The VVM Program
Editor Screen

This screen allows you to edit a new or existing VVM language program in the blue editor window. Programs can be in Assembly Language or in Machine Language format. See the help topic "VVM Language" for details on these programming languages. **Program statements must be the first non-blank characters on a line. Operands are entered as unsigned two-digit integers. Data values are entered as signed or unsigned three-digit integers**. Program lines beginning with "//" are treated as comments and are ignored.

Remaining characters following a program statement are ignored, and may be used for line numbers, to document memory addresses, or for program comments. Extra space characters are also ignored and can be used to indent statements or comments. Program statements can be entered in upper-case or in lower-case letters.

By default, programs are loaded in sequential addresses starting at 00. Memory load address can be adjusted with the *load directive* "*nn", where "nn" is a two-digit address. When this directive is found, subsequent statements are loaded beginning with that new address. Prefixing a program statement with "nn:" will also adjust the load address of that and subsequent statements.

A simple VVM Assembly Language program which adds an input value to the constant value -1 is shown below:

```
//       A sample VVM Assembly program
//       to add a number to the value -1.
IN       Input number to be added
ADD 99   Add value stored at address 99 to input
OUT      Output result
HLT      Halt (program ends here)
*99      Next value loaded at address 99
DAT -001 Data value
```

This same program could be entered in VVM Machine Language format as follows:

```
//   The Machine Language version
901  Input number to be added
199  Add value stored at address 99 to input
902  Output result
000  Halt (program ends here)
*99  Next value loaded at address 99
-001 Data value
```

The **AutoSyntax** menu (or right-click in the Editor Window) can be used to enter program statements using the mouse. This menu eliminates the need to memorize the VVM language syntax.

Use the **File** menu to store, retrieve, and print program files. Files may use a .VVM or a .TXT file extension. A sample program file named "SAMPLE.VVM" is included for your use.

Use the **Validate** button to check the syntax of your program. This action will display a Program Address Map window showing how your program will be mapped to the VVM memory space. After a program has been validated, you can use the **Load** button to load it into the Visible Virtual Machine.

The **Zero Out Memory Before Load** option determines whether the VVM RAM is initialized to zero values prior to the program load operation. When this option is deselected, RAM locations are initialized to randomly generated values before the program is loaded.

Context sensitive popup menus, activated through a right-button mouse click, are available throughout the Program Editor Screen.

The VVM Program
Execution Screen

This screen allows you to execute your program while observing the hardware mechanisms at work. See the help topic "VVM Machine" for details on the Visible Virtual Machine hardware and operation. The program which was created in the editor screen is initially loaded into RAM starting at address 00. This is called the *initial program state*.

User Activated Controls

The user controllable features of the machine are located in the lower left corner of the screen and in the menu selections in the upper left. Context sensitive popup menus, activated through a right-button mouse click, are available throughout the Execution Screen. The **Return** button returns you to the Editor Screen where you can revise and reload your VVM program, or exit the system.

Five additional controls manipulate the execution of your program:

- **Run/Resume** button. This control begins or resumes execution of the program from its *current state*. The button is labeled **Run** if the program is in its initial load state, while it is labeled **Resume** if program execution has begun. Program execution continues until the program ends either normally, through a fatal VVM Machine error, or because it is paused or canceled by the user.

- **Step** button. This control executes the current instruction and fetches the next instruction. Control of the machine is then returned to the user. This button allows you to walk through all or part of your program one step at a time.

- **Pause** button. This button interrupts the execution of the program and returns control of the machine to the user. The interruption occurs after the next instruction is *fetched*, but before it is *executed*.

- **Restart** button. Use this button to restart the program from its beginning (see the *Reload @ Restart* option, below).

- **Speed** control. This control allows you to control the speed at which the program is executed. It does not come into play while **Step**ping through the program. Using this control, you can add a delay of up to five seconds to each program instruction. By default, the control is set for a one second delay. The delay occurs after the fetch, but before the execution of an instruction. If the delay is set to one second or more, then an hourglass symbol shows next to the mouse pointer while the machine is waiting. The exact delay time can be displayed by resting the mouse pointer on the "speed" label of this control. Each t or u click changes the delay by 0.5 seconds.

- **Show Source Window** option. The Source Program Window displays the original (source) program that is currently loaded in the VVM hardware. The load address of each executable statement or program data value is also shown. As your program is executed in the VVM machine, each "current" instruction is highlighted in yellow, thus illustrating how the individual machine actions relate to your original source program. As source instructions are executed they are tagged with an "**x**" symbol in the program listing. This tag can be used to indicate the execution of loops, or to identify which conditional paths have been taken in the program. This window can be resized or moved as desired.

  Note: If any instruction or data value from the source program is modified by the execution of the program, its load address is changed to red in the Source Program Window to indicate this modification.

- **Tick** option. This option determines whether an audible "tick" sound is generated with each instruction (the sound is quelled at faster execution speeds). The sound used is the "Windows Default Sound" as set through the Windows Control Panel.

The Machine Views

By selecting the appropriate tabs, you can configure the right half of the screen to provide either of two

different views of the program execution. These are the **Hardware View** and the **Trace View**.

Hardware View provides a visual representation of various hardware components of the machine while your program is executing. Trace View shows the details of the individual instructions of your program as they are executing. Both views are detailed below.

The Hardware Components (Hardware View)

Hardware View shows the various hardware components of the VVM machine as your program is executing. These are described below, moving from the top left in clockwise order.

- **I/O Log**. This represents the system console which shows the details of relevant events in the execution of the program. Examples of events are the program begins, the program aborts, or input or output is generated. Each entry begins with a two-digit number in square brackets. This is the address of the instruction that caused the event.

- **Accumulator** Register* (Accum.). This displays the current value of the Accumulator. Legitimate values are any integer between -999 and +999. Values outside of this range will cause a fatal VVM Machine error. Non integer values are converted to integers before being loaded into the register.

- **Instruction Cycle** Display*. This shows the number of instructions that have been executed since the program began.

- **Instruction Register*** (Instr. Reg.). This register holds the next instruction to be executed. The register is divided into two parts: a one-digit *operation code*, and a two digit *operand*. The Assembly Language mnemonic code for the operation code is displayed below the register.

- **Program Counter** Register* (Prog. Ctr.). The two-digit integer value in this register "points" to the next instruction to be fetched from RAM. During the *fetch* phase of the instruction cycle, which is most often visible on the screen, the value of the register is the RAM address of the instruction which is currently in the Instruction Register. Legitimate values range from 00 to 99. A value beyond this range causes a fatal VVM Machine error.

- **RAM***. The 100 *data-word* Random Access Storage is shown as a matrix of ten rows and ten columns. The two-digit memory addresses increase sequentially across the rows and run from 00 to 99. Each storage location can hold a three-digit integer value between -999 and +999. Whenever a storage location is referenced, either by loading the program, or by the execution of the program, its background is changed to white. When an instruction is fetched from RAM and copied to the Instruction Register, its value in RAM is shown in bold.

* This hardware component supports the *How's it work?* feature (see below).

Trace View

The Trace View window provides a history of the execution of your program. Prior to the execution of each statement, the window shows:

1. The instruction cycle count (begins at 1)

2. The address from which the instruction was fetched

3. The instruction itself (in VVM Assembly Language format)

4. The current value of the Accumulator Register

The "*How's it work?*" Feature

The active components of the VVM CPU (registers, displays and RAM locations) support a special feature

called *How's it work?*. When the mouse pointer is placed over any of these components, the pointer is changed to include a "**?**" symbol. This indicates that the feature is available.

Clicking on a *How's it work?* component produces a description of how that specific device performs within the context of its current value. As a program executes, the description of each component will change to reflect the current program context.

Other Configuration Options

- **Reload @ Restart** option (on Execute menu). When this option is selected, which is the usual case, **Restart**ing the program invokes a full program load. In other words, both RAM and the registers are restored to the *initial program state*. When this option is deselected, **Restart**ing the program causes the registers to be reset, but maintains RAM in its *current state*. This option is reset to its default (selected) state each time a program is loaded or executed.

- **MessageBox @ Output** option (on Configure menu). When this option is selected (default), each output operation results in a message window announcing the output. This option can be turned off in the instance when a large amount of output is generated by a program.

- **Preserve Configuration** option (on Configure menu). This option controls whether the various VVM machine options are maintained when returning to and from the VVM Editor.

Hard Copy Output

The **File** menu provides two printing options which can document the *current state* of the VVM machine. The **Print Dump** option (*Ctrl-D*) produces a text listing of the current state of all relevant hardware elements (i.e., registers, active RAM locations, etc.). The **Print Trace** option (*Ctrl-T*) produces a listing of the program trace as shown in the Trace View window. This is, in essence, the history of the execution of your program.

The Debug Menu

In addition to the *Dump* and *Trace* printouts described above, the **Debug** menu provides two program debugging helps. These mechanisms are *Address Traps* (program *breakpoints* and data *watchpoints*), and the *Modify RAM Value facility*.

- **Address Traps**. When an *Address Trap* is placed on a RAM address, that RAM location is made sensitive to program activity. When the content of a trapped address is used as a program instruction, it is called a *breakpoint* because program execution *break*s at that *point*. When the content of a trapped address is used as a data value it is called a *watchpoint* because the system *watch*es that *point* in memory for activity. Whenever the next instruction to be executed either contains a *breakpoint*, or modifies a *watchpoint* value, the program is interrupted prior to the execution of that instruction. The user can then either **Step** through, or **Resume** execution of the program to observe the effect of that specific instruction. Address Traps are indicated in red in the VVM RAM.

- **Modify RAM Value** facility. This mechanism allows the user to modify the value of any RAM location after the program has been loaded, and even after program execution has begun. Both instructions and data values can be modified. The change stays in effect until the program is either reloaded into memory, or until the address is modified by the program.

The VVM Language

The Visible Virtual Machine (VVM) is based on a model of a simple computer device called the Little Man Computer which was originally developed by Stuart Madnick in 1965, and revised in 1979. The revised Little Man Computer model is presented in detail in "The Architecture of Computer Hardware and System Software" (2'nd), by Irv Englander (Wiley, 2000). The programming language of this machine has eleven different executable operations. There are two forms of the language - Machine, and Assembly.

In the Machine Language format, each instruction is a three-digit integer where the first digit specifies the operation code (op code), and the remaining two digits represent the (required) operand. In the Assembly Language format, the operation code is replaced by a three-character mnemonic code. The two-digit operand is optional in some instances in the Assembly version. When the operand is used, it is separated from the mnemonic operation code by a space character. In both language versions, the two-digit operand usually represents a memory address.

The Language Instructions

The eleven operations are described below. The Machine Language codes are shown in parentheses, while the Assembly Language version is in square brackets.

- **Load Accumulator (5*nn*) [LDA *nn*]** The content of RAM address *nn* is copied to the Accumulator Register, replacing the current content of the register. The content of RAM address *nn* remains unchanged. The Program Counter Register is incremented by one.

- **Store Accumulator (3*nn*) [STO *nn*]** or **[STA *nn*]** The content of the Accumulator Register is copied to RAM address *nn*, replacing the current content of the address. The content of the Accumulator Register remains unchanged. The Program Counter Register is incremented by one.

- **Add (1*nn*) [ADD *nn*]** The content of RAM address *nn* is added to the content of the Accumulator Register, replacing the current content of the register. The content of RAM address *nn* remains unchanged. The Program Counter Register is incremented by one.

- **Subtract (2*nn*) [SUB *nn*]** The content of RAM address *nn* is subtracted from the content of the Accumulator Register, replacing the current content of the register. The content of RAM address *nn* remains unchanged. The Program Counter Register is incremented by one.

- **Input (901) [IN]** or **[INP]** A value input by the user is stored in the Accumulator Register, replacing the current content of the register. Note that the two-digit operand does not represent an address in this instruction, but rather specifies the particulars of the I/O operation (see Output). The operand value can be omitted in the Assembly Language format. The Program Counter Register is incremented by one with this instruction.

- **Output (902) [OUT]** or **[PRN]** The content of the Accumulator Register is output to the user. The current content of the register remains unchanged. Note that the two-digit operand does not represent an address in this instruction, but rather specifies the particulars of the I/O operation (see Input). The operand value can be omitted in the Assembly Language format. The Program Counter Register is incremented by one with this instruction.

- **Branch if Zero (7*nn*) [BRZ *nn*]** This is a conditional branch instruction. If the value in the Accumulator Register is zero, then the current value of the Program Counter Register is replaced by the operand value *nn* (the result is that the next instruction to be executed will be taken from address *nn* rather than from the next sequential address). Otherwise (Accumulator >< 0), the Program Counter Register is incremented by one (thus the next instruction to be executed will be taken from the next sequential address).

- **Branch if Positive or Zero (8*nn*) [BRP *nn*]** This is a conditional branch instruction. If the value in the Accumulator Register is nonnegative (i.e., >= 0), then the current value of the Program Counter Register is replaced by the operand value *nn* (the result is that the next instruction to be executed will be taken from address *nn* rather than from the next sequential address). Otherwise (Accumulator < 0), the Program Counter Register is incremented by one (thus the next instruction to be executed will be taken from the

next sequential address).

- **Branch (6*nn*) [BR *nn*]** or**[BRU *nn*]** or **[JMP *nn*]** This is an unconditional branch instruction. The current value of the Program Counter Register is replaced by the operand value *nn*. The result is that the next instruction to be executed will be taken from address *nn* rather than from the next sequential address. The value of the Program Counter Register is not incremented with this instruction.

- **No Operation (4*nn*) [NOP]** or **[NUL]** This instruction does nothing other than increment the Program Counter Register by one. The operand value *nn* is ignored in this instruction and can be omitted in the Assembly Language format. (This instruction is unique to the VVM and is not part of the Little Man Model.)

- **Halt (0*nn*) [HLT]** or **[COB]** Program execution is terminated. The operand value *nn* is ignored in this instruction and can be omitted in the Assembly Language format.

Embedding Data in Programs

Data values used by a program can be loaded into memory along with the program. In Machine or Assembly Language form simply use the format "*snnn*" where *s* is an optional sign, and *nnn* is the three-digit data value. In Assembly Language, you can specify "DAT *snnn*" for clarity.

The VVM Load Directive

By default, VVM programs are loaded into sequential memory addresses starting with address 00. VVM programs can include an additional load directive which overrides this default, indicating the location in which certain instructions and data should be loaded in memory. The syntax of the Load Directive is "***nn***" where *nn* represents an address in memory. When this directive is encountered in a program, subsequent program elements are loaded in sequential addresses beginning with address *nn*.

A Sample VVM Program

A simple VVM Assembly Language program which adds an input value to the constant value -1 is shown below (note that lines starting with "//" and characters to the right of program statements are considered comments, and are ignored by the VVM machine).

```
//       A sample VVM Assembly program
//       to add a number to the value -1.
IN       Input number to be added
ADD 99   Add value stored at address 99 to input
OUT      Output result
HLT      Halt (program ends here)
*99      Next value loaded at address 99
DAT -001 Data value
```

This same program could be written in VVM Machine Language format as follows:

```
//   The Machine Language version
901  Input number to be added
199  Add value stored at address 99 to input
902  Output result
000  Halt (program ends here)
*99  Next value loaded at address 99
-001 Data value
```

The VVM Machine

The Visible Virtual Machine (VVM) is based on a model of a simple computer device called the Little Man Computer which was originally developed by Stuart Madnick in 1965, and revised in 1979. The revised Little Man Computer model is presented in detail in "The Architecture of Computer Hardware and System Software" (2'nd), by Irv Englander (Wiley, 2000).

The VVM is a virtual machine because it only appears to be a functioning hardware device. In reality, the VVM "hardware" is created through a software simulation. One important simplifying feature of this machine is that it works in decimal rather than in the traditional binary number system. Also, the VVM works with only one form of data - decimal integers.

Hardware Components

The VVM machine comprises the following hardware components:

- **I/O Log**. This represents the system console which shows the details of relevant events in the execution of the program. Examples of events are the program begins, the program aborts, or input or output is generated.

- **Accumulator Register** (Accum). This register holds the values used in arithmetic and logical computations. It also serves as a buffer between input/output and memory. Legitimate values are any integer between -999 and +999. Values outside of this range will cause a fatal VVM Machine error. Non integer values are converted to integers before being loaded into the register.

- **Instruction Cycle Display**. This shows the number of instructions that have been executed since the current program execution began.

- **Instruction Register** (Instr. Reg.). This register holds the next instruction to be executed. The register is divided into two parts: a one-digit *operation code*, and a two digit *operand*. The Assembly Language mnemonic code for the operation code is displayed below the register.

- **Program Counter Register** (Prog. Ctr.). The two-digit integer value in this register "points" to the next instruction to be fetched from RAM. Most instructions increment this register during the *execute* phase of the instruction cycle. Legitimate values range from 00 to 99. A value beyond this range causes a fatal VVM Machine error.

- **RAM**. The 100 *data-word* Random Access Storage is shown as a matrix of ten rows and ten columns. The two-digit memory addresses increase sequentially across the rows and run from 00 to 99. Each storage location can hold a three-digit integer value between -999 and +999.

Data and Addresses

All data and address values are maintained as decimal integers. The 100 data-word memory is addresses with two-digit addressed in the range 00-99. Each memory location holds one data-word which is a decimal integer in the range -999 - +999. Data values beyond this range cause a data overflow condition and trigger a VVM system error.

VVM System Errors

Various conditions or events can cause VVM System Errors. The possible errors and probable causes are as follows:

- **Data value out of range**. This condition occurs when a data value exceeds the legitimate range -999 - + 999. The condition will be detected while the data resides in the *Accumulator Register*. Probable causes are an improper addition or subtraction operation, or invalid user input.

- **Undefined instruction**. This occurs when the machine attempts to execute a three-digit value in the *Instruction Register* which can not be interpreted as a valid instruction code. See the help topic "VVM

Language" for valid instruction codes and their meaning. Probable causes of this error are attempting to use a data value as an instruction, an improper *Branch* instruction, or failure to provide a *Halt* instruction in your program.

- **Program counter out of range**. This occurs when the Program Counter Register is incremented beyond the limit of 99. The likely cause is failure to include a *Halt* instruction in your program, or a branch to a high memory address.

- **User cancel**. The user pressed the "Cancel" button during an *Input* or *Output* operation.

VVM Debugging Aids
(Execution Screen)

Hard Copy Output

The **File** menu provides two printing options which can document the *current state* of the VVM machine. The **Print Dump** option (*Ctrl-D*) produces a text listing of the current state of all relevant hardware elements (i.e., registers, active RAM locations, etc.). The **Print Trace** option (*Ctrl-T*) produces a listing of the program trace as shown in the Trace View window. This is, in essence, the history of the execution of your program.

The Debug Menu

In addition to the *Dump* and *Trace* printouts described above, the **Debug** menu provides two program debugging helps. These mechanisms are *Address Traps* (program *breakpoints* and data *watchpoints*), and the *Modify RAM Value facility*.

- **Address Traps**. When an *Address Trap* is placed on a RAM address, that RAM location is made sensitive to program activity. When the content of a trapped address is used as a program instruction, it is called a *breakpoint* because program execution *break*s at that *point*. When the content of a trapped address is used as a data value it is called a *watchpoint* because the system *watch*es that *point* in memory for activity. Whenever the next instruction to be executed either contains a *breakpoint*, or modifies a *watchpoint* value, the program is interrupted prior to the execution of that instruction. The user can then either **Step** through, or **Resume** execution of the program to observe the effect of that specific instruction. Address Traps are indicated in red in the VVM RAM.

- **Modify RAM Value** facility. This mechanism allows the user to modify the value of any RAM location after the program has been loaded, and even after program execution has begun. Both instructions and data values can be modified. The change stays in effect until the program is either reloaded into memory, or until the address is modified by the program.

The "*How's it work?*" Feature

The active components of the VVM CPU (registers, displays and RAM locations) support a special feature called *How's it work?*. When the mouse pointer is placed over any of these components, the pointer is changed to include a "**?**" symbol. This indicates that the feature is available.

Clicking on a *How's it work?* component produces a description of how that specific device performs within the context of its current value. As a program executes, the description of each component will change to reflect the current program context.

VVM System Errors

Various conditions or events can cause VVM System Errors. The possible errors and probable causes are as follows:

- **Data value out of range**. This condition occurs when a data value exceeds the legitimate range -999 - + 999. The condition will be detected while the data resides in the *Accumulator Register*. Probable causes are an improper addition or subtraction operation, or invalid user input.

- **Undefined instruction**. This occurs when the machine attempts to execute a three-digit value in the *Instruction Register* which can not be interpreted as a valid instruction code. See the help topic "VVM Language" for valid instruction codes and their meaning. Probable causes of this error are attempting to use a data value as an instruction, an improper *Branch* instruction, or failure to provide a *Halt* instruction in your program.

- **Program counter out of range**. This occurs when the Program Counter Register is incremented beyond the limit of 99. The likely cause is failure to include a *Halt* instruction in your program, or a branch to a high memory address.

- **User cancel**. The user pressed the "Cancel" button during an *Input* or *Output* operation.