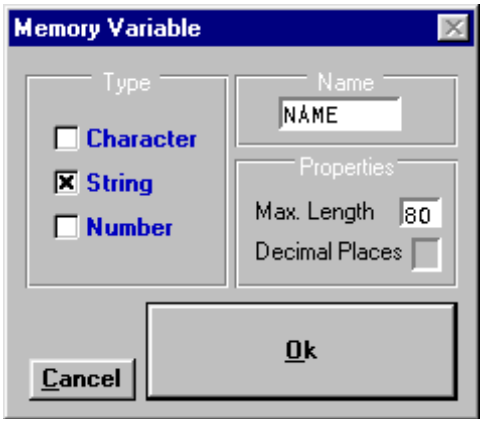
decimal value	hexadecimal value	binary value	SMART1 display
#096	$60	%01100000	`
#097	$61	%01100001	a
#098	$62	%01100010	b
#099	$63	%01100011	c
#100	$64	%01100100	d
#101	$65	%01100101	e
#102	$66	%01100110	f
#103	$67	%01100111	g
#104	$68	%01101000	h
#105	$69	%01101001	i
#106	$6A	%01101010	j
#107	$6B	%01101011	k
#108	$6C	%01101100	l
#109	$6D	%01101101	m
#110	$6E	%01101110	n
#111	$6F	%01101111	o
#112	$70	%01110000	p
#113	$71	%01110001	q
#114	$72	%01110010	r
#115	$73	%01110011	s
#116	$74	%01110100	t
#117	$75	%01110101	u
#118	$76	%01110110	v
#119	$77	%01110111	w
#120	$78	%01111000	x
#121	$79	%01111001	y
#122	$7A	%01111010	z
#123	$7B	%01111011	{
#124	$7C	%01111100	|
#125	$7D	%01111101	}
#126	$7E	%01111110	→ (6)
#127	$7F	%01111111	← (7)

(1) Back Space - This character is returned from the SMART1's keypad if the left arrow is pressed in a character input instruction. (2) Line Feed - If this character is sent to the SMART1's display, then the cursor will move one line down. If it was already on the last line, then it will go to the top. It will stay in its column. (3) Carriage Return - Coming back from a character input instruction, if ENTER was pressed. Sent to the display, it will move the cursor to the first column of its current row. (4) Space - A blank space will be displayed. (5) Yen - The PC equivalent for this character is \ . (6) Right arrow - PC equivalent: ~. (7) Left arrow - Has no PC equivalent.

A character can also take on any value from | #128 | $80 | %10000000 | to | #255 | $FF | %11111111 | . These *Extended Characters* have neither a control code, nor a displayed symbol with the SMART1.

String Variables

The last one in the SMART1's assortment of *Variable Types* is called *String Variable*. The word string generally denotes a row or series of connected things. In our case the things that are being connected are none other than characters. Well, don't we already have *Character Variables* to store characters? Yes indeed, we do, but each of them can only hold one character at a time. A *String Variable*, however, is meant to keep many of them. This kind of variable makes it very easy for us to store words, names, sentences and the like. It can even accept sequences of *Control Characters*. Theoretically we could take every possible combination of characters from our table on the previous page and put it in a *String Variable*. In actuality, though, we are bound by two restrictions. The first one is, that a *String Variable* can only hold as many characters as it has space for. When we declare a *String Variable* we must specify a value for its *Maximum Length*. It is the largest number of characters the variable is supposed to hold at one time. The absolute limit for this value is 80. The minimum is 1. This doesn't mean that our string always has to be as long as we set the *Maximum Length* to. A string is only as long as the number of characters it contains at any given time. It can even be completely empty. This brings us to the second restriction. It stipulates that we can never make a *Null Character* <NUL> a part of a string. This particular character is reserved for the purpose of marking the end of a string. Any characters following a *Null* are going to be ignored. Given this fact, we have an easy way to empty a *String Variable*. We simply assign a *Null* as its first character. The actual number of bytes a *String Variable* occupies in the SMART1's memory is its *Maximum Length* plus 1, regardless of its content.



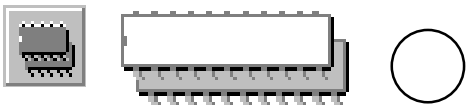
When it comes to assigning *Fixed Values* to *String Variables*, we must apply the same conventions as with *Character Variables*. We can type *Printable Characters*, use decimal, hexadecimal or binary character values or utilize the names and codes of *Control Characters*. We are also allowed to mix them in one and the same string; a detail that will undoubtedly come in handy if we ever have to interface to a printer and want to do such things as changing the size or the shape of the print font.

A single quotation mark **'** as first or as last character of any *Fixed Value* specification will be ignored. Thus: 'ABCDE' = ABCDE , '' = <NUL>.

The following example has little practical but an immense historical value. This string has been used to demonstrate diverse print commands in probably every programming language there ever was. So, once we have learned how to do it, let's come back to this page and keep up the tradition by assigning this string to a String Variable with a Maximum Length of at least 14 and then send it to a printer or to the SMART1's display.

```
$48#101$6C%1101100#111 w%1101111$72l#100!<CR>^J
```

File Definitions

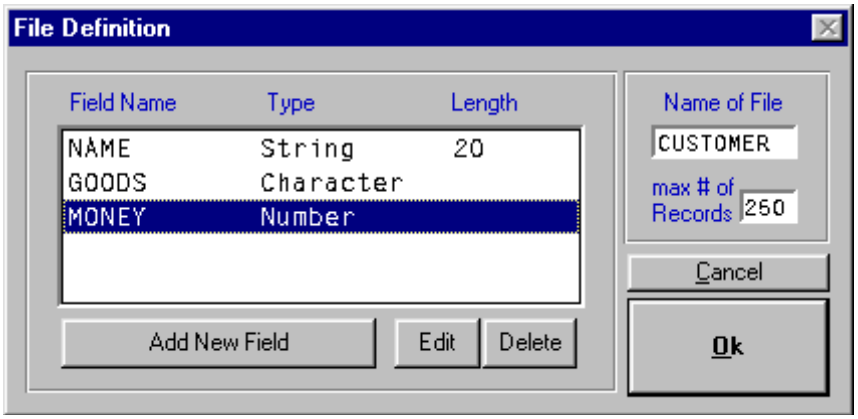


Selector Button | Node Image | Node Type

We have read about the different *Variable Types* the SMART1 can handle. Files are nothing but variables being taken one step further. We can think of a file as a list which holds recurring arrangements of variables.

To understand this a little better, let's imagine a situation where we want the SMART1 to keep track of a company's 250 customers. It's supposed to store the customers' names, the category of goods they are buying and the amount of money the company has received from each of them so far.

We could just go ahead and declare a whole bunch of variables called something like NAME1 to NAME250, GOODS1 to GOODS250 and MONEY1 to MONEY250. Aside from the fact that doing this would probably take us longer than to read the rest of this manual, it would be a very impractical approach to this problem, because to access the variables we would also have to construct the same program sequences over and over again, each time with a different variable name.



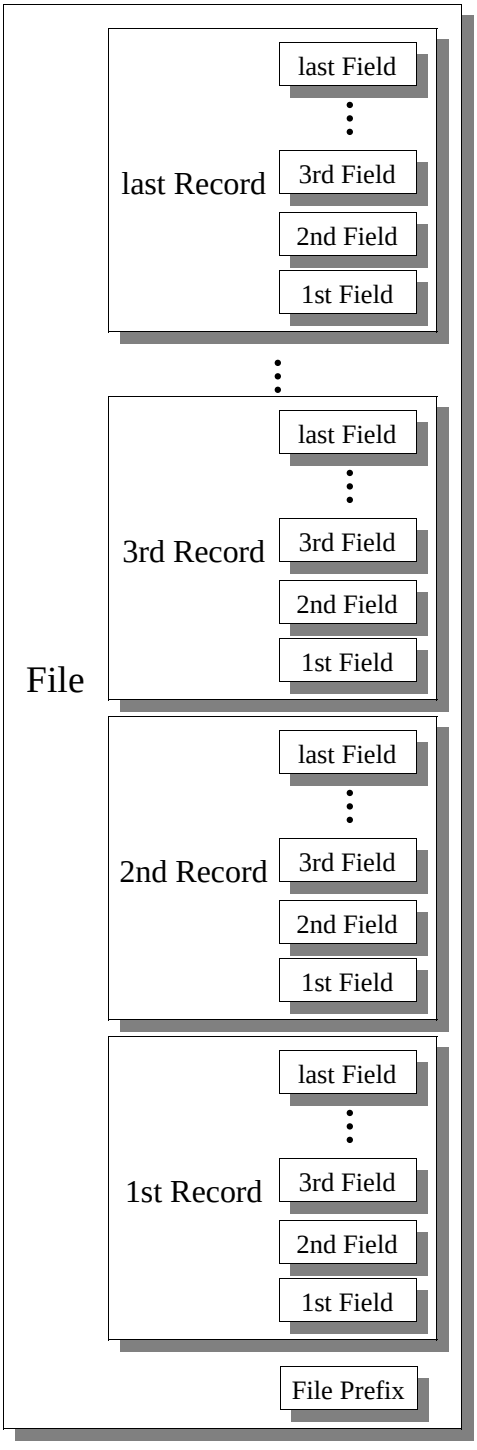
A much better way is to build a file. Clicking on the node image of a *File Definition* instruction will open up the window shown on the left. The data inside a file is not stored in variables. The term we use in a file is *Field*. Fields are very much like variables. When we add a field to our file, a *Field Declaration* window pops up. We declare fields almost exactly the same way as we declare variables. We have to give each of them a name according to the same rules that apply to naming variables and program modules. And, we have to specify whether the field is going to be of the *Number Type*, the *String Type* or the *Character Type*. There is, however, one difference between declaring a field and declaring a variable. If a field is of the *Number Type* then we can not enter a value for its maximum length or its decimal places. The reason for this is, that fields are never displayed directly. But, let's not get into this right now, it will be explained in the section about the *File Access* instruction.

Once a field has been declared, the information about it is shown in the file definition window. By clicking the *Add New Field* button we can add another field. The maximum number of fields that we can declare for one file is 16. Each of the fields must have a name that is different from all other fields in that file. Nevertheless, we are allowed to re-use the same field names in another file. It is also possible to name a field the same as a variable.

Clicking the mouse on a field description will highlight it. If we need to, we can now change the field information by pressing the *Edit* button. Pressing the *Delete* button will remove the field from our file.

The Structure of a File

We have learned that the *Field* is the smallest element of a file. The entirety of fields that we declare in a file definition makes up the next greater component, the so-called *Record.* A file can consist of many records. They all have the same structure. All fields are repeated in the same order for every single record throughout a file. The file definition window permits us to type the number of records we want our file to keep into the input box labeled *max # of Records.* In our example on the previous page, we entered the number 250, since the file was supposed to handle 250 customers. At compile-time the SMART1 programmer does the necessary steps to reserve the memory needed for the file. However, entering the number alone does not guarantee that we are really going to get as many records. The device has a finite amount of memory, and therefore, we will probably have to make a compromise in one or the other of our projects.



Whether or not the SMART1 can actually supply enough space for all the records we want, depends on several factors. First of all we have to regard the lengths of our fields. Just like a variable, a field takes up a certain number of bytes. *Character Fields* are perfectly content using only one byte, *Number Fields* will always occupy four bytes each and a *String Field* requires the equivalent of its declared maximum length plus 1 byte. If we add up the bytes of all fields in a record, then we know how much space we need to store this one record. If we then multiply the sum by the desired number of records, we get the space that is required for all the data in our file. In addition to this, every file needs some overhead space. It's called the *File Prefix* and it always absorbs eight more bytes. But, there is more. Files share the same memory pool with all variables we employ in a project. If we take all that into account and hold it against the 28000 bytes of total available (RAM) memory, we can figure out if there is enough room for our file or if we have to compromise.

Luckily, we don't really have to do all these calculations when we are building a file. We can get away, using the trial and error method. The SMART1 Programmer will let us know through a message, if our wishes tend to be a little on the greedy side. For the 250 customers in our example, however, we should have plenty of space:

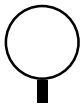
(20+1	+ 1	+ 4)	* 250	+ 8	= 6508
String	Character	Number	Records	File Prefix	total bytes

Instructions for Program Flow Management

Module Start


Selector Button

Node Image

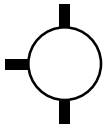

Node Type

Every executable program module must begin at some point. To define this start point, we have to place a *Module Start* instruction in our module. This instruction can and must be used only once in each executable module. Its node has no entries and only a pass exit. The *Module Start* instruction has no actual code assigned to it and therefore we don‘t need to program it. It merely acts as a pointer to the first instruction we want the SMART1 to execute every time this program module is called from anywhere in the project.

Call a Module


Selector Button


Node Image

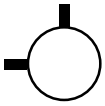

Node Type

The best modular program structure wouldn’t be any good without a command to actually call a module. That’s why we have the *Call a Module* instruction. It causes the current program module to be suspended and the program flow to continue in the module specified by the instruction. As soon as the called module has finished its task the program resumes with the instruction that is tied to the node’s pass exit. The node of a *Call a Module* instruction has no fail exit. We program this type of instruction, like all other programmable instructions, by selecting the node and pressing *ENTER* or by clicking on it using the right mouse button. The only programming required for this instruction is to type the module’s name. It will appear in the node image.

Module Stop


Selector Button

Node Image


Node Type

The *Module Stop* instruction is used to end the execution of a program module. Whenever this instruction is encountered, the current module is abandoned and the program returns to the point after the instruction that invoked the module in the first place. In most cases this would have been a *Call a Module* instruction in another module. If, however, the *Module Stop* instruction is used in the project’s start up module, then the application will quit and the SMART1’s power up screen is displayed, unless the project was compiled for immediate execution, in which case it would start right up again.

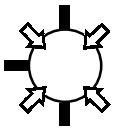
The node of a *Module Stop* instruction has neither a pass nor a fail exit and since its purpose never changes, we don’t have to program it either. There is no maximum number of *Module Stop* instructions we can have in a program module. And if we don't want to exit a particular module at all, for example the main menu of an application, then there is no need to even have a single *Module Stop* instruction.

Sometimes we will find it necessary to repeat a set of instructions inside a module until a certain condition is met. At other times specific circumstances may require to skip a part of the program in a module. It could also be the case that we simply want to place a set of instructions in a particular area of a program module, be it out of either practical or esthetical motivations. To accommodate all those instances we can resort to the following two instructions which go hand in hand.

Re-Entry Point


Selector Button


Node Image

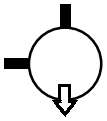

Node Type

This is the instruction we would put at the place in a module which we want to go back or forward to. The *Re-Entry Point* is not a programmable instruction. It has a pass exit but no fail exit. The number it bears gets assigned to it automatically in sequential order starting with 1. Its value is of no relevance for the order in which *Re-Entry Points* are executed. It merely serves as a distinctive label. Each new *Re-Entry Point* receives the next available number. In case we decide to erase a *Re-Entry* Point instruction from our module then all the ones labeled with a higher number, as well as their associated *Jump* instructions, will decrement their value by 1 to close the gap in the sequence. We can tie a *Re-Entry Point* instruction into the program flow just the same way as other instructions by connecting to one or both of the node's entries. However, for this instruction the usage of the ordinary entries is not mandatory. A *Re-Entry Point* instruction will be invisibly connected to the program flow by its remote entries as long as at least one *Jump* instruction points to it.

Jump


Selector Button


Node Image


Node Type

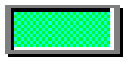
The counter part to the *Re-Entry Point* instruction is the *Jump* instruction. We use it to tell the program at which *Re-Entry Point* in the module to continue. This instruction's node has the two entries but neither a pass exit nor a fail exit. To program it, we have to click the right mouse button and enter the number of the *Re-Entry Point* we want to jump to. There is no maximum number of *Jump* instructions to direct the program flow to the same *Re-Entry Point.* We can have as many as we need.

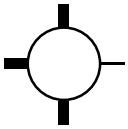
Both the *Re-Entry* Point instruction and the *Jump* Instruction are local to their module. It is not possible to jump from one program module to another. To invoke another module, the *Call a Module* instruction must be used.

Instructions for I/O Components

The Display


Selector Button

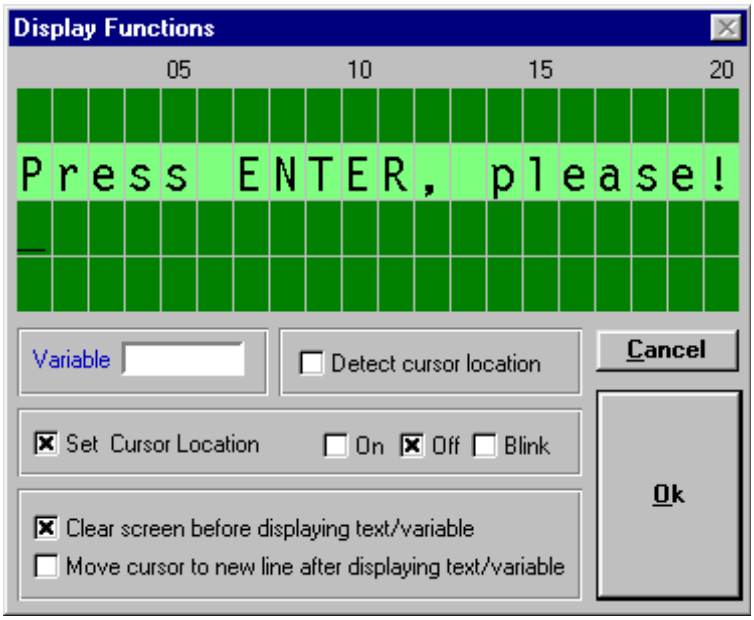

Node Image


Node Type

The display supports a total of seven different instructions. Each of them can be used either alone or cooperatively with one or more of the others. We mainly use the display to give information or commands to the SMART1's operator. For this purpose we use the *Text display* instruction and the *Variable display* instruction. With the *Detect cursor location* instruction which is the only display instruction that results in a node with a fail exit, we can also get input from the display. The remaining four instructions are of an auxiliary nature. Let's look at the main ones and bind in the other instructions when appropriate.

Text Display

A node programmed with the text display instruction has no fail exit.

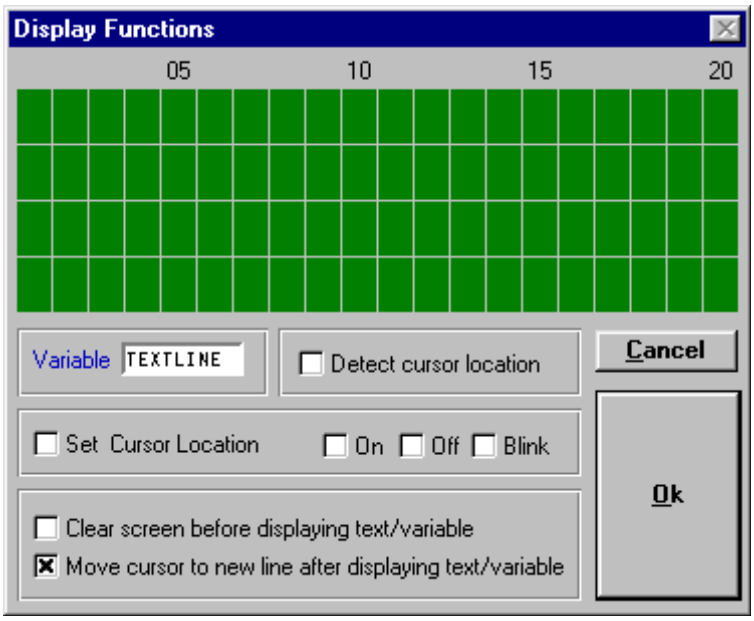


To display a text we simply click anywhere on the green grid that looks like the SMART1's display and start typing. In our example we write `Press ENTER, please!` The text appears beginning in the top left corner. This does not necessarily mean that it will appear in this location on the real display. The starting point is solely determined by the location of the cursor, even if we cannot see it. Unless we know where the previous display instruction left the cursor, we will not be able to tell where our text is going to show. To be sure about the positioning of the text we must use the instruction labeled *Set cursor location*. If we right-click anywhere on the display now, the start of the text will move to that place. When there is not enough room on one line the text will wrap around to the next line or even to the top of the display. Let's click on the first box in the second row to make our display look like the illustration. Well, we still can't be entirely sure that this is what we are going to see when the program is running. Only the spaces which are highlighted are actually being written to. The remaining area of the screen will keep displaying the same characters that were there before. This can easily be fixed by invoking the *Clear screen* instruction to erase everything prior to displaying the text. The last thing for us to take care of is the cursor mode. We don't need to display a cursor with this particular message, so we put a check mark in the box for the *Cursor off* instruction.

The SMART1 can display almost every character available on the PC keyboard. There are only two exceptions. The backslash \ will be shown as the sign for Yen ¥ and where we enter the wavy line called tilde ~ , a right pointing arrow is going to appear on the display.

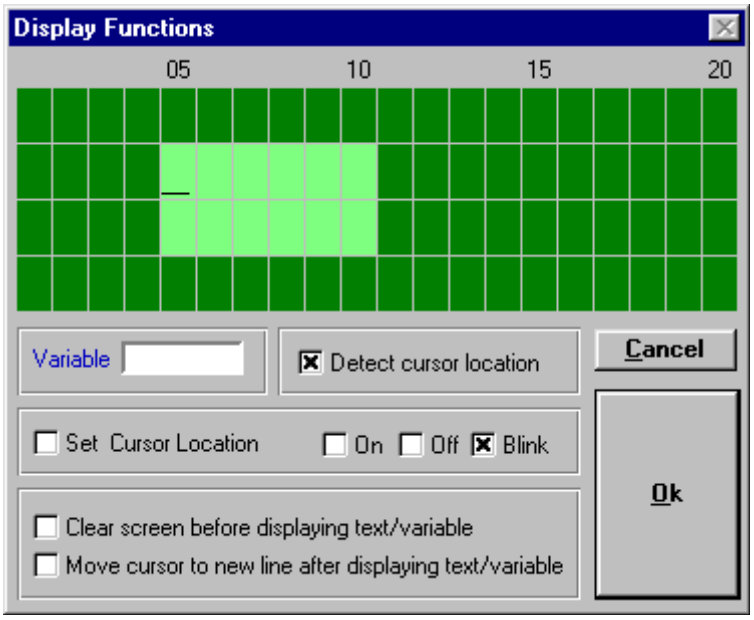
Variable Display

This is the display's second main function. Its purpose is to display the content of a variable. We can use this instruction for each of the three *Variable Types*. Like the text display, the variable display can be supplemented with one or more of the auxiliary instructions. We can *Clear the screen*, *Set the cursor location* and define the *Cursor mode*. As shown in our example picture, we can also move the cursor to the *Next line* after displaying the variable. A node programmed as variable display instruction does not have a fail exit.



Detect Cursor Location

The last one of the display's main instructions is rather unique. It is an input function and produces a node with a fail exit. We can employ this instruction to find out whether the cursor is situated inside or outside a specific area of the screen. Out of all the options, the only thing we can do in conjunction with this instruction, is to specify the *Cursor mode*. The instruction's primary application is the cursor menu. How it works is explained in the chapter about the keypad's *Moving cursor* instruction. To define the area in which we want to scan for the cursor, we use the mouse. Clicking with the left button on the display grid marks one corner of the array. The opposite corner is selected by clicking the right mouse button. The size of the zone can range from a single character space to the entire display, but it can only be rectangular. Light green blocks show its extents. At run time the area will not be highlighted or otherwise indicated on the screen.

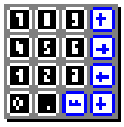


If the cursor is located inside the search area when the instruction is executed, the pass exit is used, otherwise the fail exit is used.

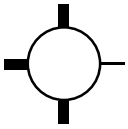
The Keypad

The SMART1's keypad can be programmed for one of four input instructions.

Selector Button



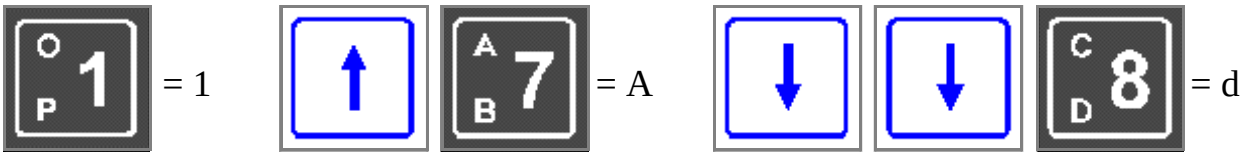
Node Image



Node Type

Character Input

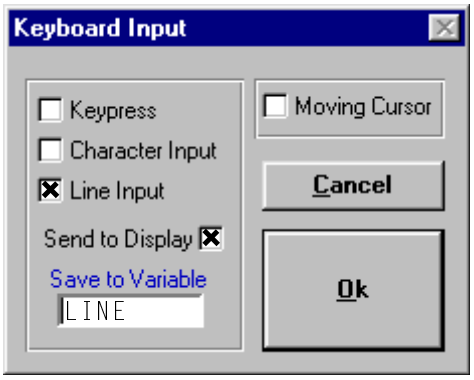
The most basic one of the Keypad instructions is *Character Input*. A node programmed with a character input instruction has no fail exit. Execution is halted until a character is recognized. We use this instruction whenever we want the program to suspend further operation and wait for the operator to press a key. To specify the character input instruction we must click on the character input check box. If the *Send to Display* box is checked then the character will be shown at the present cursor location on the display. We also have the opportunity to save it to a variable for evaluation later in the program. Naturally this variable must be of the Character type. The most common use for the character input instruction is perhaps in a menu structure. Let's say we have programmed the display to show a list of choices for the operator; each of them with an associated character. In our keypad instruction we have specified to save the character to a variable. Now the program can use the value of the variable, i.e. the character that the operator has chosen, to decide what to do next. All capital letters, all small letters, all single digit numbers, the dot, enter, space and backspace are recognized by the character input instruction. But, since there are two letters and a number on most of the buttons, how do we tell the SMART1 which character we want to enter? Well, it is not complicated at all. Whenever we press a button by itself, its main character, which is the big one in the center, is selected. This way we get the numbers, the dot and the enter key. Trying to display the enter key would result in the cursor being sent to the beginning of the current row. The right arrow gives us a space and the left arrow a backspace character. The latter is not going to appear at all if we were to send it to the display from a character input instruction. The up and down arrows do not return a character. They are used to switch the keypad into letter mode. To get a capital letter we have to visually locate it first. If it sits in the upper part of the button we would press the up arrow and then the button with the letter. For a letter in a button's lower area we would use the down arrow instead. To enter a small letter we simply press the corresponding arrow key twice. Here are some examples:



If more than two seconds have elapsed between pushing the up arrow or the down arrow and pressing a letter key then the letter mode has expired and the button's main character is returned.

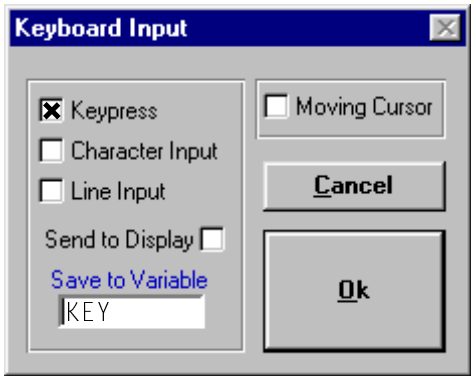
Line Input

The *Line Input* is based on the character input. A node programmed with this instruction has no fail exit. The program continues through the node's pass exit after the *Line Input* is completed. We can employ *Line Inputs* to enter words, multi digit numbers or any combination of characters. A line input instruction is indicated by a mark in the *Line Input* check box. The *Send to Display* option is automatically checked when *Line Input* is used and we can't turn it off. Again, as with the *Character Input*, we have the possibility to store what we enter in a *Variable*. In this case our variable may be either of the *String Type* or the *Number Type*. When a *Line Input* is executed the procedure is as follows: The SMART1 waits for the operator to type in a character. As soon as he does that the character is displayed at the current cursor location and the cursor advances to the next place. In case the character was a backspace and the line was not empty, then the cursor moves to the left and wipes out any character that was there before. This gets repeated until the ENTER key is pressed. If we are saving the line input to a variable of the *Number Type* then the SMART1 checks if the operator has actually entered something that qualifies to be called a number. If, for example, he mixed in a letter or entered two decimal points, then the *Line Input* is rejected. The cursor jumps back to the first character and the operator can try again. The maximum number of characters a *Line Input* instruction can process, depends on the variable we are saving to. We can only enter as many characters as the length of the variable allows. However, there is never a minimum number of characters. We can always hit the ENTER button by itself. An empty *Line Input* results in an empty string or, if we are saving to a *Number Variable*, the number will be 0.

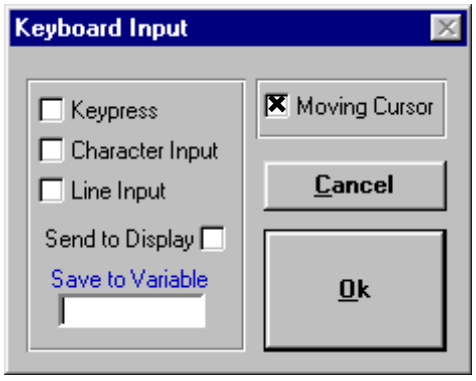


Keypress

The *Keypress* instruction is the only one of the four keypad instructions that produces a node with a fail exit. We can use it to detect if a key was pressed, while the SMART1 was busy doing something else. The Keypress instruction does not cause the program to halt. It merely directs the flow to different paths depending on whether the operator had pushed a button or not. If a key was pressed then the pass exit is used. Otherwise the program continues through the fail exit. The *Send to Display* check box is not available with this instruction. Unlike the *Character Input*, the *Keypress* instruction can not handle letters. It returns only the "raw key code" for the pressed button. The number buttons give us the number and the dot is shown as dot. The raw code for enter is a capital **E**. The arrow buttons return the starting letter of the word that represents the direction the respective arrow points to: **U**p, **L**eft, **R**ight and **D**own. We can save the raw key code in a *Character Type* variable.



Moving Cursor



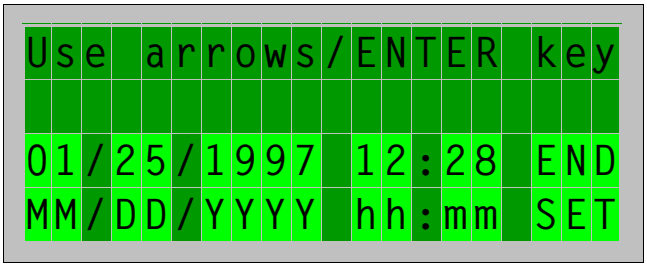
A special type of input is the *Moving Cursor* instruction. If a node is programmed as *Moving Cursor* then it has no fail exit. The program halts until the function is completed. Neither the *Send to Display* option nor the *Save to Variable* possibility are relevant with the moving cursor instruction. The only functioning keys in a moving cursor instruction are the arrow keys and ENTER. Pressing an arrow key causes the cursor to move one step in the indicated direction unless it has already reached the end of the display. The instruction is ended when the operator presses the ENTER key. The moving cursor is mostly used in conjunction with the display's *Detect cursor location* instruction to accomplish a cursor menu input. What is a cursor menu? Due to the restricted space on the SMART1's liquid crystal display we will not always be able to put all menu items we would like on the screen and still have enough room left to tell the operator which button to press for each of them. The solution is to list only the choices and let the operator "go" to the one he wants by moving the cursor there and pressing ENTER.

We can see an example for a cursor menu when we run the Set-date module, which is included in the SMART1 Programmer module library.

The cursor is moved to the desired menu item using the four arrow keys. Pressing ENTER invokes the programmed action for the particular menu item.

There are seven menu items in this example (END and SET are different items), each of which is being scanned in a separate Detect cursor location instruction after the Moving Cursor instruction is finished.

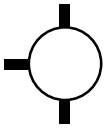
Naturally, the real display won't have dark and light areas to show the active menu items.



Setting up a Serial Port

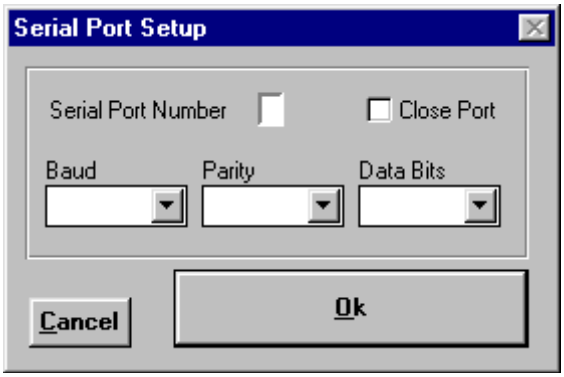

Selector Button


Node Image


Node Type

Before we can use the SMART1's serial ports to receive from or send to serial communications devices, the ports' parameters must be configured to match the settings of the connected equipment.

Each of the ports we want to use in a project must be configured separately. By right clicking on the image we open the *Serial Port Setup* window. We can specify the *Serial Port Number* and the values of the parameters for the serial communication. These are *Baud* rate, *Parity* and the number of *Data Bits*. The values for all parameters depend solely on the settings of the device we will be connecting to. They are usually shown either on the device itself or in its operator's manual. The *Baud* rate is sometimes referred to as *BPS* or *Bits per second*. It is the value for the speed at which the communications runs.



From the moment when a serial port's settings are configured, it is ready to receive and transmit data. Each port has a memory buffer associated with it, capable of storing up to 256 incoming bytes (characters). If we want to disable an already configured serial port, then we must click the check box labeled *Close Port*. This will keep it from receiving any more data until we apply the setup function again.

The five serial ports of the SMART1 are not identical. Ports 0, 3 and 4 are equipped with a signal line (RTS) which tells the connected device whether the port is ready to receive data. Ports 1 and 2 do not have this feature.

As far as the parameter setup goes, ports 1 through 4 share essentially the same characteristics, but port 0 is very different. When we enter the Serial Port Number, the parameter menus change, allowing us to access only the values that are available for the particular port. To specify a parameter we have to click first on the appropriate menu and then choose one of the items in the fly-out list by clicking on it. The supported parameter values for the individual ports of the SMART1 are as follows:

Baud (bits per second)							
Port 0		1200	2400	4800	9600	19200	38400
Port 1,2,3,4	300	1200	2400		9600		

Parity			
Port 0	none		
Port 1,2,3,4	none	odd	even

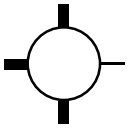
Data Bits		
Port 0	8	
Port 1,2,3,4	8	7

A *Serial Port Setup* node has no fail exit. The program will always continue with the instruction tied to the pass exit of the *Serial Port Setup* node.

Serial Communication


Selector Button


Node Image

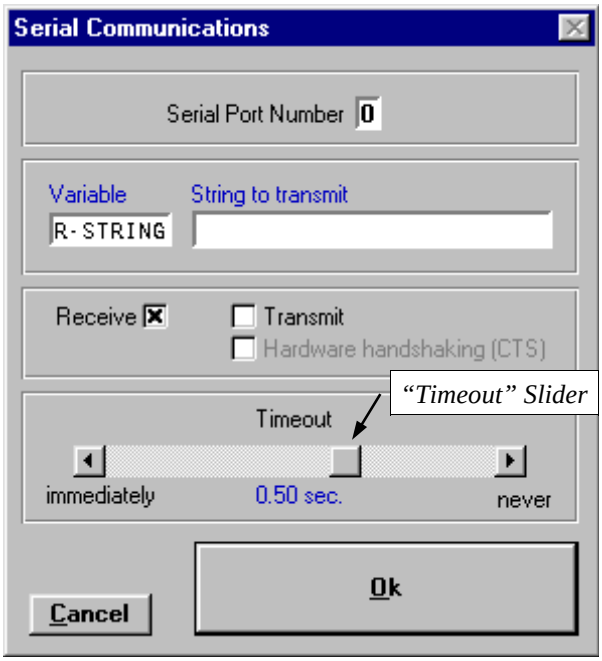

Node Type

Once we have configured a serial port with the help of the *Serial Port Setup* instruction, we can send or receive data using the *Serial Communication* instruction. We must specify the port to be used in the input box labeled *Serial Port Number*. This instruction demands caution from us. **Attempting a communication through a port that hasn't been set up will produce an unpredictable result** and most likely "hang" further execution of our project.

We can apply this instruction type to accomplish one of two different tasks. We are either expecting data from an outside source, in which case we would click the *Receive* check box, or we are going to send information to another serial device. The latter operation we must indicate by checking the box labeled *Transmit.*

Receive Mode

If we are using the *Receive* mode, we are required to name a variable in which the incoming data is going to be stored. Only *Character Variables* or *String Variables* are allowed in this situation. *Number Variables* can not be used as a target for a *Receive* function. The *Timeout* slider bar in the lower third of the programming window lets us set the time that we want the SMART1 to wait for the data to arrive. Either by clicking the buttons on the sides or by moving the slider itself using our mouse cursor, we can adjust the timeout value from *immediately* to *0.03 sec. - 0.06 sec. - 0.13 sec. - 0.25 sec. - 0.50 sec. - 1.00 sec. - 2.00 sec.* to *never*.



If the variable we specified happens to be a character, then the SMART1 will wait until one character comes in the serial port or until the time expires. In case a character was received it will be stored in the variable and the program will continue at the node's pass exit. If, however, a timeout occurred before a character was received, then the node's fail exit will be used and the content of the variable will remain unchanged.

The situation is somewhat different when we specify a *String Variable* as target. Its contents are always cleared at the beginning of a *Receive* instruction. Then the SMART1 takes each character it receives from the serial port and appends it to the end of the string. This goes on until either one of the following three conditions becomes true. When the string's capacity is reached, meaning as many characters as the *Maximum Length* of the *String Variable* were received, the receiving function is fulfilled and the program resumes at the node's pass exit. The second condition under which the function is considered to be complete, is when a *Null Character* was received. This special character is

always deemed the end of a string. The only circumstance leading to a failed instruction, while receiving into a *String Variable*, will be a timeout. It is going to occur when the time in between receiving of two characters was longer than we allowed for in the timeout setting. In such a case the instruction tied to the node's fail exit will be executed. The *String Variable* will contain all characters that were received up until the timeout.

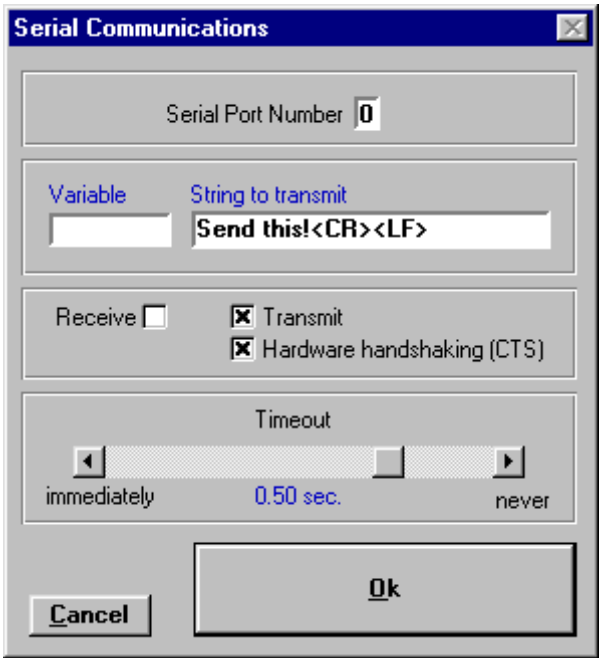
Transmit Mode

The *Transmit* mode provides us with a lot more choices. First of all, there is no restriction on the type of variable we can use with it. The contents of *Character Variables* or *String Variables* are transmitted as they are. A *Number Variable*, on the other hand, is automatically converted into a string before being sent out. For the formatting of this string the SMART1 applies the number's *Length* and *Decimal Places* settings.

Another option we can take advantage of with the *Transmit* procedure is, that we don't even have to use a variable at all. We can simply type the data we wish to send into the input field titled *String to transmit*. For this we may use and mix all the different methods of entering character values.

A feature that is also available only in conjunction with the *Transmit* function is the *Hardware handshaking (CTS)*. What this option does, is to first test a signal line coming from the device we're sending to. The transmission of data will commence only after this line says "clear to send". Well, at least for serial port 0 this is true. Ports 1 through 4 will report a missing handshake signal to the *Serial Communication* instruction only after the first character has been sent. This is due to the fact that the chip in charge of those four ports has a built in 1 byte transmit buffer for each channel in which it keeps the character until such time as it can be transmitted.

At any rate, if the time we've set on the *Timeout* bar has expired before the SMART1 was able to send the next character, then the instruction is going to fail and the program will choose the path connected to the node's fail exit.



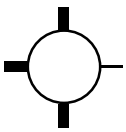
When we are programming a *Transmit* function, then the *Timeout* slider bar will be available only if the *Hardware handshaking (CTS)* box is checked. The timeout values are in this case slightly different from what we've read about the *Receive* mode timeouts. They range from *immediately to 0.02 sec. - 0.03 sec. - 0.06 sec. - 0.13 sec. - 0.25 sec. - 0.50 sec. - 1.00 sec. to never*.

The node of a *Serial Communication* instruction will develop a fail exit only in those cases where the *Timeout* slider is enabled and set to a value other than *never*. This applies for the *Transmit* mode as well as for the *Receive* mode.

The Clock


Selector Button


Node Image

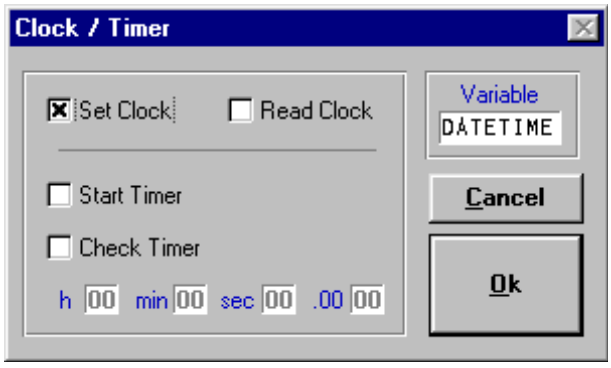

Node Type

The SMART1's time keeping circuit can be looked at in one of two ways. To set or read the current date and time values we can treat it like a *Clock*. If we need to measure time durations, then we can handle it as a *Timer*. Among the two different modes we are being offered four instructions to choose from. All of them require the use of a *String Variable* which must have a length of at least 16 characters. The style in which data is transmitted to and received from the clock chip is somewhat similar to the string that gets sent out by a scale indicator. Of course, instead of weight, units and other scale related information the clock's string has fields that hold information about date and time. They range from year to month to day and so on all the way to hundredths of a second. This is the string's structure:

	year	month	day	weekday	hour	minute	second	.00
	Y Y	M M	D D	W W	h h	m m	s s	0 0
example:	9 7	0 7	2 9	0 3	1 8	4 1	2 3	5 8
valid range:	00..99	01..12	01..31	01..07	00..23	00..59	00..59	00..99

Set Clock

This is one of the clock mode instructions. We can use the *Set Clock* instruction to change the unit's date and time. When we call this instruction we have to supply a string that contains characters according to the structure mentioned above. This means, we must always submit data for each of the fields even when if we only want to change one of them and leave the others as they are.



The clock will not be ticking in a SMART1 that is being programmed for the very first time. It has to perform the *Set Clock* instruction at least once to get started. From then on it keeps going and going and .....

When we program a node with a *Set Clock* instruction it will have a fail exit. It is going to use this fail exit, if the string we told it to send is making no sense to the clock chip.

The collection of modules in the SMART1 library folder (directory) includes one called "Set-date.mod". It's Western Scale's suggestion on how to set the date and the time. It goes hand in hand with either one of the many modules that have names like "Ddmmmyy.mod", "Mmddyy.mod" or something similar. They read the clock and format the information in different ways to accommodate the various preferences that people have for displaying date and time.

Read Clock

This instruction receives data about time and date from the SMART1's internal clock. Again, we have to supply a string with a capacity to hold at least 16 characters. The content of the string doesn't matter in this case. The *Read Clock* instruction will overwrite it anyhow. After completion of the instruction the string will be formatted as described on the previous page. Since a node programmed with a this instruction has no fail exit, the program will continue via the pass exit.

The *Read Clock* instruction returns the time in 24 hours. The two characters that stand for the hour can range from '00' to '23'. For everybody who is more familiar with 12 hours a.m and 12 hours p.m., the tables below list the hours of the day in 12 hour mode and in 24 hour mode.

a.m.												p.m.											
12	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3	4	5	6	7	8	9	10	11

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23

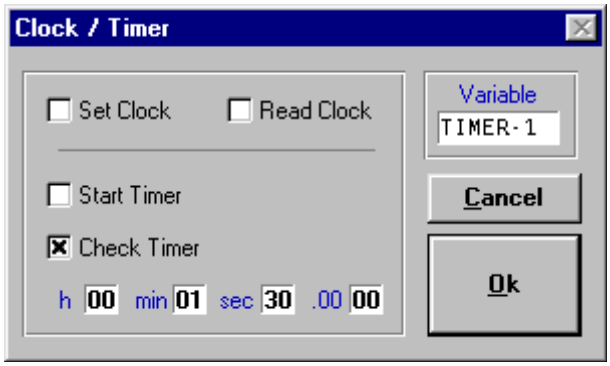
Within our assortment of library modules there is one that does a pretty good job performing these translations of hours. It's called "Am-pm.mod". It works best in conjunction with "Mmddyy.mod" or any other one of the program modules that read the clock.

Start Timer

Let's look at the clock in timer mode. Similar to the one in our microwave oven that keeps the heat on for a preset number of seconds, we can also have a timer in our SMART1 program. Actually, we can have many timers, and they will be working independently from one another. In contrast to the oven timer, here we don't specify the length when we start a timer. The *Start Timer* instruction saves information about the current time in the specified variable. If we need another timer while the first one is still running we would create the new timer simply by using the instruction with a different variable. Like the *Read Clock* instruction, a node programmed with a *Start Timer* instruction has no fail exit.

Check Timer

For this instruction we need to specify the duration we want to check for. We do this by entering numbers in the input fields for h, min, sec and .00 (hundredths of a second). At the moment the *Check Timer* instruction is executed the current time is tested against the supplied variable. This variable must have been set before using a *Start Timer* instruction. If since then a time longer than the specified duration has expired, the pass exit is used. Otherwise the program continues through the node's fail exit.

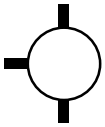


The module "Delay.mod" in the library folder demonstrates one way to achieve a flexible timer.

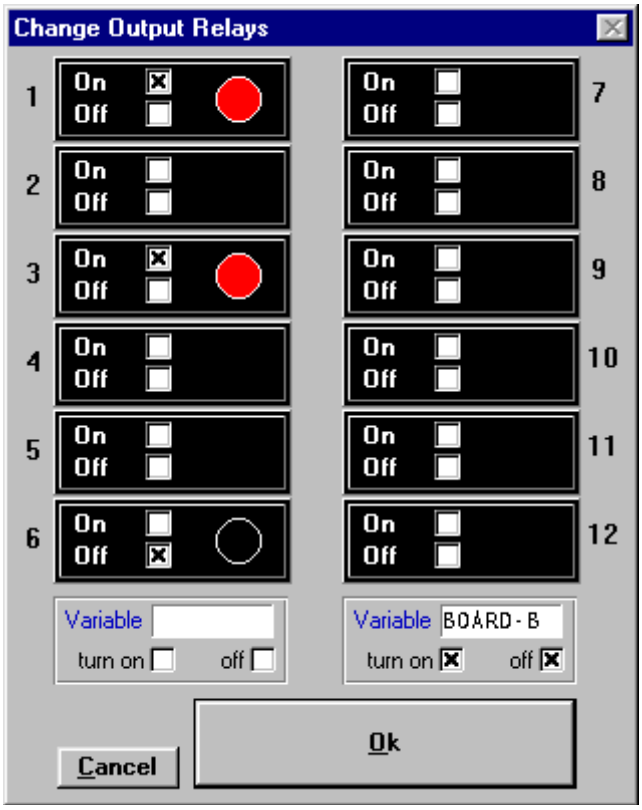
The Output Relays


Selector Button


Node Image


Node Type

This instruction type enables the SMART1 to control the relays on one or two of Western Scale's *Setpoint Module Output Rack* Boards. Each of the boards can hold 1 to 6 relays capable of switching either AC or DC currents of up to 3.5 Amps. The Boards must be connected to the SMART1's I/O expansion port through a suitable cable. (We can find a sketch in the appendix.)



The instruction's programming window lets us decide between two different ways to switch the relays. First, there is the direct method. If we want to turn a particular relay on or off, we simply click its *On* check box or its *Off* check box. This technique was applied to the relays on the left side of our example. This row represents the relays located on output board A, whereas relays numbered 7 through 12 are associated with output board B. The example shows relays 1 and 3 to be turned on, and relay 6 to be turned off. A red light is shown on every relay that's going to be switched on. On relays being switched off, the light is black. The ones with no light at all will remain in whatever state they are, if we are using the direct method.

The other alternative we can apply to change our relays involves the use of a *Character Variable*. At run time the pattern of the variable's lower 6 bits will be imposed onto the relays. Using this method, we don't know at programming time what the status of a relay is going to be. Therefore no lights are displayed on the relays. Depending on the check marks in the boxes labeled *turn on* and *off*, one of the three procedures shown in the table below will determine how the content of the specified variable is going to affect the relays. The content

0 irrelevant	1 irrelevant	**0**	**0**	**1**	**1**	**0**	**1**	bits
turn on ☒	off	-	-	**ON**	**ON**	-	**ON**	1 bit = on, 0 bit = no change
turn on	off ☒	-	-	**OFF**	**OFF**	-	**OFF**	1 bit = off, 0 bit = no change
turn on ☒	off ☒	**OFF**	**OFF**	**ON**	**ON**	**OFF**	**ON**	1 bit = on, 0 bit = off

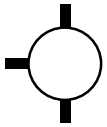
relay number	6	5	4	3	2	1	board A
relay number	12	11	10	9	8	7	board B

Due to the fact that there's no decision being made in a *Change Output Relays* instruction, its node does not have a fail exit.

The 4-20 mA Outputs

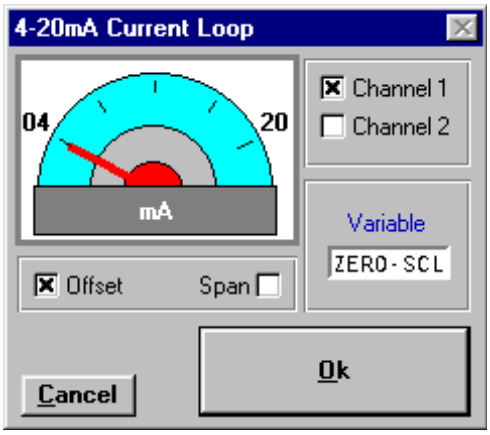

Selector Button


Node Image


Node Type

We have three instructions for the 4-20 mA Outputs. Each of them requires a variable of the *Number Type* and they all produce a node with no fail exit. To select the channel we want the SMART1 to manipulate, we must click either on the check box labeled *Channel 1* or on the one labeled *Channel 2*. The receiving device(s) must be connected to the SMART1's I/O expansion port via Western Scale's *4-20 mA* Boards.

Offset Calibration



Before we can successfully use a 4-20 mA device with the SMART1 we have to conduct a calibration. This is the process in which we adapt the SMART1's respective 4-20 mA channel to the receiving device. In case of the *Offset Calibration* instruction the variable we're supplying must contain the value at which we want to generate an output current of 4 mA. For a project that utilizes the current to reflect a weight, this value will probably be 0. Watching the readout of the device connected to the 4-20 mA Output channel, we now must adjust the current to a value of 4 mA. To do this we can use the four arrow keys on the SMART1's keypad. Once we are happy with the output current, we can press the ENTER key to save the offset value and leave the *Offset Calibration* instruction. Here is how the arrow keys alter the output current:

arrow key	→	↑	←	↓
pressed once	tiny increase	big increase	tiny decrease	big decrease
held down	modest increase	giant increase	modest decrease	giant decrease

Span Calibration

We must apply the same procedure as for setting the offset also to adjust the span. This time, though, our variable must contain a value that will be the equivalent for 20 mA (in a scale application most likely the maximum weight). Again, we are pressing our arrow buttons, only this time until we get an output current of 20 mA. The ENTER key concludes the instruction.

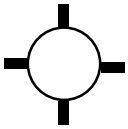
Changing the Current

If neither *Offset* nor *Span* are checked, then the SMART1 will simply change the current of the specified channel to represent the number that was submitted as variable content. The instruction uses the previously stored settings for offset and span to calculate the output current. The current will then be held at a constant level until another instruction of the 4-20 mA type is executed.

The Remote Inputs


Selector Button

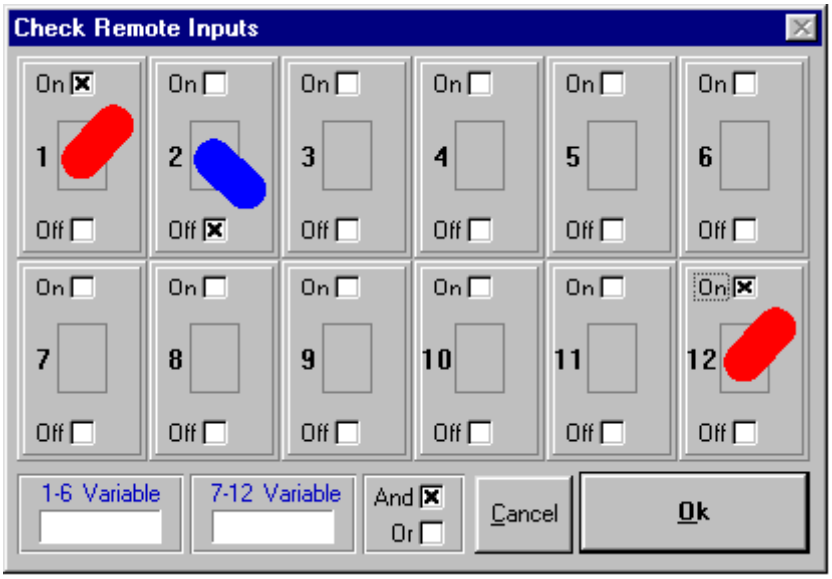

Node Image


Node Type

This is the instruction type that gives our programs the capability to make decisions based upon the status of remote switches. The switches' contacts must be connected to the SMART1's I/O expansion port using one or two of Western Scale's *Remote Input Boards*.

A node programmed with a *Remote Inputs* instruction always has a pass exit and a fail exit. The decision about which one of the exits gets used is made at the time of execution by **comparing the remote switches' actual status** to a **pattern** that we set in the instruction's programming window. The pattern tells the SMART1 which ones of the switches we expect to be **On** (contact closed) and which ones we expect to be **Off** (contact open).

When we click on a check box a lever either pointing up or down is displayed. Only switches that are marked one or the other way are taken into the pattern and will be checked at run-time. All other switches have no influence on the outcome of the instruction. To remove a switch from the pattern we simply click on the one of its check boxes that is activated.

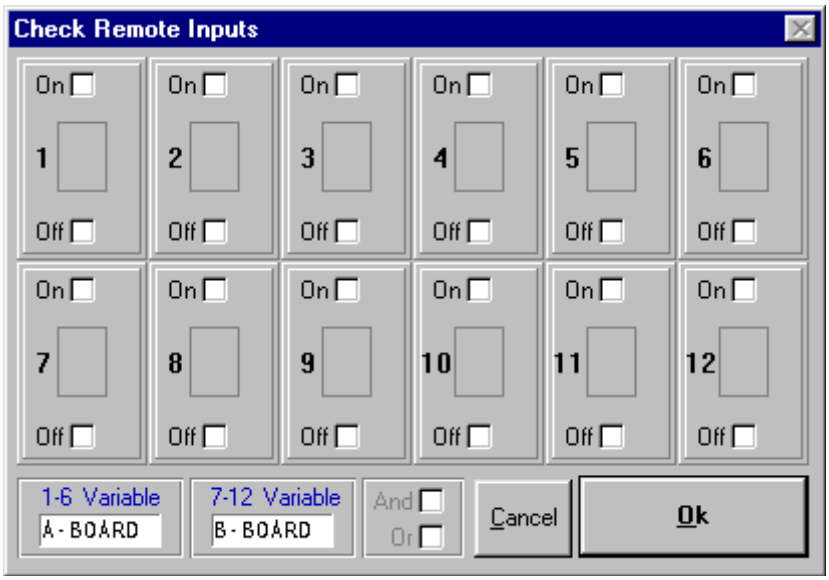


Let's assume for a moment that the only switch we had marked was switch 1 and that we expect it to be in the on position. If at time of execution the contact that is connected to remote input 1 is closed, then the program will continue through the pass exit of our instruction, otherwise it will follow the path at the fail exit. Checking only one switch and finding out whether it is on or off is the simplest application for the *Remote Inputs* instruction and quite possibly the one we will be using the most. But there is much more that we can do with it.

The programming window in the above example shows switches 1,2 and 12 in various states as well as an activated check box labeled *And*. It got activated automatically when we clicked on the second switch. This check box determines the logical operation that will be applied to evaluate the inputs if more than one switch is to be checked. **And-logic** means that **every one** of the switches in our pattern must be matched exactly by its associated remote input. Only if this is the case, then the program will proceed via the pass exit. If any of the remote inputs doesn't match its counterpart in our pattern, then the instruction's fail exit is employed. In our example switch 1 must be on and switch 2 must be off and switch 12 must be on.

As an alternative to using and-logic we can click on the check box labeled *Or* and select **Or-logic**. Then our instruction will pass as long as **at least one** of the switches in our pattern is matched by its respective remote input. The instruction will fail only if none of them match.

Storing the Status of Remote Inputs



In addition to comparing the remote switches to a pattern we can also save their status in character type variables for future operations. Actually, we don't even have to specify any switch patterns if all we want to do is to capture the actual status of the *Remote Inputs* in variables. We must use one variable to store the information of inputs 1 through 6 and another variable for inputs 7 through 12.

When we save the status of a *Remote Inputs* board to a variable, then all six inputs of that particular board are scanned and saved. The information is stored in form of a bit-map. The following paragraph explains the details. This will perhaps make sense only to the people among us who have had their share of exposure to low level programming. Everyone else may just skip those lines and the table. Remember, though, that there was something written about this topic and if an application really needs this kind of stuff, then come back, read it and try to figure it out.

Given that a character variable is nothing else but a byte, we have 8 bits to deal with. Each of the six remote inputs gets one bit assigned to it and the remaining two bits are held in a fixed state. If the particular input is on (closed contact) then its bit will be 1, if the input is off (open contact) then the associated bit will be 0. If a remote input has no switch hooked up to it, then it is treated like an open contact and its bit will be 0. The bits are mapped as shown in this table:

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
board A	**0**	**1**	**input 6**	**input 5**	**input 4**	**input 3**	**input 2**	**input 1**
board B	**0**	**1**	**input 12**	**input 11**	**input 10**	**input 9**	**input 8**	**input 7**

An attempt to save the status of a not connected *Remote Inputs* board into a variable, will cause the instruction to fail. Under such circumstances the content of the variable is not going to change and the program will continue via the node's fail exit.

The instruction will also fail every time we try to compare a switch pattern to an input on a not connected *Remote Inputs* board.

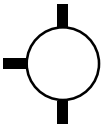
If we write a program that takes a variable containing the remote inputs' status and applies it to an Output Relays instruction, then we have direct manual control over the outputs. We turn a switch on - the relay goes on, we turn the switch off - the relay goes off!

Instructions for System Functions

Assigning a Value


Selector Button

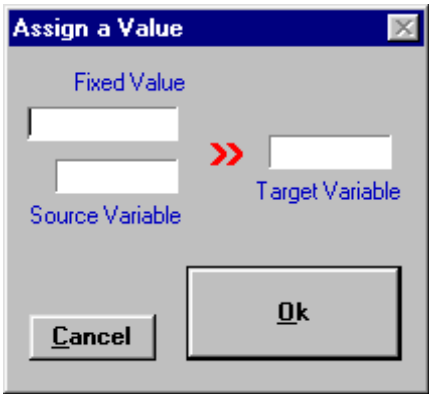

Node Image


Node Type

We've been talking quite a bit about variables and their contents in the previous parts of the manual. Now, how exactly do we get a certain value into a variable? Nothing easier than that, we use the *Assign a Value* instruction.

In the field named *Target Variable* we have to type the name of the variable that we want to assign the value to. The value itself we can get in one of two ways. We can enter it in the input field that is labeled *Fixed Value* or we can tell the SMART1 to take the content of another variable at run-time and assign it to our target. The latter case requires us to specify the name of the variable we want to take the value from in the *Source Variable* input field. The *Source Variable* will not lose its content. It will merely be copied to the *Target Variable*.

There are some important things that we have to pay attention to when we are using the *Assign a Value* instruction.



If we copy from a *Source Variable*, then we have to make sure that both variables are of the same type. This means that we can only assign a character to a character, a string to a string or a number to a number. For cases in which we would like to copy the content of a variable to another one of a different type, we must use the *Converting Variable Types* instruction.

If we enter the value manually, then it must represent valid data for the type of our *Target Variable*.

A *Character Variable*, for instance, can only accept a single character. We are, however, allowed to enter it using any of the different methods we have read about in the section that defines *Character Variables*.

The same applies to *String Variables*. When we enter a *Fixed Value* for a *String Variable* we can use displayable characters mixed with decimal expressions, hexadecimal values, binary values, control codes or control code names in any possible way. To make an empty string, for example, any one of the following *Fixed Values* will work: '' #0 $0 %0 ^@ <NUL>. While we are speaking about this particular character, let's memorize the one very important fact about it in connection with strings. Wherever the *Null Character* (this is its name) appears, that's where the string ends, no matter which or how many characters follow after it. Still, nothing will keep us from putting it into a string. The only restriction that applies for assigning a *Fixed Value* to a string is that the *Target Variable* must be long enough to accept the *Fixed Value*. If we have specified to copy the value from a *Source Variable* instead, then the *Target Variable* must have been declared to be at least as long as the *Source Variable*.

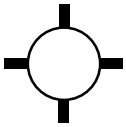
For a variable of the *Number Type*, on the other hand, we don't have to consider its length or the number of decimal places it was declared to hold. These settings are relevant only for those occasions when a number is translated into a string, formatted for display purposes or sent to a serial communications port. As far as the content of the *Fixed Value* is concerned, a *Number Variable* can recognize number digits, a decimal point and a minus sign only.

A node that is programmed with an *Assign a Value* instruction does not have a fail exit.

Comparing Values


Selector Button

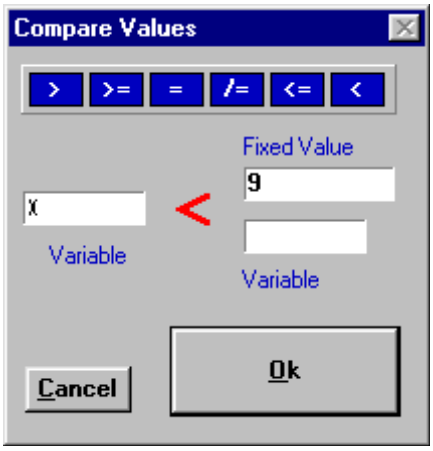

Node Image


Node Type

This particular instruction is used to analyze the content of a variable. It does it by comparing one variable either to a another variable or to a *Fixed Value*. We have six different comparison functions available for this purpose. They are:

>	*greater than*	>=	*greater than or equal to*
=	*equal to*	/=	*not equal to*
<=	*less than or equal to*	<	*less than*

We can select a comparison function by clicking on it. It's symbol gets displayed in the center of the programming window. The variable names or the *Fixed Value* are entered in the appropriate input fields. For this instruction we have to follow the same conventions that also apply to the *Assign a Value* instruction. **If two variables are compared, then they must be of the same type. If a *Fixed Value* is used, then it has to be valid for the type of the variable.**



The data we enter in the programming window builds an expression that is going to be evaluated when the instruction is executed. If the expression is TRUE, then the node's pass exit is used, if it's FALSE, then the program will follow the path out of the fail exit.

The expression we made in our example is X < 9. Assuming that X is a *Number Variable*, the evaluation of this instruction will return a TRUE for all instances where the content of X is a number smaller than 9. Only if X is equal to 9 or greater, then the expression will be FALSE. Now, if we had used > instead of <, then we'd be looking at a whole different picture. The instruction would return TRUE for X values greater than 9 but FALSE for 9 or less.

Comparing numbers and deciding which of them is greater or smaller is pretty familiar territory for all of us, because we know the order in which numbers are arranged. If, however, we want to compare characters or strings, then we first have to find out how the elements of these *Variable Types* are organized.

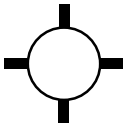
Characters are lined up in the order of their binary values, which also happens to be the order of their decimal and hexadecimal values. The smallest one is the Null character. Its value is %0 or #0 or $0. The biggest character has no particular name but a value of %11111111 or #255 or $FF. All other characters are somewhere between those two, in exactly the same order as they are listed in the **Table of Characters** on page 29. The letter *A*, for example, is smaller than the letter *B* and a small letter *z* is greater than a capital letter *Z*.

The order of strings is determined primarily by their length. Regardless of their contents, the SMART1 considers the longer string also the bigger one. Only if two strings have the same length, and we're talking actual length here, not declared maximum, then a character by character comparison is performed. The string that wins the battle between the first unequal pair of characters is going to be the greater one. Following these rules, "BIG" is smaller than "little".

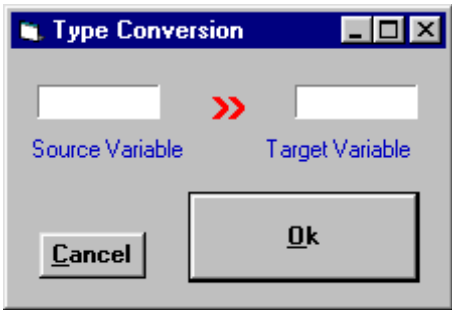
Converting Variable Types


Selector Button


Node Image


Node Type

This instruction lets us convert the content from any one of the SMART1's three *Variable Types* to a value for any other type. Similar to the *Assign a Value* instruction, this instruction also utilizes a *Source Variable* and a *Target Variable*. The name of the variable we are copying the value from must be entered in the input field labeled *Source Variable*. In the *Target Variable* field we type the name of the variable that is supposed to receive the converted value. Unlike the *Assign a Value* instruction, where both variables have to be of the same type, the *Converting Variable Types* instruction will only work with variables of different types. A node programmed with this kind of instruction will always have a fail exit. Depending on the *Variable Types* involved in the conversion, the instruction follows different rules.



If we want to convert the content of a **Number into a String** value, then we must specify a *String Variable* with a maximum length equal to the declared length of the *Number Variable* or greater than that. If, at run-time, the number is within the allowed range for its declared length and decimal places, then the conversion will be successful. The content of the *String Variable* is going to be formatted following the rules for displaying of *Number Variables*. The program will continue through the node's pass exit. In a case where the conversion fails, the fail exit is going to be used and the *String Variable* will remain unchanged.

Converting the content of a **String into a Number** puts no restrictions on the declared lengths of the involved variables. For a successful conversion the string can either be empty, in which case the result will be 0, or it must contain blank spaces and numeric characters only. It may also have one decimal point and one minus sign which can be preceded only by spaces. Ignoring all non numeric characters in the string, the value of the lined up digits must not be greater than **16777215**. The instruction will pass if all those conditions are met. Otherwise the conversion will fail and the number is not going to change.

The procedure of translating a **Number into a Character** only works if the number is positive and smaller than 10. If this is the case, then the *Character Variable* will receive the integer portion of the number. Again, a failed conversion won't alter the *Target Variable*.

If we are trying to convert any non numerical **Character into a Number**, then all we get is a failed instruction. The conversion can be done only to characters 0 1 2 3 4 5 6 7 8 and 9.

We can convert almost every single **Character into a String**. The only exception is the Null character (<NUL>). It will cause the instruction to fail.

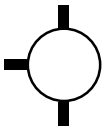
The final variation of the *Converting Variable Types* instruction is the one that converts a **String into a Character**. It takes the very first character of the *String Variable* and puts it into the *Character Variable*. Only a completely empty string can make it fail.

conversion examples	**N**=*number (len 6, dec 2)*	**S**=*string (len 80)*	**C**=*character*	*(_ =blank space)*					
variable	**N**	**S**	**N**	**C**	**N**	**S**	**C**	**S**	**N**
content	2.876	__2.88	2.88	2	2	__2.00	_	_	0

Mathematics


Selector Button

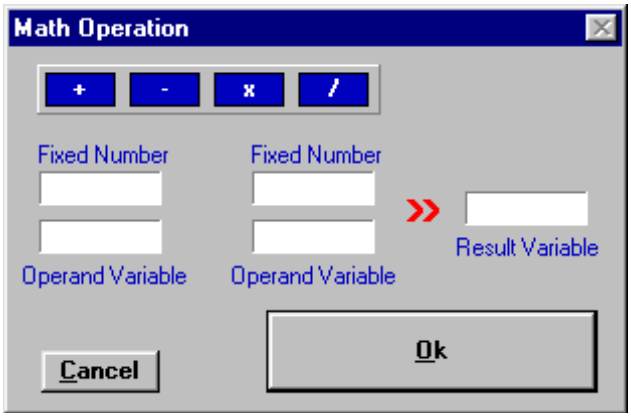

Node Image


Node Type

With this instruction we can turn the SMART1 into a simple calculator. It goes without saying that we can only use variables of the *Number Type* with it. The *Math Operation* instruction gives us the possibility to apply either one of the four basic arithmetic operations.

+	*Addition*	**x**	*Multiplication*
-	*Subtraction*	**/**	*Division*

To define a particular operation we must click the sign in the upper part of the programming window that corresponds to the operation. The symbol will then be displayed in the area between the two operands. The instruction lets us do calculations involving either two *Operand Variables* or one *Operand Variable* and one *Fixed Value*. We can't do a calculation with two Fixed Values. If we want to do that, then we'll have to first use our "gray matter" and then the *Assign a Value* instruction.



The result of the calculation is always saved in a variable. We have to type its name in the input field labeled *Result Variable*. There is no requirement that says we have to specify different variables. The *Result Variable* can be the same as either one or both *Operand Variables.*

A node of a *Math Operation* instruction has no fail exit. The program will always continue with the instruction tied to the pass exit.

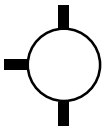
The settings specifying the *Maximum Length* and number of *Decimal Places* for the concerned variables are completely irrelevant for any calculations. After execution of the instruction the *Result Variable* will always contain a number, no matter what values the operands had or which operation was executed. However, being able to display that number is an entirely different story. This can only be done if the number's value lies within the range allowed by its settings. For a number that is too big or too small for its declared settings, a string of stars as long as the variable's *Maximum Length* will be shown. The same goes for transmitting the number out of a serial port.

At first look we might think that the four basic operations are not nearly enough to cover all the high-tech applications we are planning to attack with the SMART1. But, let's think again. Since everything else in arithmetic is supposed to be based on those four operations, why shouldn't we be able to make the thing do a more complicated calculation. It's only a matter of applying a suitable algorithm. The library module named "Sq_root.Mod", for example, shows one possible approach to calculating a number's square root.

Extracting Sub-Strings

Selector Button

Node Image

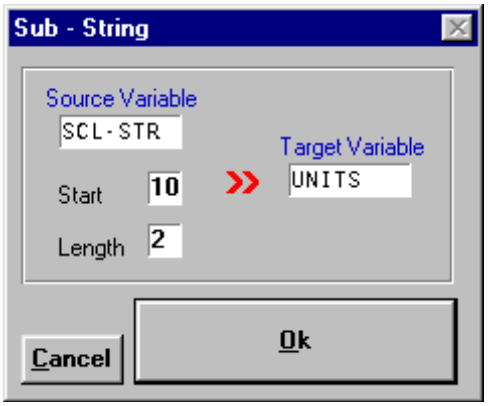


Node Type

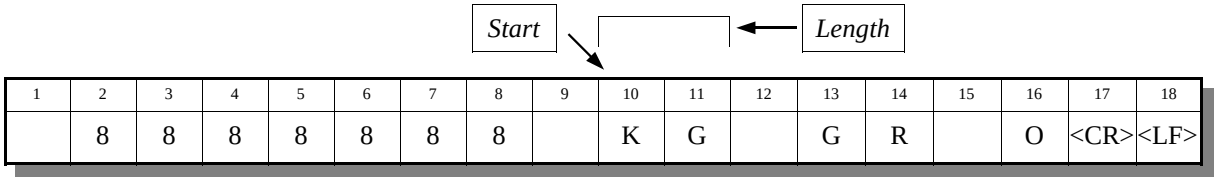
This is the first of two instructions which enable the SMART1 to manipulate the contents of *String Variables*. This particular instruction's purpose is to isolate a part of a string.

It does that by finding the *Sub-String* in the *Source Variable*, and storing it in the *Target Variable*. Both these variables must be of the *String Type*. The parameters that we have to specify for *Start* and *Length* determine where in the *Source Variable* the *Sub-String* begins, and how long it's supposed to be.

To demonstrate exactly how the instruction works, let's have a look at an example which was taken from a module that reads the serial output string from a DF1500 scale indicator, and then splits it into its individual components.



We are assuming that the string we'll get from the indicator is going to have an overall length of 18 characters, and that the two characters in the 10th and the 11th position will represent the units of measure. We are further assuming that when the *Sub-String* instruction is executed, a variable named SCL-STR will contain that string. It will be our *Source Variable*. The variable called UNITS is to be used for storing the information about the units of measure. We have to type its name in the input field labeled *Target Variable*. Knowing that the data we're interested in begins with character number 10, we enter a 10 as parameter for *Start*. And, because the units of measure are symbolized by two characters, we specify 2 as the *Length* of the *Sub-String*. Naturally, the *Target String* in our example can't have been declared with a *Maximum Length* less than 2.



Presuming that everything went okay, the UNITS variable in our example will contain 'KG' when the instruction has finished its task. But, there's also the possibility that something goes wrong. Here is what would happen:

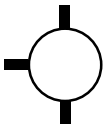
If during execution of a program a situation arises in which the *Source Variable* does not contain as many characters as are necessary to fill the projected *Length*, then the *Sub-String* will be truncated. In case the string in the *Source Variable* is not even as long as specified in *Start*, the *Target Variable* will be empty. However, since a node programmed with the *Sub-String* instruction does not posses a fail exit, the program will regardless of the result in the *Target Variable* always continue at the pass exit.

There is one more rule for this instruction: We are allowed to use the same variable for both source and target.

Concatenating Strings

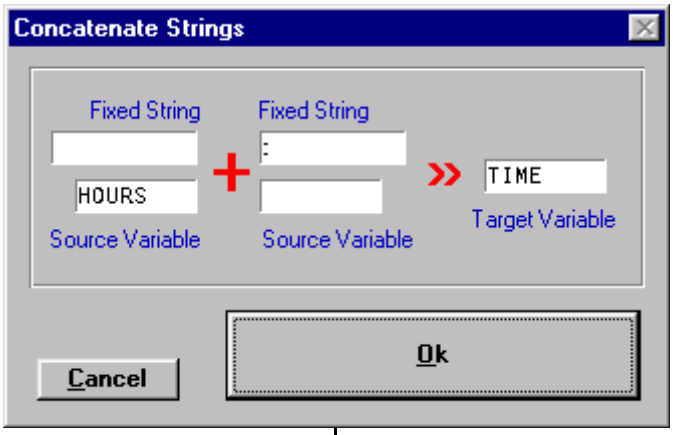
Selector Button

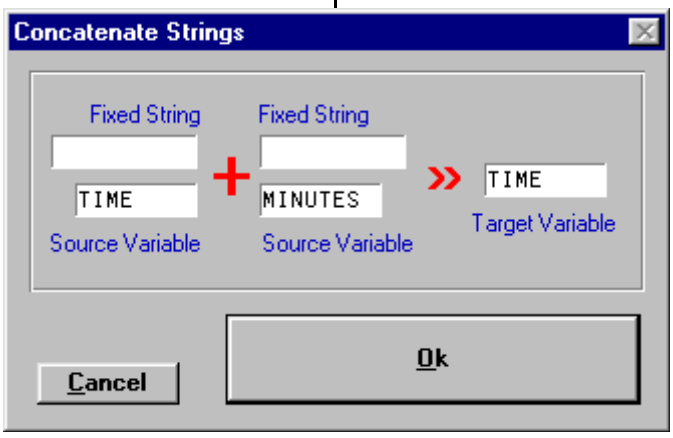
Node Image


Node Type

The second one of the SMART1's string manipulation instructions is called *Concatenate Strings*.

According to the dictionary, "concatenate" means to join or link together. And that's basically what the instruction does, it links two strings together. It also saves the newly created string in a variable, so that later on we will be able to do something with it. For obvious reasons, this variable must be of the *String Type*. We have to enter its name in the field labeled *Target Variable*. The strings to be concatenated may be either the contents of two *String Variables* or one variable's content and one *Fixed String*. The *Concatenate Strings* instruction doesn't permit us to join two *Fixed Strings*. If we absolutely need to do that, then we must resort to the same procedure which is applied for calculations with two *Fixed Values*. It was described in the section about the *Math Operation* instruction, a few pages before this one.





To concatenate more than two strings, we have to execute the instruction several times, always using the same *Target Variable*. From the second time on, the variable used as target must also be one of the sources. If, for example, we wish to build up a string to show the time in the format 'hh:mm', then we would probably apply a method similar to the one shown in the pictures on the left:

The first step will link the string in the variable HOURS with the colon in the *Fixed String*, and then save the result in the variable TIME.

The second instruction is going to take the content of the TIME variable, and append the content of the variable named MINUTES to the end of it. Again, the result will be saved in the variable TIME.

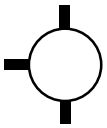
This example revealed that like the *Sub-String* instruction, the *Concatenate Strings* instruction does not require us to specify different variables for source(s) and target.

The instruction does not generate an error if the result of the concatenation is longer than the *Maximum Length* of the *Target Variable*. If this should be the case, then all characters that don't fit into the *Target Variable* will simply be lost. Consequently, a node programmed with this instruction has no fail exit.

Bit Manipulation

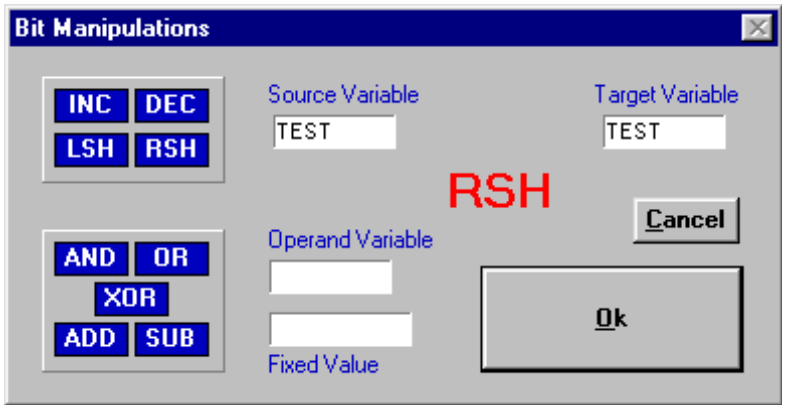
Selector Button

Node Image



Node Type

We can program the *Bit Manipulation* instruction to perform one of nine different operations. These are divided into two groups. The first group, represented by the four commands in the upper part of the programming window, requires a *Source Variable* and a *Target Variable* only. For the operations of the block in the window's lower half, we must also specify the name of an *Operand Variable* or enter a *Fixed Value* instead. All variables used in this instruction must be of the *Character Type*. It is possible to use the same variable for the source, for the operand and also for the target. We can select the operation to be executed simply by clicking on the command which corresponds to it. The command will then appear in the central area between *Source Variable* and *Target Variable*.



In particular, the commands of the first group and their associated functions are:

INC - increment

The *Source Variable's* value is read, incremented by 1 and stored in the *Target Variable*.

Source	Target	
%01000001	%01000010	binary
$041	$042	hexadecimal
#065	#066	decimal
'A'	'B'	printable

DEC - decrement

The *Source Variable's* value is read, decremented by 1 and stored in the *Target Variable.*

Source	Target	
%01000001	%01000000	binary
$041	$040	hexadecimal
#065	#064	decimal
'A'	'@'	printable

LSH - Left Shifting

The *Source Variable's* value is read, the leftmost bit is dropped and all other bits are moved one position to the left. A 0 bit is put in the rightmost position.

Source	Target	
%01000001	%10000010	binary
$041	$082	hexadecimal
#065	#130	decimal
'A'		printable

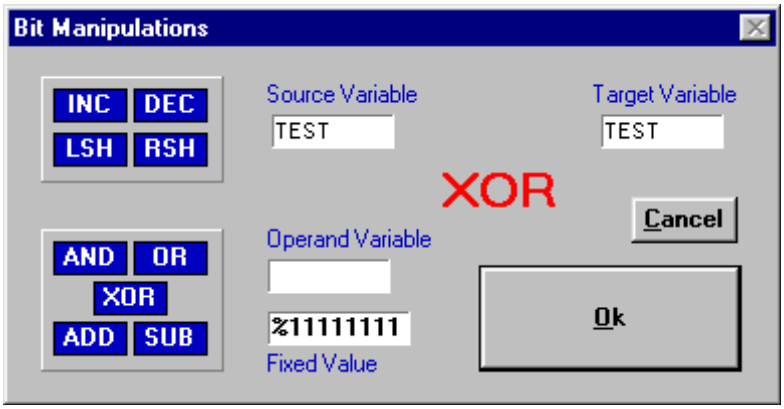
RSH - Right Shifting

The *Source Variable's* value is read, the rightmost bit is dropped and all other bits are moved one position to the right. A 0 bit is put in the leftmost position.

Source	Target	
%01000001	%00100000	binary
$041	$020	hexadecimal
#065	#032	decimal
'A'	' ' (space)	printable

All commands in the second group of the *Bit Manipulation* instruction require an *Operand Variable* of the *Character Type* or a *Fixed Value* instead. To type the latter, we may choose any of the methods describing a character value.

The particular commands and their functions are:



AND - logical and

Source	Operand	Target	
%01000001	%01000010	%01000000	binary
$41	$42	$40	hexadecimal
#065	#066	#064	decimal
'A'	'B'	'@'	printable

The *Source Variable's* value is read, each of its bits is paired with the corresponding bit of the operand. If both bits in the pair have a value of 1, the corresponding bit of the *Target Variable* will also be 1. All pairs with one or two 0 bits will result in a 0 bit in the associated location of the *Target Variable*.

OR - logical or

Source	Operand	Target	
%01000001	%01000010	%01000011	binary
$41	$42	$43	hexadecimal
#065	#066	#067	decimal
'A'	'B'	'C'	printable

The *Source Variable's* value is read, each of its bits is paired with the corresponding bit of the operand. If one or both bits in the pair have a value of 1, the corresponding bit of the *Target Variable* will also be 1. All pairs with two 0 bits will result in a 0 bit in the associated location of the *Target Variable*.

XOR - logical exclusive or

Source	Operand	Target	
%01000001	%01000010	%00000011	binary
$41	$42	$03	hexadecimal
#065	#066	#003	decimal
'A'	'B'		printable

The *Source Variable's* value is read, each of its bits is paired with the corresponding bit of the operand. If one bit in the pair has a value of 1, the corresponding bit of the *Target Variable* will also be 1. All pairs with two 0 bits or two 1 bits will result in a 0 bit in the associated location of the *Target Variable*.

ADD - add an operand

Source	Operand	Target	
%01000001	%01000010	%10000011	binary
$41	$42	$83	hexadecimal
#065	#066	#131	decimal
'A'	'B'		printable

The *Source Variable's* value is read, the operand is added to it. The result is saved in the *Target Variable*.

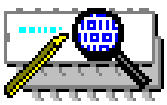
SUB - subtract an operand

Source	Operand	Target	
%01000001	%01000010	%11111111	binary
$41	$42	$FF	hexadecimal
#065	#066	#255	decimal
'A'	'B'		printable

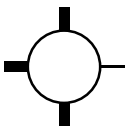
The *Source Variable's* value is read, the operand is subtracted from it. The result is saved in the *Target Variable*.

Access to Files

Selector Button



Node Image



Node Type

This instruction type is probably the most complicated one of all. It includes the seven different instructions necessary for handling data organized in a file structure.

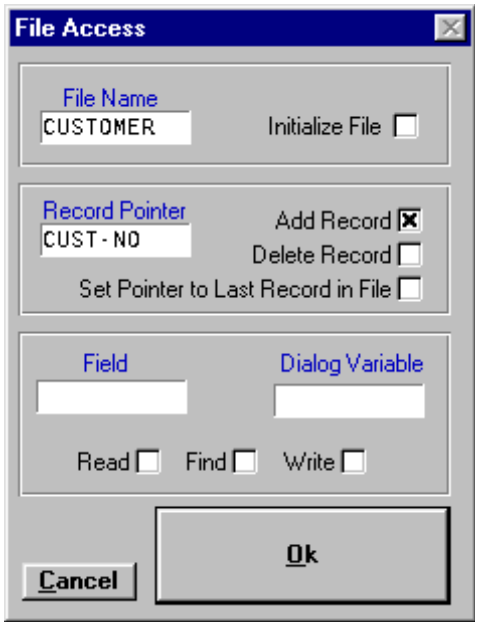
Initializing a File

The basic file management instruction is the *Initialize File* instruction. A node that is programmed with this instruction does not have a fail exit. All we must do to specify it, is to click on the check box labeled *Initialize File* and to type the name of the file in the *File Name* input field. Naturally, for any instruction of the *File Access* type to work, the stated file must have been defined in one of our project's program modules.

Before we can make use of any file, it is absolutely necessary to first apply this instruction to it. It will do two things. One is to initialize the file by setting the maximum number of records in the file prefix. Its other function is to reset the internal pointer that keeps the number of used records. Since this instruction essentially wipes the file clean, it would be unwise to put it in a startup sequence that is getting executed every time the SMART1 is powered up. The way we would probably do it, is to place the instruction in a module that must be explicitly called by the operator.

Adding a Record

Prior to writing new data to a file we must open up some space for it. We do this with the *Add Record* instruction by checking the *Add Record* box. The instruction frees up just enough memory bytes to take exactly one new record. Like with all other instructions of the *File Access* type, we also have to enter the file's name. An additional requirement for this instruction is to enter the name of a so called *Record Pointer*. This is a *Number Variable* that we must have declared somewhere in the project. The value of this variable tells the SMART1 where in the file to insert the space for the new record. For a previously untouched file we have to assign the value 1 to this pointer before we call the *Add Record* instruction. If the file already has existing records, then the new record is added at the location which the *Record Pointer* "points" to and all data above it is moved one record upwards. To add a record to the end of a file we always have to load the *Record Pointer* with 1 more than the number of existing records.



A *Record Pointer* value which exceeds the number of actual records by more than 1 will cause the instruction to fail. The node's fail exit will also be used, if the file had already reached its maximum number of records.

Deleting a Record

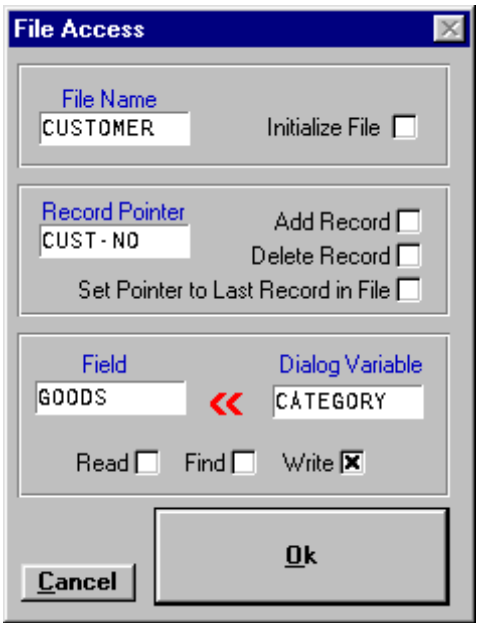
An instruction that is handled like the *Add Record* instruction but has the exact opposite purpose, is the *Delete Record* instruction. For this one too, we need to specify the *File Name* and a *Record Pointer* variable. But this time, we have to click the *Delete Record* check box.

Given the fact that we can only delete an existing record from a file, the value of the *Record Pointer* must be somewhere in between 1 and the number of the file's last current record. If this is the case, then the record which is pointed to will be erased and all the ones above it will slip down to fill the gap. If, for whatever reason, the *Record Pointer* should have a value less than 1 or bigger than the number of the last record, then the instruction will fail.

Obtaining the Current Size of a File

The third instruction of this group produces a node without a fail exit. We choose it by clicking on the checkbox with the title *"Set Pointer to Last Record in File"*. The instruction's purpose is to first look how many records there are in the file and then to store that number in the *Record Pointer* variable. For an empty file it will return 0.

Now it's getting interesting. For the final three instructions of the *File Access* type we have to employ another additional variable. It's called the *Dialog Variable* and its purpose is to "talk" to a field inside our file. All data transfers in and out of a file are handled through *Dialog Variables*, no field can be manipulated directly. The Type of the *Dialog Variable* must always be the same as the type of the field it is dealing with.



Writing to a Field

The *Write* instruction is defined by a check mark in the box labeled *Write* and indicated by a symbol showing the data flow going from the *Dialog Variable* to the *Field*.

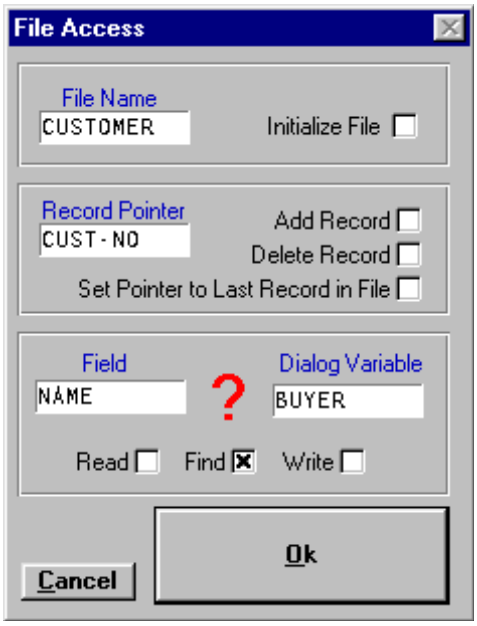
The instruction will copy the variable's content into the field to which the *Record Pointer* points. If the data to be transferred happens to be a string then the maximum length of the *Dialog Variable* must not exceed the *Field*'s maximum length. The settings for maximum length and decimal places for variables of the *Number Type* are completely irrelevant in a file context.

Again, an out of bounds *Record Pointer* value will result in a failed instruction. The *Field*'s contents will remain unchanged in such a case.

Reading from a Field

The *Read* instruction does the opposite operation of the *Write* instruction, it copies data from a field to the *Dialog Variable*. We have to check the box labeled *Read* to specify it. The data flow symbol will point from the *Field* towards the *Dialog Variable*, which, if the data to be read from the file is of the *String Type*, must have been declared with a maximum length of at least the *Field*'s maximum length. As with the *Write* instruction, the maximum length and decimal places of *Number Variables* are of no consequence when we read data from a *Number Field*.

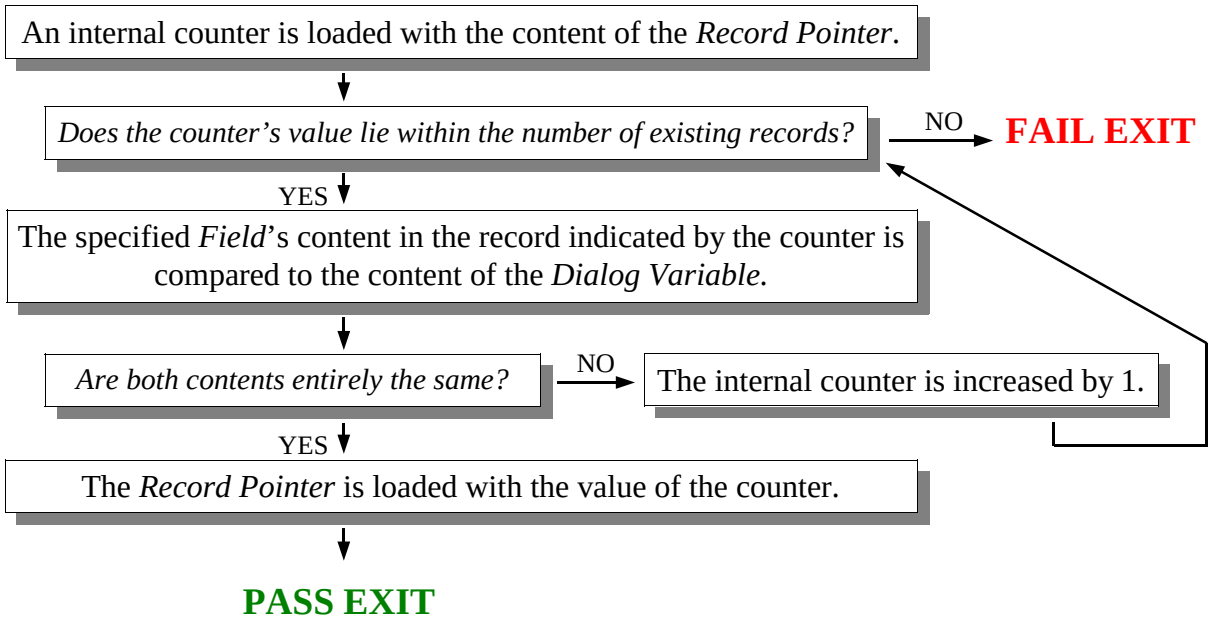
Finding Data in a File



The final instruction of the *File Access* type is the *Find* instruction. We can use it when we want to look for the occurrence of particular data in a file.

In order to invoke this instruction we must - most of us guessed it - click on the check box that is titled *Find*. Other than that, we have to supply all the same information as we do for the *Read* and *Write* instructions. It needs to know the *File Name* and it requires a *Record Pointer* as well as a *Dialog Variable*. A question mark between the *Field* and the *Dialog Variable* is the symbol for this instruction.

No *Field* is being written or read, the only thing the instruction does, is to search the file for the data we told it to find. It works somewhat like this:



Program Comments

Selector Button

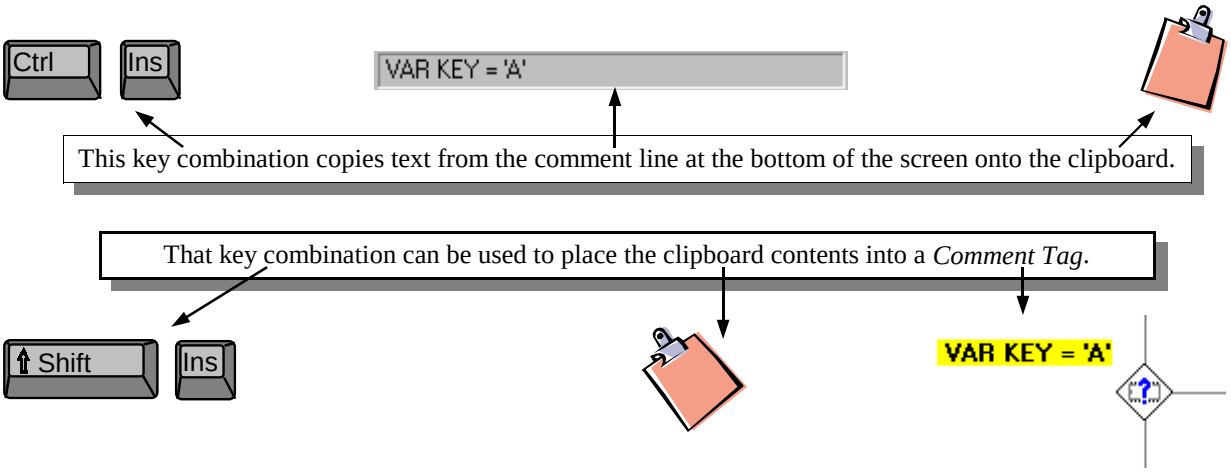
Node Image

Node Type

Since the images of nodes in a module only reveal the type of the particular instruction, but say nothing about their internal program, it's probably a good idea for us to develop the habit of putting comments into our program modules. For exactly this purpose we can use the *Comment Tag* instruction. Well, technically speaking it's not really an instruction. It neither gets executed nor does it have any other effect on the project. The SMART1 ignores *Comment Tags* completely. They are, however, a very valuable tool for somebody who wants to change a project. They can tell him what we had in mind when we first created the program.

The programming of a *Comment Tag* instruction is not done in a programming window. The same method that is used for *Jump* instructions and *Call a Module* instructions is applied here too. When we click the right mouse button, a text input field appears on top of the instruction. In it we can type our comments. A unique property of the *Comment Tag* is, that its size is adjusted automatically to fit its contents. To add a new line to a comment we have to press and hold the **Ctrl** key and press the **Enter** key. Pressing the **Enter** key by itself will close the input field and update the *Comment Tag* with the new text. If we changed our minds while entering a comment, we can press the **Escape** key and thereby discard the changes we just made to the comment.

A tool that might come in handy in conjunction with the Comment Tag is the so called clipboard. This is a temporary storage location for information we cut or copy. The clipboard is always present in Windows, hence, it's present in the SMART1 Programmer. Whenever we use a cut or copy function, information is placed on the clipboard. When we apply a paste function, on the other hand, the information from the clipboard is placed in the currently active (text)object. Information on the clipboard remains there until we cut or copy another piece of information onto it, or until we quit Windows. Most of us have most likely already discovered that the comment line at the bottom of the screen displays some sort of instruction summary whenever we move the mouse cursor over a programmed instruction. Here's where the clipboard comes into the picture. Every time we press the key combination **Ctrl Insert** in such a moment, the text shown in the comment line is taken over onto the clipboard. If we now click the right mouse button on a *Comment Tag* and press the key combination **Shift Insert**, the text from the comment line is going to appear in our *Comment Tag*.



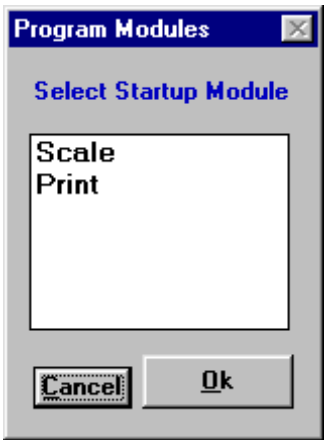
CHAPTER 3 - TO ROUND IT ALL UP

The Software Shell, second part

Compiling a Project

We have read all there is to read about the particular instructions the SMART1 Programmer has to offer and, hopefully, even understood some of it. We have placed and moved instructions, we have programmed them and we have tied them together. Now, we want to make nails with heads and compile our first little project. There are two ways to start the compiling process. We can either select the ***Compile*** function from the ***Process*** menu or click the ***Compile Button***  on the shortcut button bar, if it is visible.

For practical purposes, let's assume that our project contains two program modules.



That being the case, the window shown on the left will pop up. It lists all program modules in the project and asks us to select a startup module. This will be the first one to be executed, every time the SMART1 is powered up. To specify which of the project's program modules is going to be the startup module, we move the mouse cursor over its name and click the left button. The module's name will now be highlighted. Pressing the Cancel button leaves the window without a specified startup module. Therefore no compiling will be done. If we push the Ok button, however, the window closes and the compiling process is on its way.

There are many steps on the way to a successfully compiled project. For example, the SMART1 Programmer must check all nodes to find out if they are programmed and properly tied together. It must also check if all variables were declared and are used in accordance with their types. It has to calculate the amount of memory required for all variables and files, and determine if everything is going to fit into the SMART1's memory chip. Then it must translate every single instruction into machine code and, finally, link all the code together in the proper sequence. All these things will take their time. During the compilation the mouse cursor is going to take the shape of an hourglass.

If everything went well, another window is going to appear. It is conveying a cheerful message to us that says: "We did it!"

After we click the Ok button, the window will vanish and we can go ahead to send the compiled code to the SMART1. How that is accomplished is described on the page titled *Sending Code to the SMART1*.

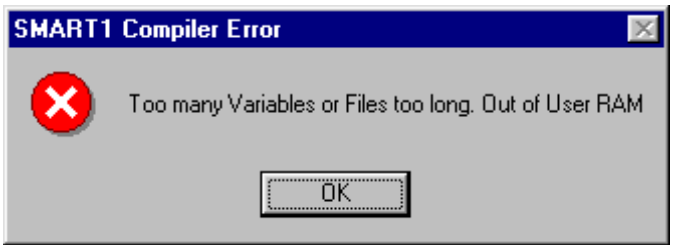
So much for the theoretical case. The compile process' outcome which is much more likely, at least during our our first few humble attempts, won't include the above window.

That is mainly due to the fact that in each of the before mentioned tasks, which the SMART1 Programmer has to perform, lies the danger, that we either made a mistake or asked something of the SMART1 that it can't quite accommodate. If this should happen, our project can't be compiled. Instead, the SMART1 Programmer's built in error checking functions are going to kick in, and we will be confronted with an assortment of less cheerful ***Error Messages***.

The following pages contain a list of all possible error messages and explanations for them.

Compile Error Messages

While compiling, the SMART1 Programmer will try to "catch" all the things which, if downloaded, would make no sense to the SMART1. We call these things errors. There is quite a variety of them. Some can occur in conjunction with only one specific instruction, others could be caused by a number of different instructions. And, there are errors that can't be blamed on any particular instruction, but on the sum of the circumstances in a project. Each of the individual errors comes with it's own error message. Here are the errors which are not directly caused by an instruction:



Out of RAM

This error is going to occur as the result of file declarations that exceed the amount of available space in the SMART1's **R**andom **A**ccess **M**emory chip, used for data storage.

The solution to this problem is to reduce either the number of records or the size of the individual fields. Since files and variables share the same chip, we might also be able to fix the situation by cutting down the number of variables or, in case of string variables, reducing their length.

Not executable

If the designated startup module contains no executable instructions, the error message shown on the right will be displayed. It will happen when the *Module Start* instruction is missing.

To correct the error we can either select a different program module as startup module, or we can place executable instructions in the module.

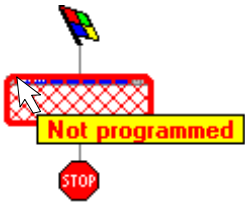
Out of Program Space

This is probably the most severe error we can encounter. When this message is displayed, the code in our project has become too big for the SMART1. We can try to solve the problem by reducing the sizes of constants used in the project. These are all the fixed values in the individual instructions. The largest ones of them are probably the fixed texts in display instructions. If this approach doesn't do the trick, then we will have to "weed out" all the less important program sequences in our modules or try to come up with a leaner programming style.

If any of the above errors was encountered, the compile process will be aborted and the ***Error(s) found*** message will be shown. It will also be displayed if an an error originated in an instruction.

A list of all errors that can be caused by instructions is printed on the next page.

More Compile Error Messages



If an error was caused by an instruction, then that instruction will be covered by a red mask. When we move the mouse cursor over such a masked instruction, a message will pop up telling us what the particular error is. Here's a summary of all possible errors.

Conversion to same type ... Convert type ..... The variables used must be of different types.

Duplicate file definition ... File definition ..... A file with the same name already exists.

Duplicate variable declaration ... Variable declaration ..... A variable by that name already exists.

Entry tie missing ... Any node with entries ..... None of the node's entries are being used.

Exit tie missing ... Any node with exits ..... At least one of the node's exits is not used.

Invalid fixed value ... Assign/Compare ... The fixed value is not valid for the type of the variable.

Invalid jump address ... Jump ..... There is no re-entry point with this number in the module.

Invalid record pointer ... File access ..... The specified record pointer is not a number variable.

Invalid variable type ... Key/Math/Sub-String/Concatenate . Impossible with this type of variable.

Insufficient space in target ... Assign/Concatenate/Convert type ... The target variable is too short.

Not programmed ... Any programmable node ..... The node has yet to be programmed.

Parameter(s) out of range ... Sub-String ..... Start and/or length are outside the source variable.

Referenced subprogram is not executable ... Call module . No start instruction in specified module.

Subprogram not found ... Call module .... The specified module is not part of the current project.

Type mismatch ... Assign/Compare/File access ..... The variable (field) types must be the same.

Undeclared field ... File access ..... No field with the specified name was declared.

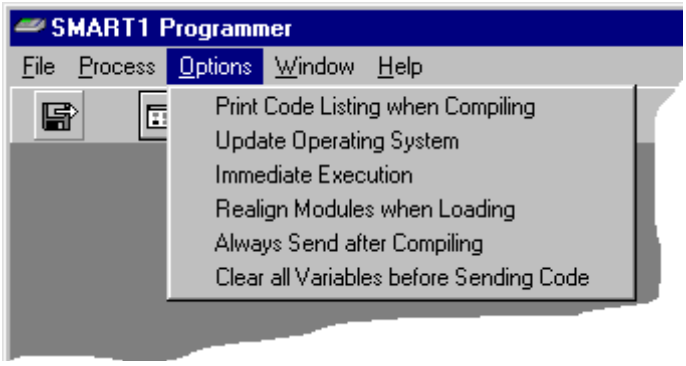
Undeclared file ... File access ..... A file with the specified name does not exist.

Undeclared variable ... Any instruction using variables . The specified variable was not declared.

Unreferenced entry point ... Re-entry point ..... No jumps or ties referring to this re-entry point.

Variable must be String with Length 16 ... Clock ..... The variable must have this specific format.

Program Options



The SMART1 Programmer includes an assortment of optional features which can be accessed via the ***Options*** menu.

At program start none of these options are operative. To activate one we must click on it using the left mouse button. When an option is in effect, it will have a check mark displayed next to it.

Print Code Listing when Compiling

If the compile process is being executed while this option is checked, the SMART1 Programmer will send a list to the printer showing the names, sizes and locations of all variables and files used in the project. It will also print a code listing for every program module. This listing consists of addresses and function summaries for each of the instructions in the particular program module.

Update Operating System

If we have activated this option, the SMART1 Programmer will incorporate the operating system in the compile process and send it to the SMART1 along with the compiled user code.

We must use this option if we are sending a project to a SMART1 that is being programmed for the very first time, or when we are going to re-program a unit in the future and want to use an updated version of the operating system.

This option will be checked automatically whenever we activate the ***Immediate Execution*** option.

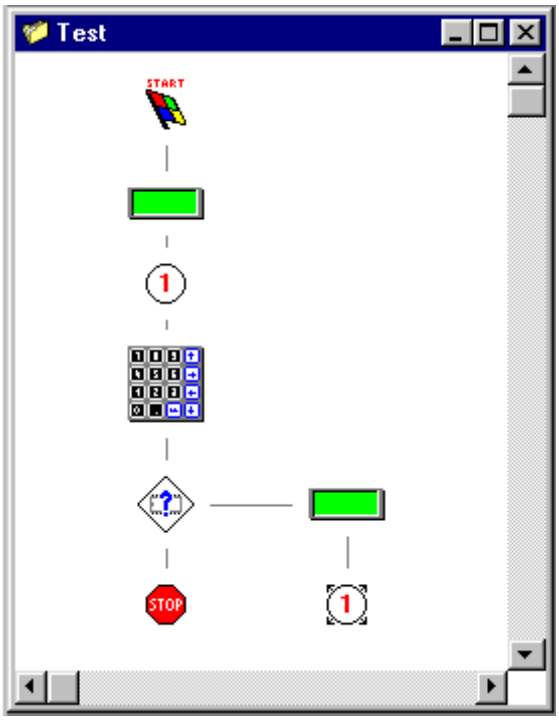
Immediate Execution

Using this option, we can make the SMART1 bypass the startup screen and immediately execute the project's startup module.

If the SMART1 was already programmed utilizing this option, then it is not necessary to use it again. The ***Update Operating System*** option will be checked automatically, whenever this option is in effect.

In a situation where we need to re-program a SMART1 which was previously programmed using this option, but which is not supposed to bypass the startup screen in the new project, we will have to activate the ***Update Operating System*** option by itself.

Realign Modules when Loading



At rare occasions it might be possible that a program module we just loaded looks a little odd. It could happen, as the example to the right is showing, that the ties in the module do not quite reach all the way to the nodes.

This kind of distortion could occur when the particular module had been created on another computer, either under a different version of Windows (Win 3.11/Win 95) or with a different screen ratio or resolution.

If this should be the case, then we can engage the ***Realign Modules when Loading*** option and reload the program module or the entire project. After doing this, our program module should look normal and we may turn the option off again.

Always Send after Compiling

If this option is checked, it will cause the SMART1 Programmer to automatically pop up the ***Send Code*** window every time a project was successfully compiled.

This eliminates both the need to click the Ok button in the compiler message as well as the process of opening the ***Send Code*** window.

Clear all Variables before Sending Code

If this option is checked, then the array of the SMART1's memory which holds variables and files will be flushed. When the SMART1 starts a project that was downloaded with this option in effect, all variables will be cleared. The same is true for files. All of their fields are going to be empty, and the files will not be accessible until the necessary ***Initialize File*** instructions are applied to them.

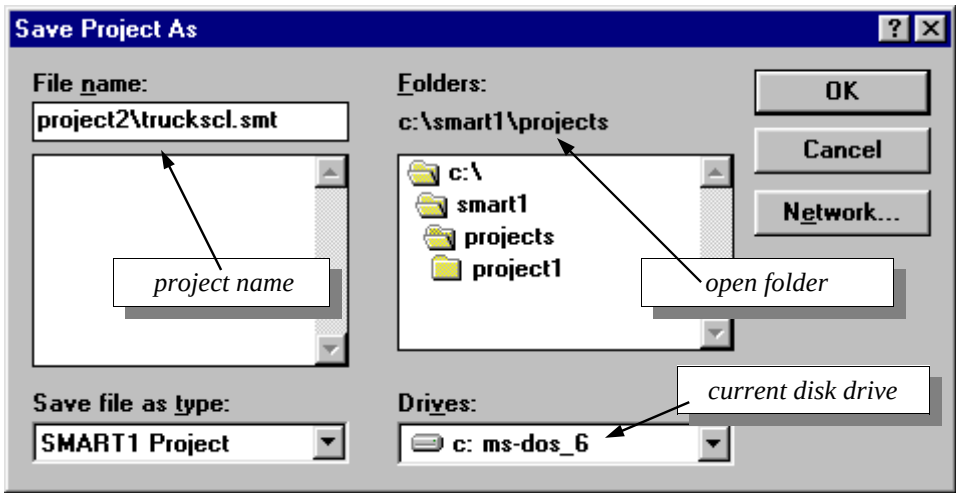
After the clearing process the particular values for the variables of the three individual types will be as follows:

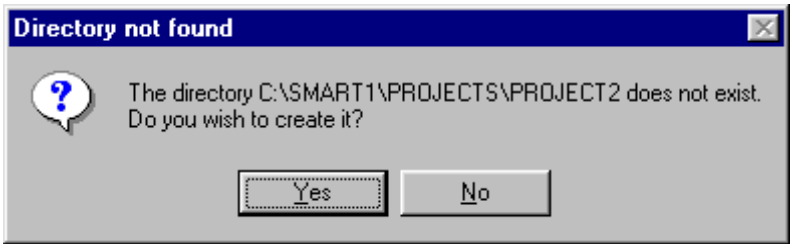
All ***Number Variables*** are going to have a value of 0 (zero). Every single ***Character Variable*** will hold a ***Null Character***, and all ***String Variables*** are going to contain empty strings.

Saving and Loading a Project

We have created our first little project and even successfully compiled it. Now, the time has come to save it onto disk. Again, there are different methods to do this. For once, we can click our way through the ***File*** menu and the ***Project*** menu to the ***Save Project*** function. The same task can also be accomplished by holding down the ***Alt*** key and pressing the keys ***FPS*** in sequence. If the ***Save Project button*** is visible, then we may simply click on it to save the project.

If we are saving this particular project for the first time, or if we have selected the ***Save Project As*** function from the project menu, then a file dialog window is going to open up. It shows us the current disk drive as well as the folder that is currently open. It also allows us to enter a name for our project. The name we specify must conform to the same regulations that apply to names for program modules. The file extension assigned to SMART1 project files is "**.smt**". The project with all its program modules will be saved in the current folder. If we want to save a project in a different folder, then we can either select it from the list of folders, or enter its name along with the name of the project. The latter was done in the example above. In cases where the specified folder does not exist on the current disk drive, another window appears, asking us if we wish to create the folder. Clicking the *Yes* button will create the folder and save the project in it. If we click the *No* button, the project will not be saved.



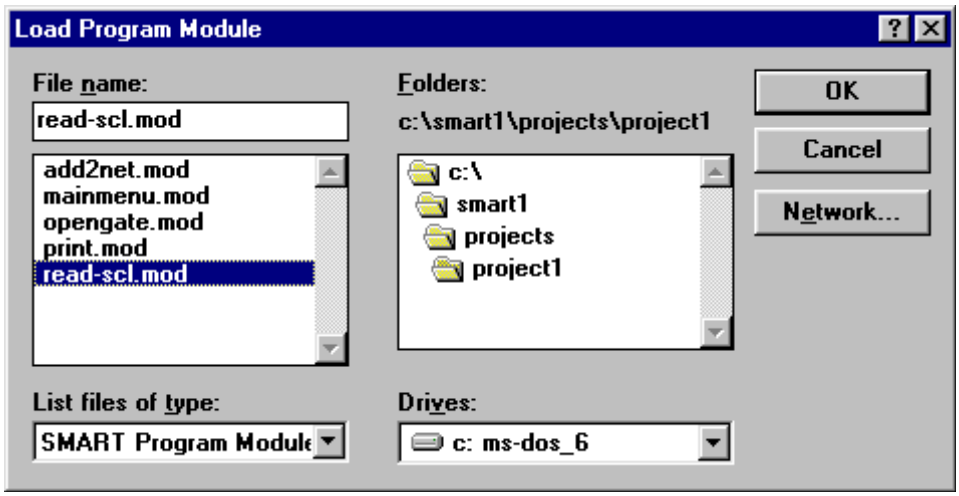


Whenever we use the ***Save Project*** function with a project that had already been saved before, no file dialog window will appear. The project will be saved under the same name and in the same folder that was used before.

The steps for loading a previously saved project are similar to the steps to save it. The quickest way is to click on the ***Load Project button***. We can also go through the menus, either by clicking the mouse or by typing the hot keys sequence ***Alt FPL***. If a project is presently loaded and had been changed, then we will be asked, whether we wish to save it or not. Again, a file dialog window will open up. This time it's titled "Load Project". As soon as we have selected a project from it and click ***Ok***, the project will be loaded into the SMART1 Programmer.

Loading an existing Program Module

Let's assume we are currently working on a project that requires the printout of a weight. We are just about to open a new program module to specify the instructions necessary for reading the scale indicator, when we remember that we made a project once before, that included a scale reading procedure. So, instead of creating a new module we choose the item ***Load program module*** from the program module menu. A file dialog window allows us to locate the module. After finding it on the computer's hard disk we select it and click the ***OK*** button. An icon for the module takes its place beside the other modules on the project screen. The module is now part of our project and we are able to call the scale reading procedure from any place in our project simply by putting a *Call a Module* instruction in that place, and programming it with the name of our scale reading module.



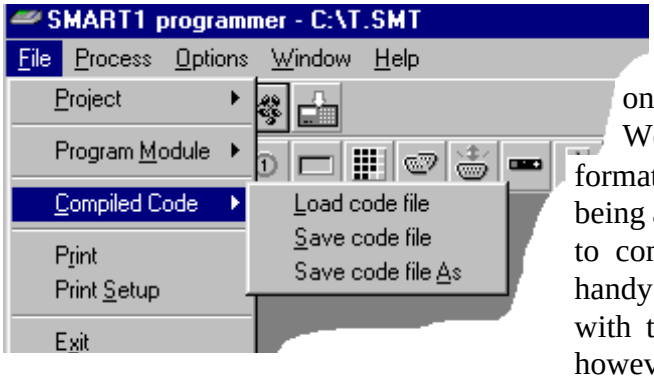
Saving a Program Module

If we need to save a program module by itself, then we have to select it first. This is simply done by clicking anywhere on the module. We can recognize the selected program module by its highlighted title bar. Once this is done we can save it in almost the same way as we save a project. The hot keys sequence ***Alt FMS*** will get us through the menus and save the module in the project's folder (directory). Of course, we can also use the mouse and click our way to the ***Save Module*** function.

In cases where we want to save a module in a different folder, we can do this by using the ***Save Module As*** function. We can even create a new folder to save the module in. For this we can apply a procedure equivalent to the one for saving a project, which was described on the previous page. However, if after saving the program module in another folder either the ***Save Module*** function or the ***Save Project*** function is used, the module will again be saved in the project folder.

The ***Save Module As*** function is also the one we have to use, if we want to rename a program module.

Making and Saving a Code File

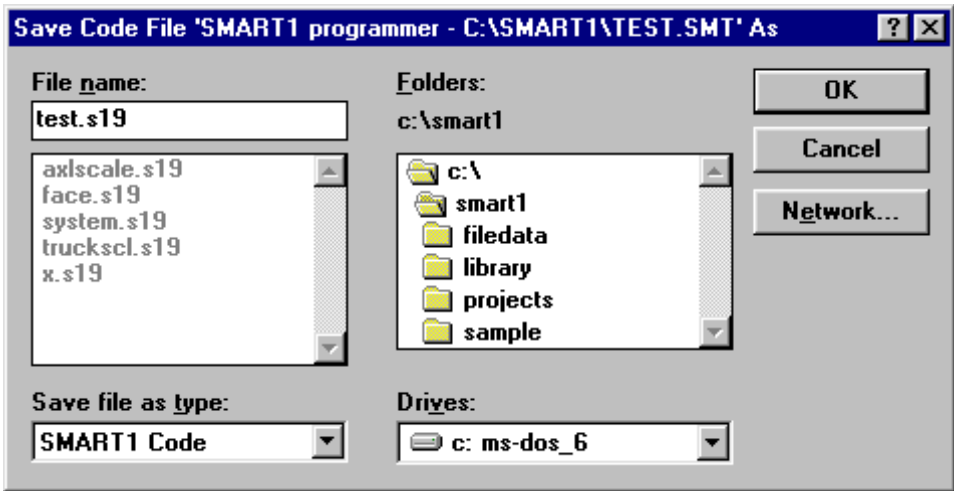


If we want to, we can choose to store the project for a SMART1 in its compiled format on disk.

We refer to files that are saved in this special format as ***Code Files***. They give us the advantage of being able to quickly reload a project without having to compile it again. A feature that will come in handy if we need to program another SMART1 with the exact same project at a later time. It is, however, not possible for us to change a project that was saved in the ***Code File*** format. For that we will still have to load it in the ordinary way.

Browsing through the menus, one or the other among us may have already noticed that the functions named ***Save Code File*** and ***Save Code File As*** are hardly ever available. This is indeed the case. The only time these functions are not grayed out, is when we have just compiled a project with the ***Update Operating System*** option in effect. The reason for this is, that the SMART1 which we are going to program with this project may have never been programmed before, and therefore won't have an operating system.

The steps we have to perform in order to make a code file are as follows. First the project must be loaded in the SMART1 programmer. Then we have to activate the ***Update Operating System*** option. We may select it either by itself or together with the ***Immediate Execution*** option. As a final step we have to compile the project. If everything went smoothly, we can now save the code file. Using the ***Save Code File*** function will always place the file into the project's directory. To save it somewhere else, we can employ the ***Save Code File As*** function. A file dialog window, similar to the ones we already know from saving projects and program modules will open up. We are allowed to select the folder (directory) and disk drive of our choice from the respective lists in the window.



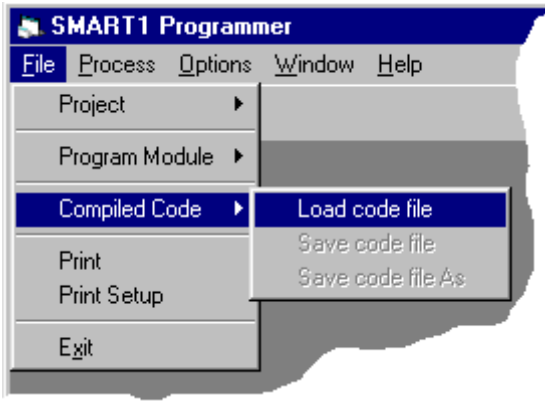
The file extension assigned to SMART1 ***Code Files*** is "**.s19**".

The option enabling us to create a new folder for storing of a program module or a project, is not available with the saving of code files.

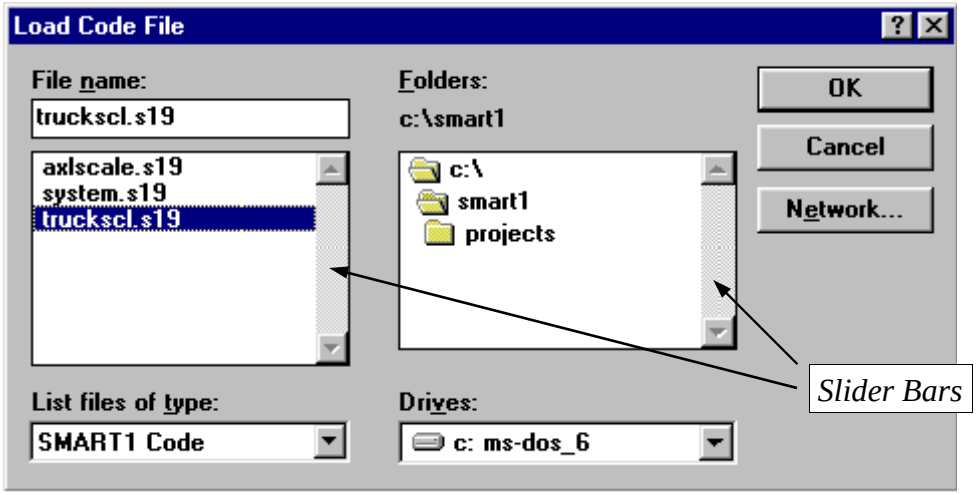
Hey, there's got to be something to be upgraded in the SMART1 Programmer's next version.

Loading a Code File

For all the people who either haven't read the previous page or have already forgotten all about it, let's recapitulate what *Code Files* are. Each *Code File* is an entire project, complete with every single one of its instructions, plus the SMART1's operating system, altogether translated into machine code and saved in a file, ready to be downloaded. *Code Files* are particularly useful when it comes to making duplicates of the same application to run on several SMART1s. Another scenario might be that we need to ship an updated program to a customer to enhance the SMART1's performance. In this case we would send him a *Code File* instead of the project in form of individual program modules. This way we won't have to reveal our programming secrets to the customer and it also makes it very easy for him to get the new program into the SMART1. Here is what he or we would do:

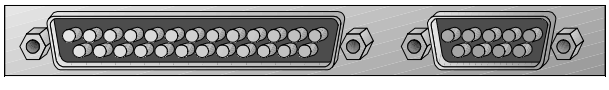


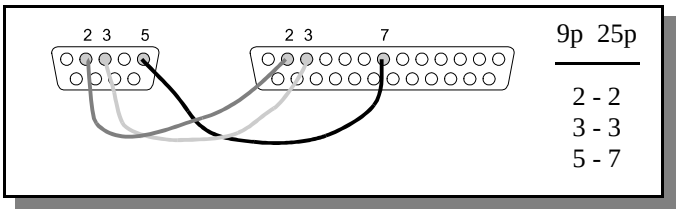
To load a *Code File* we simply mouse-click our way through the ***File*** Menu via ***Compiled Code*** to ***Load code file***. A file dialog box appears, showing us the current disk drive, a list of Folders (in Windows 3.11 called Directories) and all the files in the currently open folder that have names ending with **.s19** . These are our *Code Files*. We select the one we want to load by clicking on it and then press the *OK* button. If the file we are looking for is not in the list, then this can be due to several reasons. There might be too many files to fit in the window. In this case the slider bar becomes active and by clicking its up or down arrow we can scroll through the list until our file name appears. It could also be that the *Code File* was saved in a different folder or even on another disk. Then we have to select the proper drive and/or folder from the "Drives" or "Folders" list. Let's assume we had received a *Code File* on a floppy disk and it's sitting in drive *A* of our computer. Clicking the tab on the "Drives" box will give us a listing of all disk and network drives we can access. Here again, if the space does not allow for showing everything at once, we can use a slider bar to move through the list. We scroll up until drive **a:** is displayed and then click on it. The folders and file lists get updated and now we can load the *Code File*. The second part of the updating process, the actual downloading of the code is described in the chapter **Sending Code to the SMART1**.

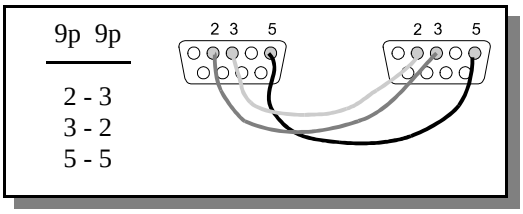


Sending Code to the SMART1

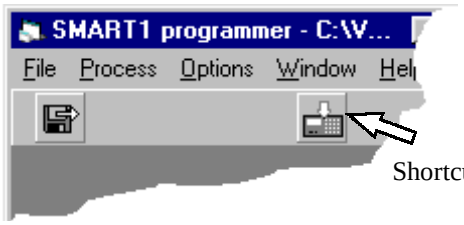
We now have a compiled project sitting in our computer's memory. What next? We have to get the code into the SMART1. Do do this we need to run a cable from it's serial port 0 (that's the one farthest away from the power jack) to a free serial port on our computer. Most computers are equipped with two serial ports. They are usually located on the back and are laid out as either a male connector with 25 pins or a male connector with 9 pins. The example on the right shows them sitting next to one another, but this is not always the case. One of the ports is typically used for the mouse. The other one we will connect the SMART1 to. The cable we need is called a "Null Modem Cable". It must have a 9 pin female connector on the end that plugs into the SMART1. The connector on the other end has to fit the computer's serial port. We can either buy the cable in a computer store or make our own simplified null modem cable with the three wires we need for this job. Here are the pin-out versions for the different connector types:



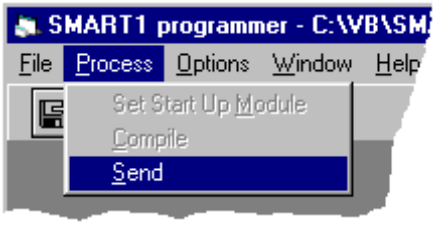




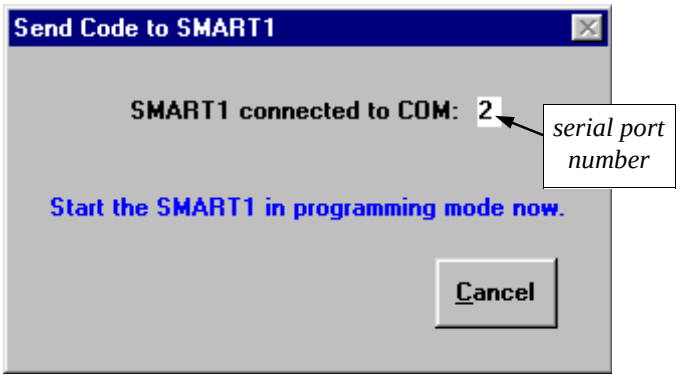
Once the cable is in place we get back to the SMART1 Programmer. Either by pushing the shortcut button or by clicking ***Send*** in the ***Process*** menu, we bring up the ***Send Window***.



Shortcut button "Send"



If the serial port number we plugged the SMART1 into is different from the number shown in this window, then we have to click on the number and change it. We are prompted to start the SMART1 in programming mode. This is accomplished by applying power to the unit while pressing its programming button. From here everything should be going by itself. The project's individual components are downloaded and as soon as this is done, the Send Window will disappear and the word **DONE** will be shown on the SMART1's display. The final step is to pull the power connector from the unit and to plug it back in. Our program has been updated and is now ready to run.



Tips and Tricks, Dos and Don’ts

Moving Instructions to a New Module

Since the features for copying and pasting of instructions didn't make their way into version 1.00 of the SMART1 Programmer, we can't move instructions from one module to another. There is, however, a **Trick** we can use, if the current module is getting too crowded and we want to place parts of it in a new module. The procedure is like this: First, we save the module. Then we save it under a different name, thereby renaming it. The third step is to load the module with the old name back into our project. Now we have two identical program modules with different names. All that's left to do, is to get rid of the instructions we don't need in the individual modules.

Retaining Data during Re-Programming

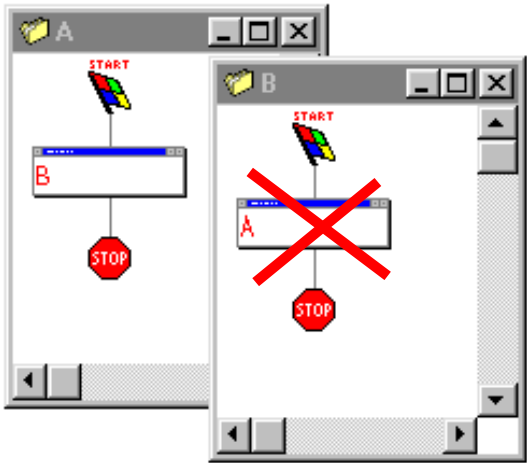
Every time a project is compiled, the SMART1 Programmer calculates the space required for all variables and files. According to these calculations it will assign memory addresses in such a way that the next item is always placed onto the next available memory location.

Let's imagine what will happen, if we are re-programming a SMART1 because our customer wanted us to change a certain part of the project. As long as we are neither changing any variables nor modifying any files, we'll be fine. The trouble starts when we start to add, remove or change the sizes of variables or files. All of a sudden some of our variables might have weird contents and our files might not work at all. Because we changed one or more variables, everything located above these variables will have moved to different spots in memory and therefore have different contents.

Now that we know about the danger, let's get some **Tips** on what to do against it. First we must know that memory is distributed in the same order in which the program modules are loaded into the project. This means, that the files and variables in the module that is processed first will always be placed at the start of the memory. So, we simply place all the files and variables that will contain vital data into this module. Everything processed after that will now have no influence on this data. We are not required to include any executable instructions in this module. To find out which module is the first one to process, we can look at the list of modules in the ***Windows*** menu. It's the one on the top. It does not neccessarily have to be the project's ***Startup Module***, so let's not get them confused! We can designate another module as the first one to process, if we don't want to use the current one. To do this we must activate the module of our choice by clicking on it. Then we have to press and hold the ***Ctrl*** key and push the ***F1*** key at the same time. The project will be reloaded in the new order.

A scenario that can't be taken care of quite that easily, will arise if we have to change the structure of a file itself. If we can foresee that a situation like this could emerge, then it's good advice to include some functions in the project that are capable of uploading/ downloading file data to and from a PC. This way we can upload the existing data, modify it for the new structure and then download it to the re-programmed SMART1. The SMART1 Programmer installation disk includes a little example project, demonstrating download and upload functions. The disk also includes a utility program for the PC called ***SMART1 Data Loader***. It can be used to receive file data from a SMART1, modify it and send it back. We can install both these items by selecting the ***Custom Install*** option in the installation program.

Recursive Calls



A definite **Don't** is to use the ***Call a Module*** instruction to go back to a higher ranking program module. The people among us who entered the realms of programming only recently, could be easily trapped by this possibility. The only proper way to return to the calling module is to apply the ***Module Exit*** instruction. Whenever a call to a module is executed, the address of the calling instruction is stored in the so called stack. This is an area in the SMART1's memory chip located adjacent to the variables and files. When the module has finished its task the address is pulled from the stack and the program continues where it left off. If, however, the program flow in the called module never gets to a ***Module Exit*** instruction, but instead calls the module which called it in the first place, the stack is sooner or later going to flow over. This means, it will expand into the memory used for files and variables. The moment at which this will happen depends on how often the calling loop is being executed and how much space is occupied by the project's variables and files. If the SMART1 is frequently turned off, it might never happen. If it does, though, the program will be stopped before any data is lost and an error message will be shown on the screen. Turning the SMART1 on and off will restart the program. It is going to work until the stack runs over again.

Detecting a Number Overflow

We have read in the chapters about *Number Variables* and the ***Math Instruction***, that numbers can only be displayed as long as they are not bigger than what their maximum length allows, and only if their numeric resolution doesn't exceed the value 16777215. Well, sometimes we might get into a situation, where adding up numbers could lead to such an overflow. One thing we can **Do** to keep the display from putting a row of stars in place of our number, is to ***Compare*** the number to the largest displayable value before it's shown. If it is greater than that value, we have to use a number that was declared with a larger capacity. In case our number already had a length of 8 or more and wasn't using decimal places, then we could divide it by 10, convert it into a ***String*** and append a "dead" zero to its end. Doing this, we will only lose the value of the number's last digit as compared to the whole number.

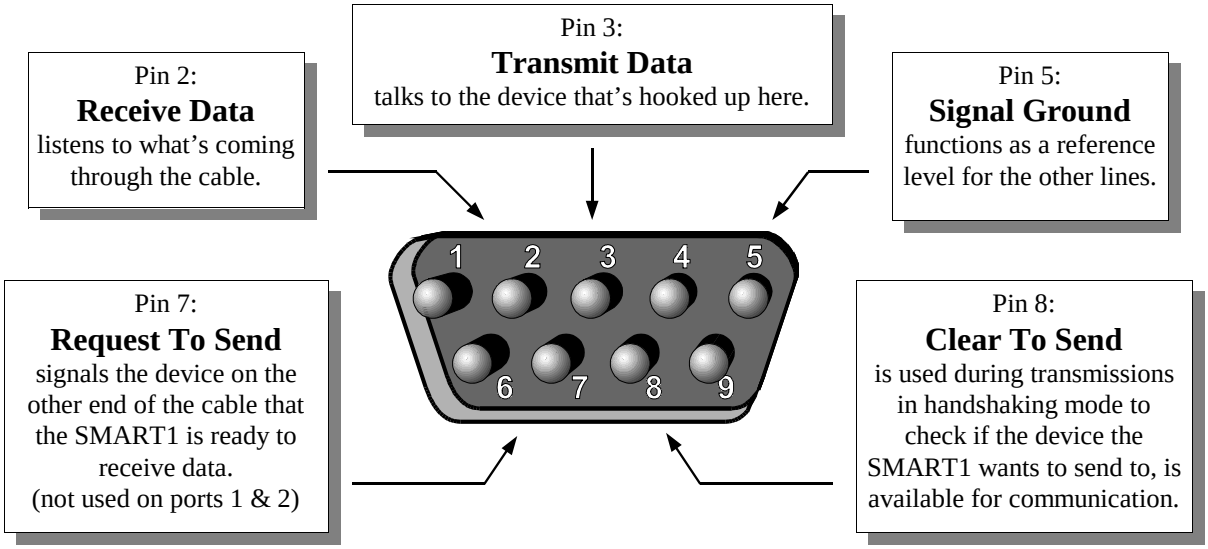
Transmitting to a Printer

As we probably remember from reading the chapter about serial transmissions, serial ports 1 through 4 will "swallow" the first character without reporting a possible "printer busy" status back to the SMART1. What we can **Do** in order to not be fooled by this behaviour, is simply to send two ***Carriage Return*** characters whenever we need to test the printer's status. They won't cause any actual printing, but still give us the result we're aiming for.

One final **Tip** regarding printers: Let's not forget that most of them will only print the received data if it is followed by ***Carriage Return*** and/or ***Line Feed***.

APPENDIX

Serial Port Pins



Pins 1,4,6 and 9 have no particular purpose in any of the SMART1's serial communication ports.

I/O Expansion Cable

The instruction types for Output Relays, 4-20 mA Outputs and Remote Inputs will function only if their respective hardware components are connected to the SMART1's I/O expansion port. A cable assembly for this intention is available from Western Scale.

For trouble shooting purposes, here's how the wires should connect the SMART1 to the expansion board sets A and B: