

Leopard II Manual

Table of Contents

1	OVERVIEW OF THE LEOPARD II	3
1.1	HARDWARE DESCRIPTION	3
1.2	HARDWARE SPECIFICATIONS.....	3
1.3	SOFTWARE SPECIFICATIONS	4
1.4	INSTALLING THE LEOPARD II	4
1.4.1	<i>Hardware Installation</i>	4
1.4.1.1	Power.....	6
1.4.1.2	Connections for the built-in and expansion I/O interfaces.....	7
1.5	INSTALLING THE C-MAX UTILITY	8
1.5.1	<i>Installing from a CD</i>	8
1.5.2	<i>Installing from a compressed file</i>	8
1.5.3	<i>C-Max Installation</i>	9
1.5.4	<i>Configuring C-Max</i>	9
1.6	WRITING YOUR FIRST PROGRAM	13
1.6.1	<i>Creating a Project</i>	13
1.6.2	<i>The System Map</i>	14
1.6.3	<i>The Touch screen Layout</i>	15
1.6.4	<i>Writing the Program Code</i>	18
2	PROGRAMMING THE LEOPARD II	24
2.1	THE C-MAX PROGRAMMING MODEL	24
2.2	LADDER LOGIC.....	24
2.3	PROGRAM FLOW RULES	24
2.4	PRECEDENCE OF LOGIC TESTS.....	26
2.5	SINGLE AND CONTINUOUS EVENTS	27
2.6	TIMING IS EVERYTHING	28
2.7	C-MAX VARIABLES.....	29
3	C-MAX COMMAND REFERENCE.....	31
4	C-MAX USER'S GUIDE.....	51
4.1	PROJECTS	51
4.1.1	<i>Creating a Project</i>	51
4.2	THE SYSTEM MAP	52
4.3	THE PROGRAM EDITOR	54
4.3.1	<i>Editing Commands</i>	55
4.4	SAVING/LOADING/PRINTING PROGRAMS	57
4.5	THE TOUCH SCREEN EDITOR.....	57
4.5.1	<i>Adding and Editing Buttons</i>	59
4.5.2	<i>Adding and Editing Status Lights</i>	61
4.5.3	<i>Adding and Editing Screen Text</i>	63
4.5.4	<i>Managing Touch Screen Files</i>	64
4.5.4.1	Individual Screen Saving, Loading;.....	64
4.5.4.2	New Touch Screen File	64
4.5.4.3	Open Touch Screen File	65
4.5.4.4	Save Touch Screen Files As	65
4.5.4.5	Project Screen Saving, Loading.....	65
4.6	C-MAX UTILITIES	65
4.6.1	<i>Infra-Red</i>	66
4.6.1.1	Learn Infra-Red Command.....	67
4.6.1.2	Transmit Infra-Red Command.....	68
4.6.1.3	Transmit Remote Infra-Red Command	68
4.6.1.4	Download Infra-Red File to Controller.....	68
4.6.1.5	Upload Infra-Red File from Controller.....	68

4.6.2	<i>X10</i>	69
4.6.2.1	Send X10.....	69
4.6.2.2	Monitor X10.....	70
4.6.2.3	Send Leviton X10.....	71
4.6.3	<i>Program File</i>	73
4.6.4	<i>Controller Utility</i>	73
4.6.4.1	Set Controller Clock to PC Clock.....	73
4.6.4.2	Get Controller Clock.....	73
4.6.4.3	Reload Controller Executive.....	73
4.6.4.4	Controller Memory Dump.....	74
4.6.4.5	Retrieve Controller Parameters.....	74
4.6.4.6	Auto Address Modules.....	75
4.6.4.7	Debug Timers and Variables.....	75
4.6.5	<i>Module Utility</i>	76
4.6.5.1	Retrieve Module Parameters.....	76
4.6.5.2	Set/Clear Relay.....	77
4.6.5.3	Speak Easy.....	78
4.6.5.4	ASCII Bobcat.....	79
4.6.5.5	Serial Messages.....	79
5	APPLICATION NOTES.....	83
5.1	FORMATTING VARIABLES IN C-MAX 2.0.....	84
5.1.1	<i>Controller Variables</i>	84
5.1.2	<i>Formatting Options</i>	85
5.1.3	<i>Screen Display Example</i>	86
5.1.4	<i>Formatting Variables in ASCII Strings</i>	90
5.2	CREATING ICONS FOR LEOPARD BUTTONS.....	96
5.2.1	<i>Introduction</i>	96
5.2.2	<i>Designing an icon</i>	97
5.2.3	<i>A full screen image and hidden buttons</i>	104
5.2.4	<i>Changing icons under program control</i>	108
5.3	A SAMPLE C-MAX APPLICATION, HEAT/COOL THERMOSTAT.....	112
5.4	AUTO ADDRESSING YOUR ADICON™ 2500 SERIES.....	121
5.4.1	<i>Adding a new module to an existing installation</i>	123
5.5	USING EXPANSION MODULES.....	125
5.6	OCELOT/LEOPARD PARAMETERS.....	128

1 Overview of the Leopard II

The Leopard II is Applied Digital Inc's (ADI) second generation of the highly successful Leopard touch screen home automation (HA) controller. The Leopard II is much more than a simple touch screen for sending commands. It features a complete programming language allowing the user to execute tasks based on complex "If/Then" logic. Tasks can be activated not only from the touch screen but also by various input and output (I/O) sources, or programmed to execute automatically based on time or date criteria, or by any combination of these. The screen can also display information acquired from external sources or from internal calculations in the user program. An external computer can be connected to the Leopard II and information exchanged in real time between the two, allowing expanded capabilities like web access and user created software applications to interface to the home automation system. Finally, the Leopard II supports an ever growing list of expansion modules to add capabilities like reading digital and analog inputs, activate relays, measure temperatures, humidity, etc. and even have slave Leopard or Ocelot controllers to facilitate access to its resources.

1.1 Hardware Description

The Leopard II consists of a main central processing unit that holds the user program in non-volatile memory using flash-RAM technology. It has built in I/O interfaces for:

- X-10 devices. X-10 is a powerline communications protocol supported by multiple vendors of HA equipment.
- Infrared control. There is an Infrared (IR) receiver located on the front panel of the unit and an IR emitter can be connected to the I/O connector or jack to enable the controller to transmit IR commands. An external IR receiver can also be used with the Leopard II (with the front panel IR receiver disabled).
- Serial port. The RS-232 serial port is used to load programs and other data into the Leopard II and also serves as an interface to any external computer program that supports the Leopard II as an interface to a HA system.
- RS-485 expansion bus. The bus is used by the proprietary Adnet protocol to allow the Leopard II to communicate with the various expansion modules and slave controllers available from ADI. Like all other ADI controllers, the Leopard II can be configured as either a master or slave controller.
- LCD Touch screen. This is certainly the most visible I/O part of the unit. The screen can display a mixture of user configured buttons with captions or graphic icons, virtual "status lights" to show the status of an X-10 or I/O point, and also display text with variable output. Screen objects defined as buttons can be pressed by the user to trigger events within the user program.

If you are already familiar with the ADI Ocelot controller, then learning to use the Leopard II should be quite straightforward since the two controllers share the same basic processor and I/O core. You can think of the Leopard II as an Ocelot to which a touch screen has been added.

1.2 Hardware Specifications

Size: 7 3/8"W x 5 5/8"H x 2"D
Face plate size: 7" x 8"
LCD Screen: 4 1/8" W x 3 1/8"H
LCD Resolution: 320 x 240, monochrome

I/O: 4 Position Screw Terminals

2 pins for power (9-12 V AC/DC @ 1A)
2 pins for ADICON™ 2500 expansion modules
12 Position Screw Terminals

IR In
IR Out
RS-232

X-10
RJ11 Jack

For direct connection to TW-523
3.5 mm Stereo Jack

For connection to IR mini-emitter

Each Unit Includes: User's Guide and C-Max 2.0 Control Wizard Software
6' IR Mini-emitter
Comms Cable for LEOPARD II - PC communication
25' cable for TW523 communication (TW523 not included)
9-12V @ 1A power supply
5 designer faceplates.
Colors: Black, Gray, Tan, Marble and White.

1.3 Software Specifications

Ladder Logic programming model
4096 Program Lines
128 Variables (integer, 0 to 65535)
64 Timers (1 second resolution, 1 to 65535 seconds)
1024 IR codes
128 ASCII messages
15 Alphanumeric Pager Messages
256 Screen objects
200 Screen Icons (max)

1.4 Installing the Leopard II

1.4.1 Hardware Installation

The Leopard II requires an opening of 7 3/8" in width by 5 5/8" in height for the proper flush mount fit. The actual size of the back case unit is 7 1/4" x 5 1/2". Use the template provided to cut out the mounting hole for the case. Be careful when choosing your mounting location to avoid high voltage lines as they may cause problems with the electronics and communication of the equipment. Never bundle the communication or low voltage wiring with high voltage wiring as this may cause communication problems. Another consideration when choosing a mounting location is the necessity of getting the 12-volt, I/O and communications wiring to the Leopard II. This means planning ahead to avoid cable routing obstacles such as studs and horizontal braces in the wall. Cables should always run from the Leopard II to an easily accessible area. Making electrical supply connections that are hidden inside the wall is not recommended and can be a violation of electrical codes.

Depending on the wiring configuration you choose to use, you will need at least two sets of wires. The wiring options are discussed in the following paragraphs. The back case can be mounted in a wide variety of materials with either the flip-out wings or the optional 4 holes provided in the case. Using the flip-out wings, the mounting material thickness can be between ¼” to 1”. Be sure to leave 1” of clearance at the top and bottom inside the wall or cabinet for the flip out wings to achieve their final vertical position. To avoid damage to the electronics and the plastic material all mounting surfaces should be completely finished before the final installation of the Leopard II (ie: avoid installing it while there is still construction or finishing work being done in the room). Keep an unobstructed surface area of at least 1” around the exterior of the mounting opening to ensure the proper fit of the faceplate. Fig. 1 shows the flip out wings and the physical location of the various connectors and jacks.

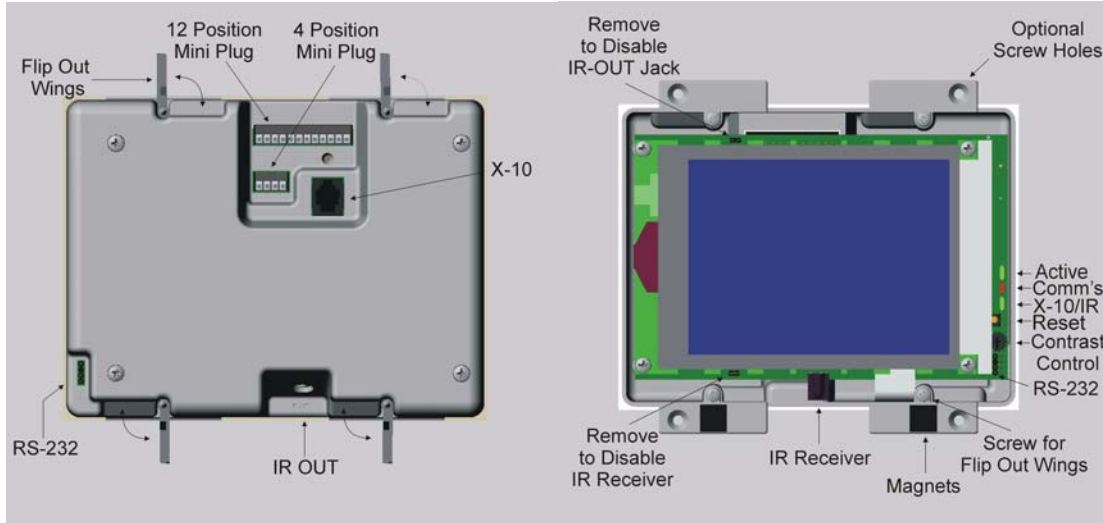


Fig. 1

Once you have located the desired mounting location for the Leopard II and have cut out the mounting hole with the provided template, it is time to check the fit of the case. Use a #2 Philips screwdriver turning in a clockwise direction to tighten the four screws attached to the flip out wings. Be careful not to over-tighten the screws, as this could cause damage to the wing and the case. If you choose to use the optional mounting holes provided, the screw head must be of the flat head type. The head of the screw must be flush or below the surface of the mounting tabs. If not, it will cause interference with the fit of the faceplate. The optional hole is based on the head of a drywall screw. Be sure to attach all of the needed wires before final installation. You can easily remove the Leopard II from the wall opening when using the flip-out wing. Loosen all of them in a counter-clockwise direction. Be careful not to loosen them too much or the wings may fall off into the wall cavity. If the screw starts to back out of case, just push it in until the head is flush. The top two will lay back down in the horizontal recess. The bottom ones may not flip back so tilt the top of the case out first, which will allow you to remove the unit.

To complete the installation, choose one of the provided faceplates. Hold it by the bottom two corners with your thumb and index fingers. Touch the top of the faceplate slightly above the top tabs letting the faceplate slide down until the hooks catch. Then lower the bottom corner down until the faceplate is flush with the surface. The magnets will hold the faceplate closed. Fig. 2 shows how the faceplate hooks onto the unit's mounting tabs.

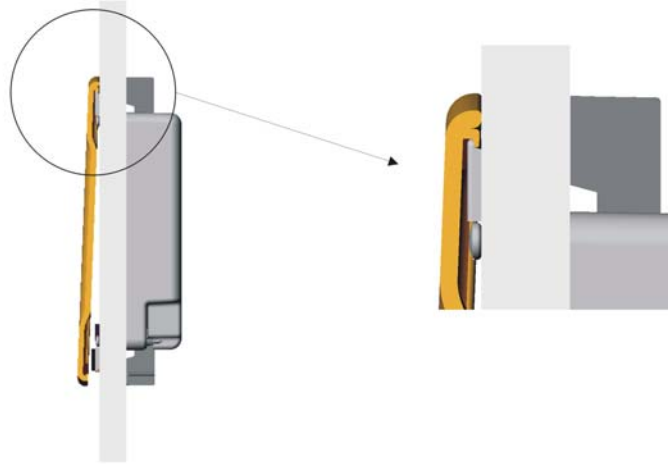


Fig. 2

1.4.1.1 Power

The Leopard II requires either a 12-volt AC or DC 1 Amp (1000mA) power supply. A typical installation uses the power supply plugged into a wall outlet and the 12-volt wire runs through the wall to the unit. The wire may be extended using 18-gauge wire up to 100 ft. The 12-volt power is connected to the 4-position mini plug on the back of the Leopard II. A 3/32" slotted screwdriver is required for securing wires to this plug. When you are looking at the back of the Leopard II with top up, the power will be the left two terminals of the 4-position plug. Polarity of the power wires is not important for this unit. All connection terminal designations for the Leopard II are shown in figures 3 and 4.

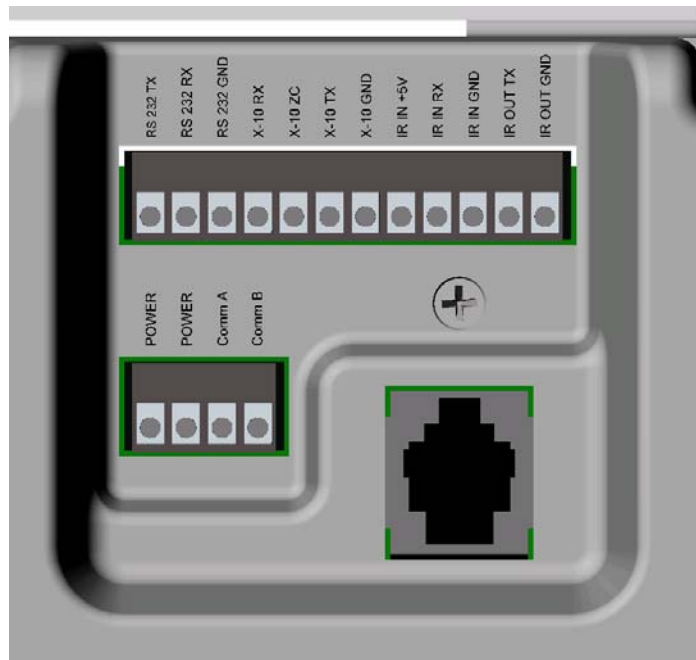
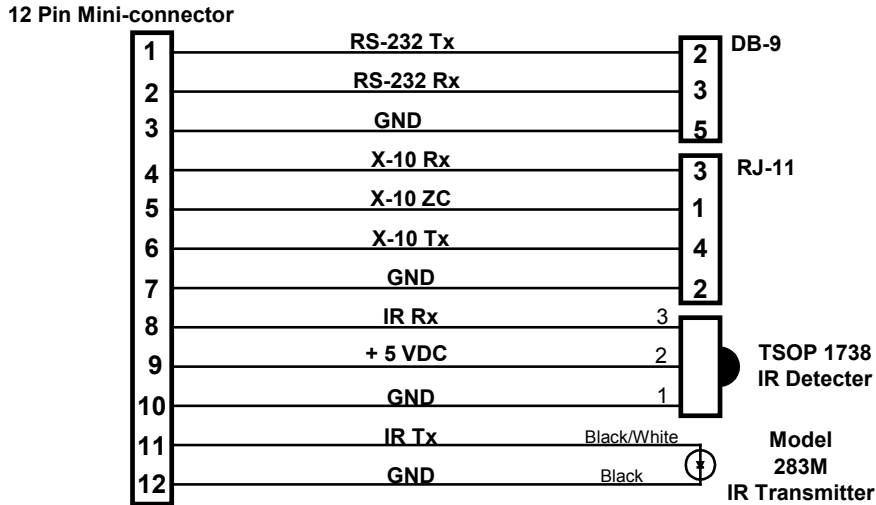
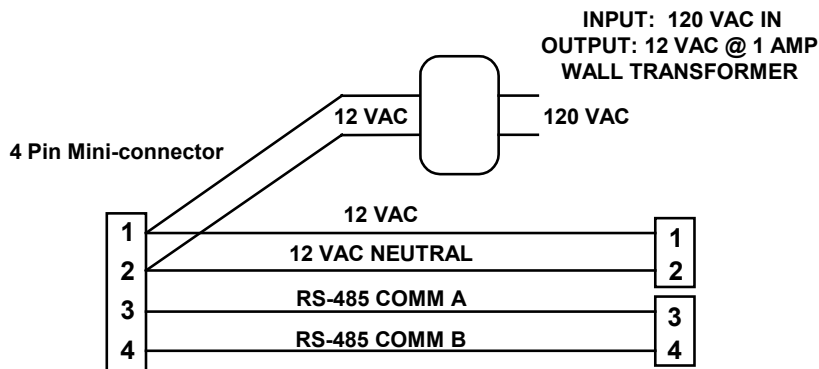


Fig 3



SIGNAL CABLE



POWER & COMM CABLE

LEOPARD II INTERFACE CABLES

Fig 4

1.4.1.2 Connections for the built-in and expansion I/O interfaces

X-10

The RJ11 jack is for the X-10 communications. Connect the supplied RJ11 cable to the jack, and then plug the other end into a PSC05 or TW523 X-10 interface.

Infrared

The IR out jack is located on the bottom of the Leopard II. It is for a standard 5v mini IR emitter. The built in IR receiver is located at the bottom center of the faceplate.

Serial port

There are two places to hook up the supplied RS-232 cable. The first one is under the faceplate in the bottom right hand corner. It is a 3-pin header that is keyed to fit the small end of the cable. The other one is on the back of the unit in the bottom left hand corner. These headers are in parallel and only one can be used at a time.

RS-485 bus (Adnet)

The right two places of the 4-position mini plug are used for the Adicon 2500 RS-485 communications. The inside one is Comm A and the outside is Comm B. A 3/32" slotted screwdriver is required for this plug. These wires would be hooked to one of our Adicon 2500 modules Comm A and Comm B in a daisy chain.

Alternate connection method

The Leopard II also has a 12-position mini plug. It includes connections for RS-232, X-10, IR in and IR out, all in one plug. This plug is configured to use two Category 5 (Cat-5) cables. The left 8 positions would use one Cat-5 cable. The other four and the Comm A & B of the 4-position mini plug would use the other Cat-5 leaving one pair (two wires) as spares. The 12-volt power should be run as a separate cable. These cables can be bundled together. This allows for all of the connections to be run to a remote location. Refer to the wiring diagram for the pin out of the cable. A 3/32" slotted screwdriver is required to secure wires to this plug. Note that when alternate connection methods are possible for certain signals, only one such method should be used at a time.

1.5 Installing the C-Max Utility

Now that your hardware is installed, you will need to install and configure the C-Max utility software. The C-Max program serves as a program editor, touch screen editor, event viewing tool, and many other functions. In order to program and use the Leopard II you will first need to install C-Max on your personal computer. All specific examples given in this document are given using C-Max version 2.00e which was the current version as of this writing. Your computer needs to be running Microsoft Windows 95 or later to use C-Max.

1.5.1 Installing from a CD

Your Leopard II was packaged with a CD containing the C-Max utility. The installation from a CD is very similar to installing most other Microsoft Windows applications. Simply insert the CD in your drive and the installation program should be launched automatically. If the installation does not start by itself, browse your CD drive and look for a file named "Setup.exe" in its main directory and then open the file. You will then be prompted through the installation process. Follow the instructions in the [C-Max Installation](#) paragraph below.

1.5.2 Installing from a compressed file

If you obtained C-Max from a source such as the Internet, it will come as a self-extracting compressed file. Simply execute to file as a program and the decompression will be automatically started. The installation program will suggest that the files be written to the "C:\temp" directory but any other temporary installation directory may be used and specifying a non-existent directory will cause the directory to be created. Once that is finished, you will need to browse to the temporary directory that was used and run the "Setup.exe" file to launch the installation of C-Max. You will be prompted through the installation process. Follow the instructions in the [C-Max Installation](#) paragraph below.

1.5.3 C-Max Installation

The default installation path will usually be “C:\Program Files\ADI\Adicon2500”. Unless there is a need to install it in another directory, it is easiest to leave the default path as it is.

The installation program will overwrite any existing installation of C-Max. Any programs and touch screen files you may already have will be safely preserved. If for whatever reason you wish to keep a copy of your previous C-Max version and any programs or touch screen files created by it (because once modified with version 2.0, they will no longer be backward compatible), then copy the entire contents of the “C:\Program Files\ADI\Adicon2500” directory (or wherever you had installed it) to another directory before installing C-Max 2.0.

Note that some things like the installation path are saved in the Windows Registry, so if you decide to install the new version in a different directory than an older one, the Registry settings will now point to the new directory. This can cause unexpected behavior if you then attempt to use the old version. For example, starting up the old version and reloading the executive will locate the flash512.bin file using the path of the new version (ie: you will be reloading the new executive version instead of the old one...).

1.5.4 Configuring C-Max

Once C-Max is installed on your personal computer, there a few settings that must be finalized before you can use it. Start the C-Max utility by clicking on **Start** → **Programs** → **C-Max Control**. You can also create a shortcut for your desktop by creating a new shortcut to the “Cpuxa.exe” file in the installation directory. Once started you will get a screen similar to the following (fig. 5)

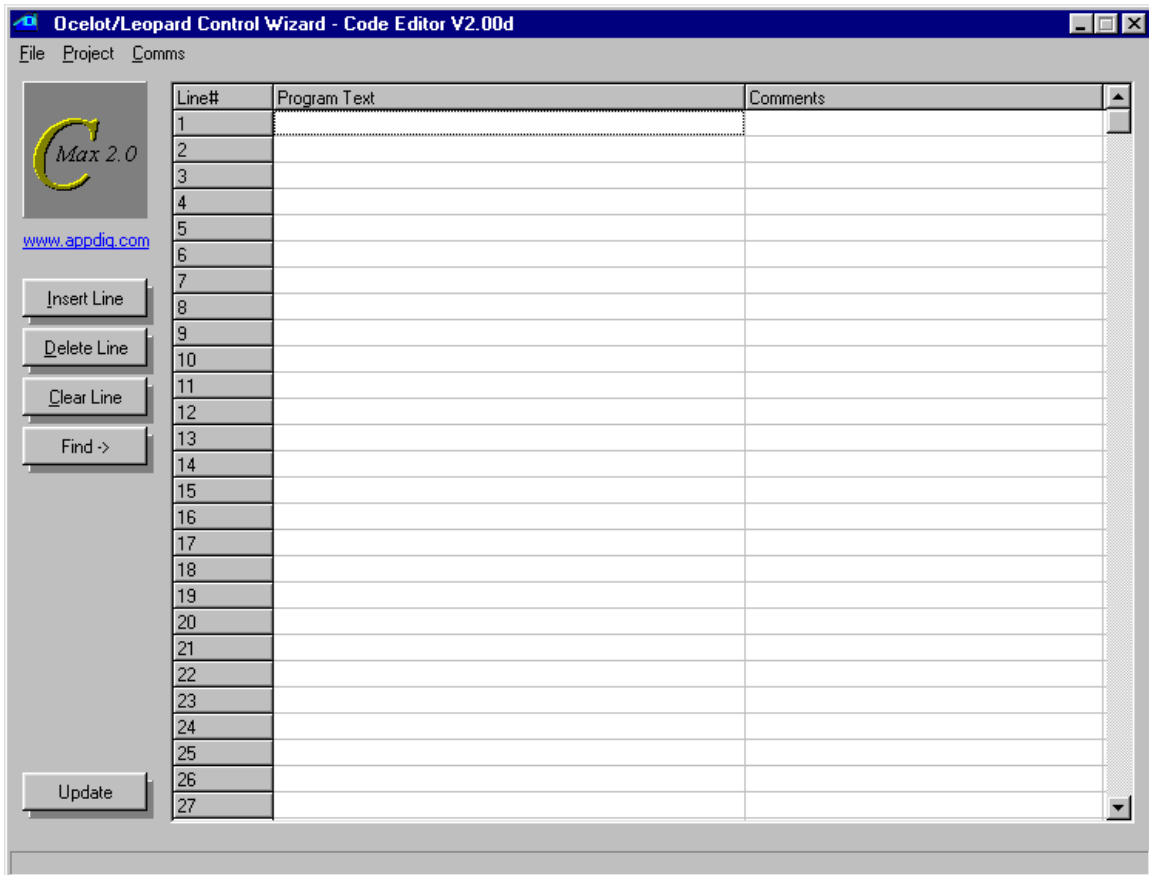


Fig 5

Click on Comms → Comms Setup and a screen similar to the following will appear (fig 6):

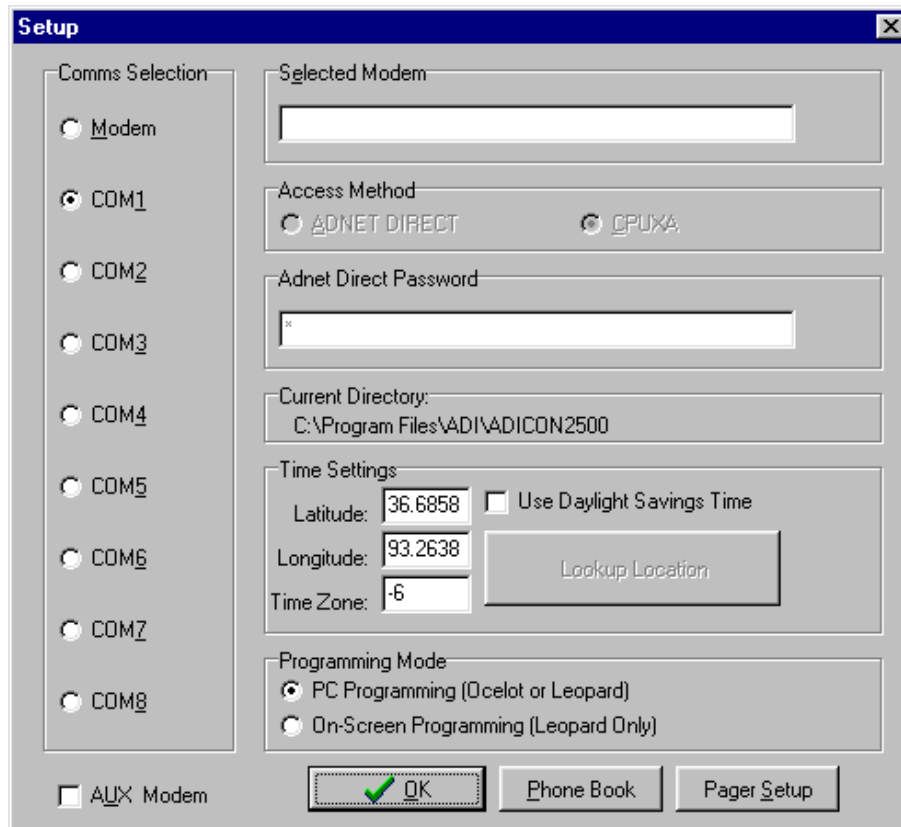


Fig. 6

In the “Comms Selection” area, choose your computer’s serial port to which you connected the Leopard II’s serial interface.

Make sure the “Programming Mode” area has the “PC Programming (Ocelot or Leopard)” radio button selected.

Finally, look at the “Time Settings” area and enter your latitude and longitude, along with your time zone (in relation to GMT) and check the daylight savings time box if applicable. To know your required time zone setting, you can open your Windows time setting utility by double clicking on the time in your system tray and then clicking on the “Time Zone” tab. You will see a window similar to the following (Fig. 7):

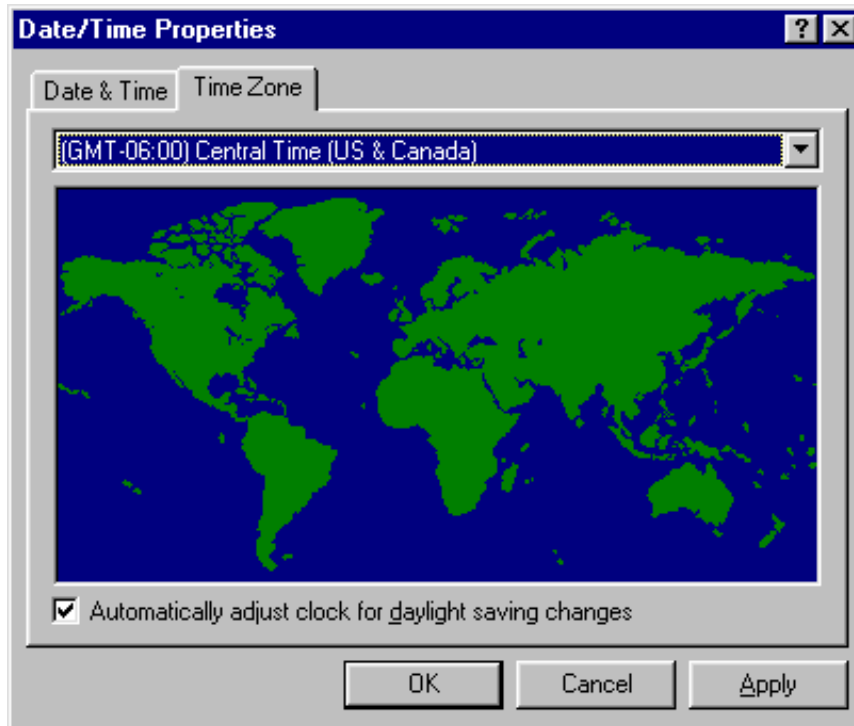


Fig. 7

In this example you can see the “GMT -06:00” that corresponds to the Central Time zone, and the “-6” that was correspondingly entered in Fig. 6

These settings allow the Leopard II to automatically calculate the sunrise and sunset times for each day of the year. Once finished, click on “OK” to close the setup window. You can now verify your settings and connections by clicking on Comms → Attach to Controller. You should see a screen similar to the one in fig. 8:

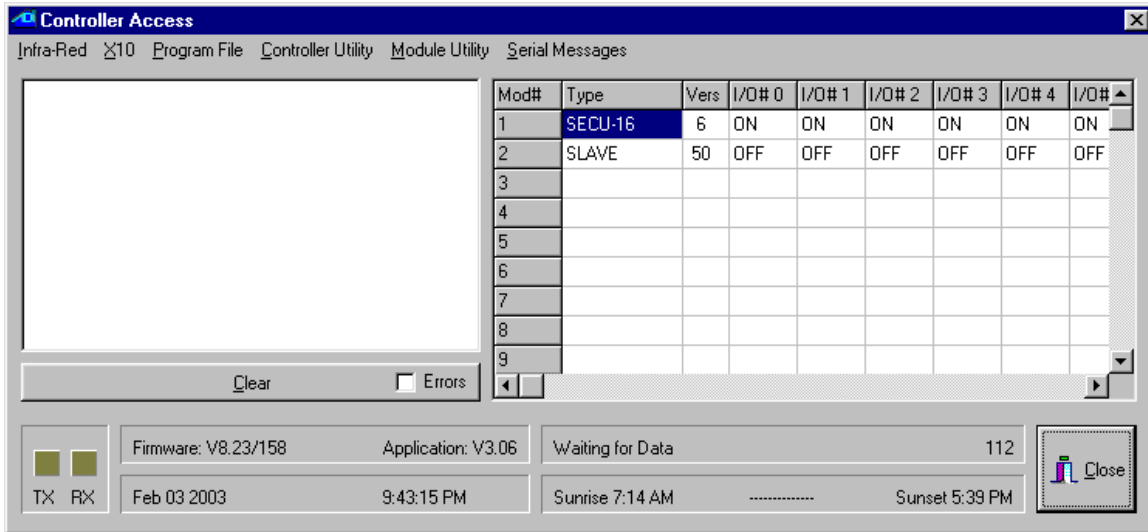


Fig. 8

This is the Controller Access screen. All the pull down menus listed at the top give you access to various utilities that will be covered elsewhere in this manual. The important thing to look for right now is the TX and RX squares in the lower left corner of the screen. They should both be blinking alternately as C-Max obtains the ongoing status of the Leopard II. You should also look at the “Firmware” and “Application” fields to see that a value has been read from the Leopard II. The actual values may differ from the ones shown in fig. 8 since ADI regularly updates the firmware and software of its controllers. When you access this screen for the first time, you may get a message box on your screen stating that your controller does not have the latest executive version and ask you if you want to load it now. This might happen if the version of C-Max on your computer is newer than the version that the Leopard II was first initialized with. You should agree (click on Yes) to the reloading of the new executive. You can find out more about reloading the executive and it’s exact purpose in the manual section dealing with the C-Max utilities.

If you see the TX and RX squares blinking regularly then your C-Max installation and configuration is complete and you may now begin using it. If the squares do not blink and you get a message box with a communications error then you will need to verify that the correct serial port is selected and/or verify the actual cabling between your PC and the Leopard II.

If C-Max programming is totally new to you and you would like to have a feel for how to proceed with creating a working application, or you would just like to have the immediate satisfaction of seeing it “do something” then the next chapter will be of interest to you. It literally holds your hand through the process of creating a simple working program. If you are already experienced with versions prior to C-Max 2.0 then a good place to start is by reading the section on **Projects** since this is the main addition to C-Max as of version 2.0.

1.6 Writing your First Program

This chapter has been inserted as a transition between the installation guide and the programming guide sections of the manual to give first time users the opportunity to familiarize themselves with the process of creating a working Leopard II application by showing the step by step creation of a project. The goal is not to provide a how-to guide on programming logic itself but simply to illustrate the typical steps involved in creating a project from start to finish. This is analogous to the first exercise often given to students learning a new computer language where the goal is to simply create a program that prints or displays “Hello World”.

1.6.1 Creating a Project

Since this is a Home Automation controller, saying “Hello” to the world will consist in creating a screen with a text label and two buttons, used to turn an X-10 controlled light On and Off. We will begin by starting up the C-Max program and looking at the main screen (see fig. 5). This is the program editor window and is the main startup screen. The first step is to create a project, so click on the **Project** menu at the top left and on the pull down menu, select **New Project**. You will see a file browsing window like the one in fig. 9

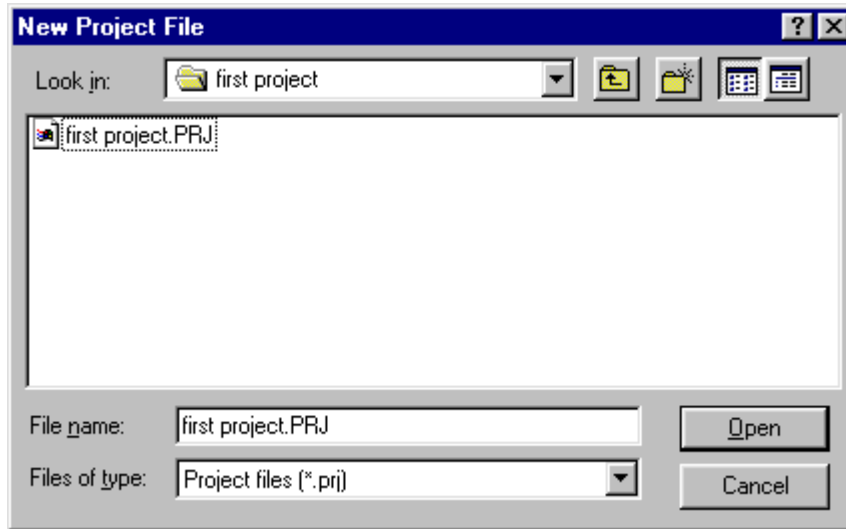


Fig. 9

It is a good idea to keep each project in its own directory, so click on the new folder icon (the small file folder with a sparkle on it) and create a folder for your project. Then go into that folder and enter the file name you want to use for this project. In the example shown in fig. 9, both the folder and the project have been named “first project”. Clicking on **Open** will create the project and the window will close.

Now, anytime you want to save your work in progress, just click on **Project** on the main window and then click on **Save Project** to save all the files related to the project currently opened.

1.6.2 The System Map

C-Max is the editor for several ADI controller models and expansion modules so it is necessary to tell it what our particular configuration will be. That way, the program editor will offer the right options when several possibilities exist. This project only assumes that we are using the Leopard II by itself so this is the only item that really needs to be configured in the System Map. Click on **Project** and then on **System Map**. You will see a window similar to fig. 10:

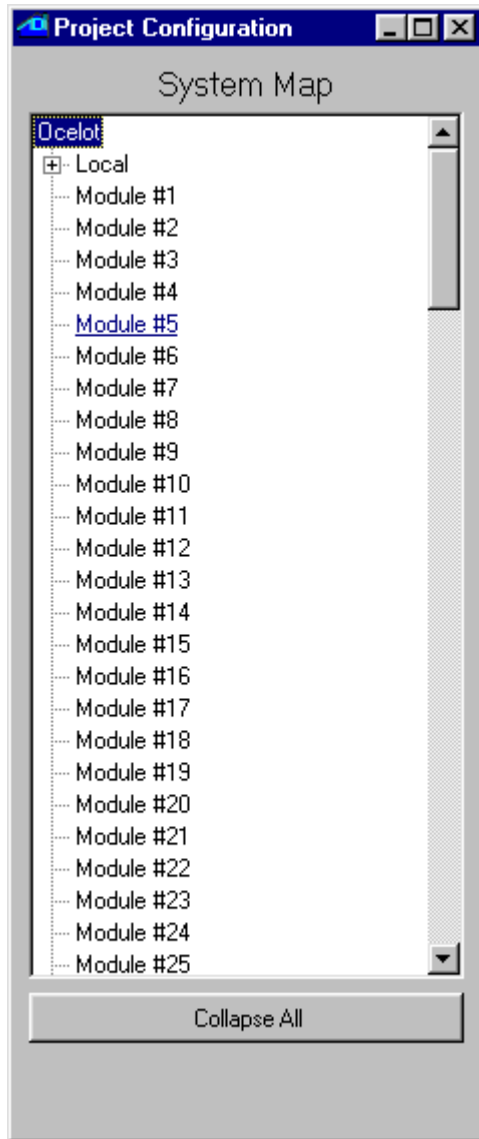


Fig. 10

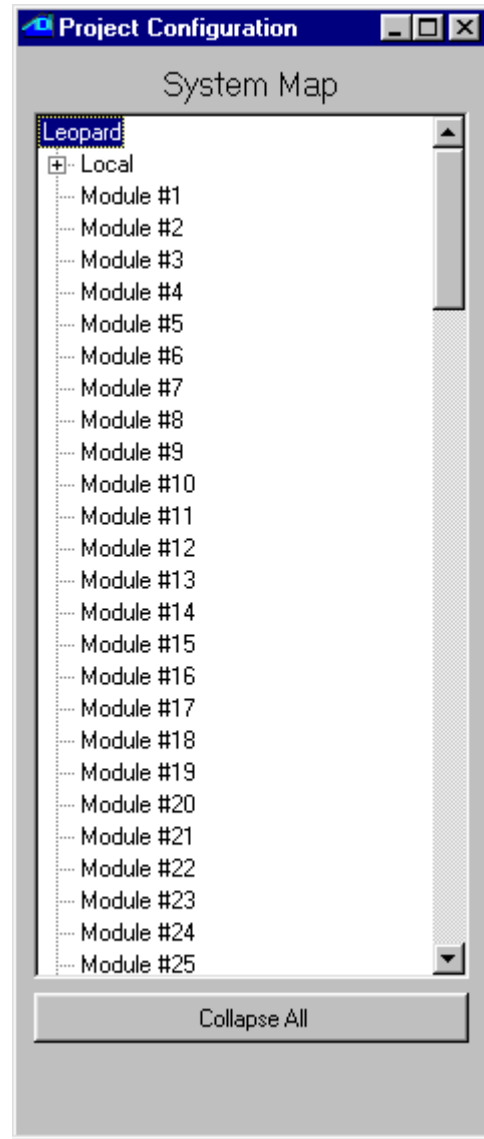


Fig. 11

This window shows a list of the modules and the small “+” signs show that these items can be open in tree-like fashion to define other items. The first item at the top defines the master controller and shows **Ocelot** by default. The Ocelot is another ADI controller similar to the Leopard II but without a touch screen. Since we want to be able to use the touch screen functionality of the Leopard II, right click on **Ocelot** and choose **Leopard** in the small list of choices shown. Your screen should now look like fig. 11. You may now close the System Map window by clicking on the “x” in the upper right hand corner.

1.6.3 The Touch screen Layout

Now that the editor “knows” our configuration, we may begin to create the actual application. We could begin with either the program text or the screen layout. Since we need to know the button’s screen object number (which is assigned as the button is created) when referring to it in the program, it will be easier to start with the screen and take note of the object numbers for our program. From the main screen, click on **Project** and then **Touch Screen Setup**. You will see a window like the one in fig. 12:

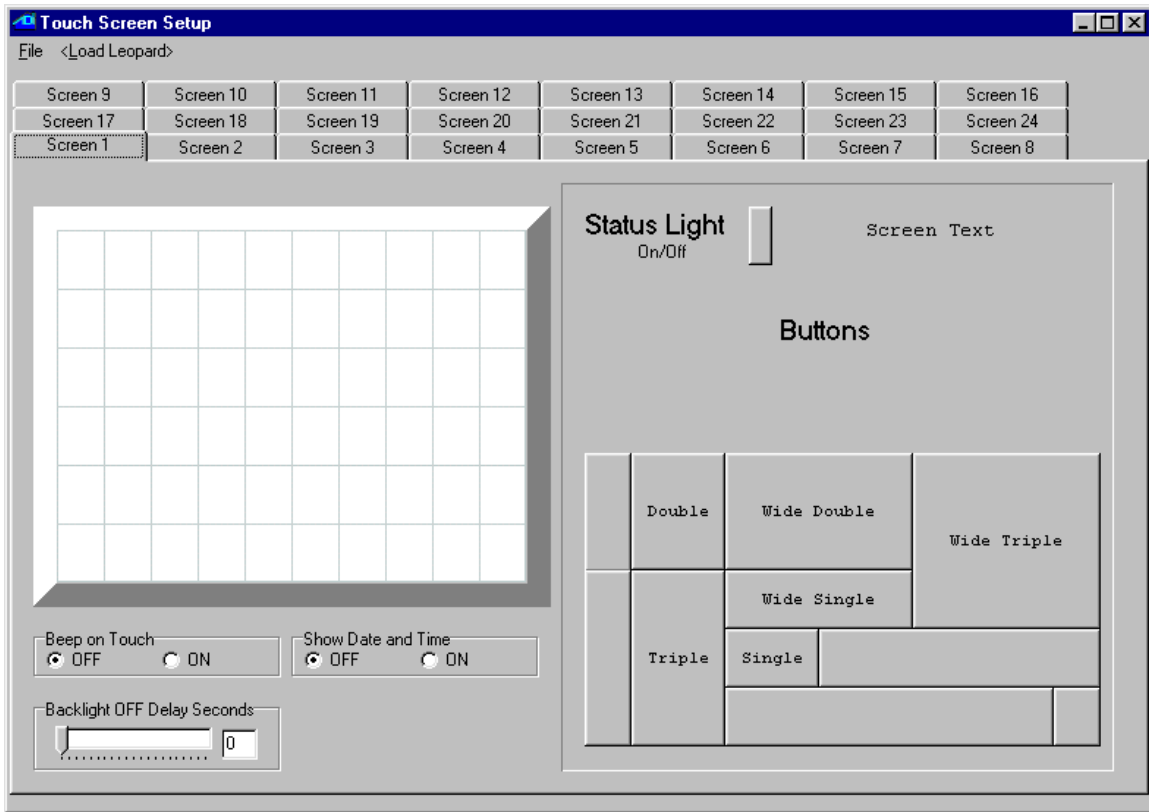


Fig. 12

Let's start by putting a text heading on the screen. Click on **Screen Text** on the right hand side (there is no frame around this choice). You will see a small text window in the upper left corner of the touch screen area. Right click on it and choose **Enter Text**. An editing window will appear on the screen (fig. 13)

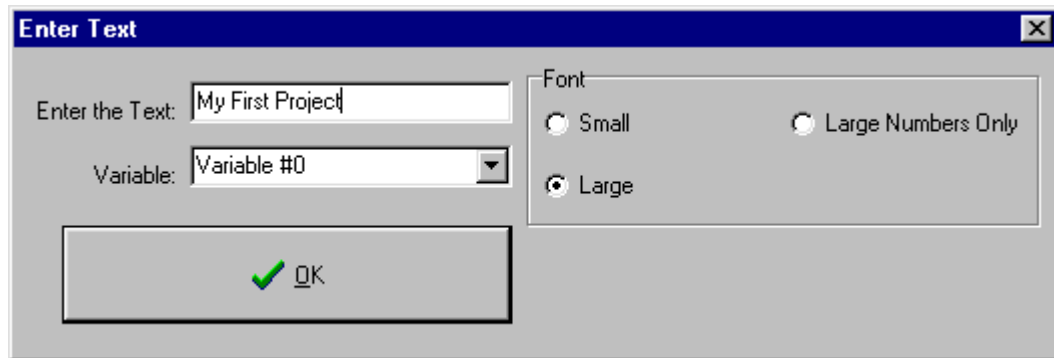


Fig. 13

Enter the text you wish to see as a window heading. In our example, “My First Project” has been entered. The **Large** font radio button has also been selected. For this project, you can ignore the **Variable** list box. When you click on **OK** you will see your text object with the new text and large size in the upper left of the screen area. Click and drag it towards the upper center of the screen. Note that moving the mouse pointer over it will show that it is object #1.

Now we want to add the two buttons to control our X-10 light. In the button area, click on the one identified as “single”. It will appear in the upper left of the touch screen. Again, right click on the object and choose **Enter Caption**. You will see an editing box (fig 14).

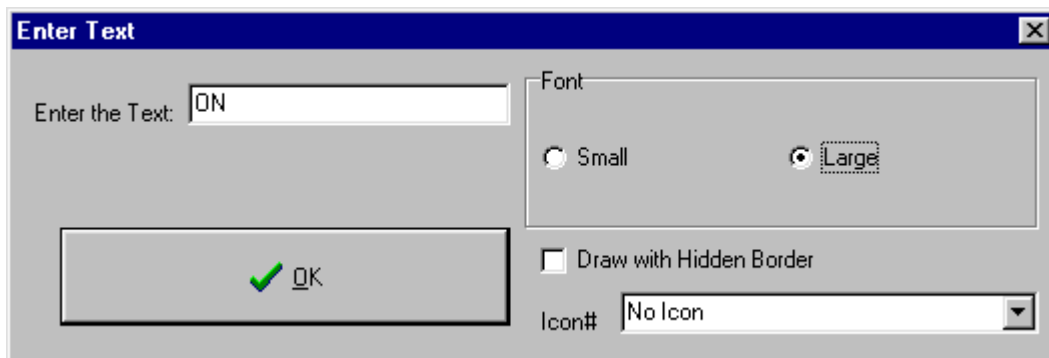


Fig. 14

This will be the ON button so enter “ON” as the caption and choose the **Large** font. Ignore the other options and click on **OK**. The button will now show it’s caption. Drag the button under the previously created heading. At this point your screen should look something like fig 15:

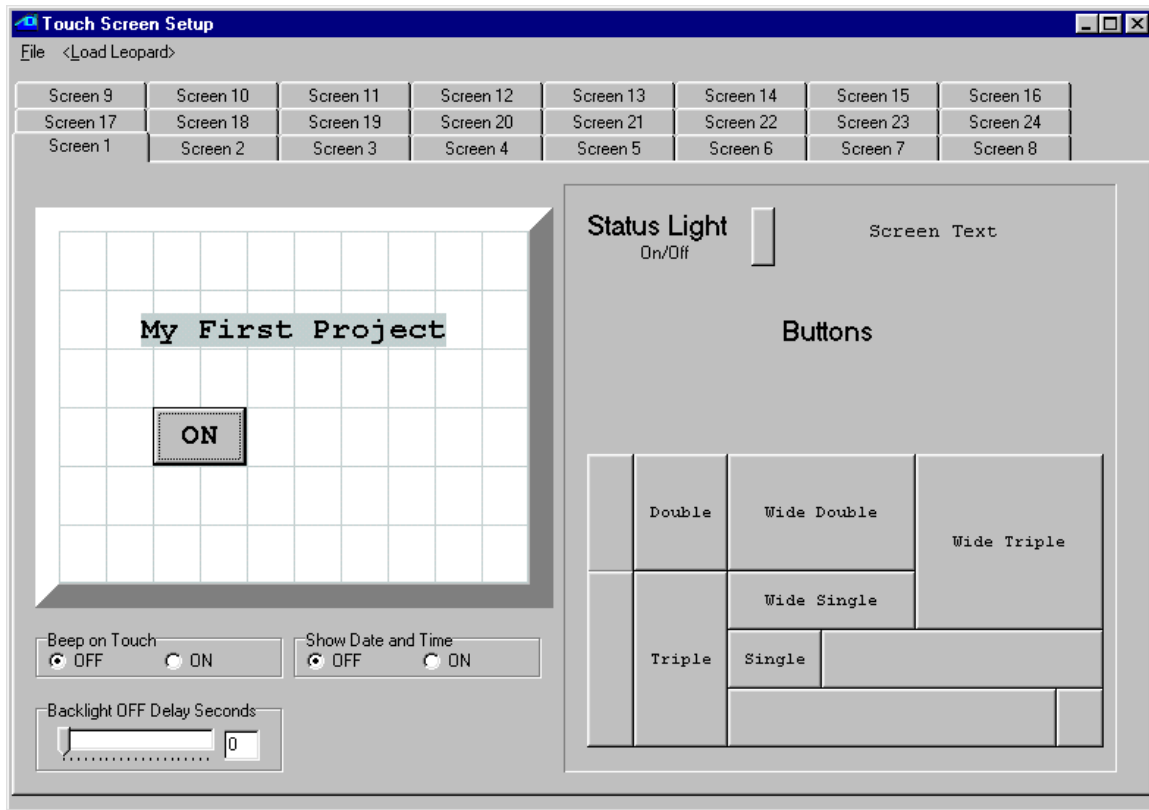


Fig. 15

Float your mouse cursor over the button you have just created and note that it is object #2. Create the OFF button by repeating the procedure just explained and give it an “OFF” caption. It’s object number will be 2. Under the screen area, notice the four radio buttons and the slider/edit box. You may select **Beep on Touch** to hear a low volume beep every time the screen is touched (since there is no tactile feedback). **Show Date and Time** will display the date and time in the upper right corner of the screen if selected. Finally, use the slider or edit box to enter a time delay (in seconds) after which the screen backlighting will turn off unless it is touched again.

Our screen definition is now finished and should look like fig. 16:

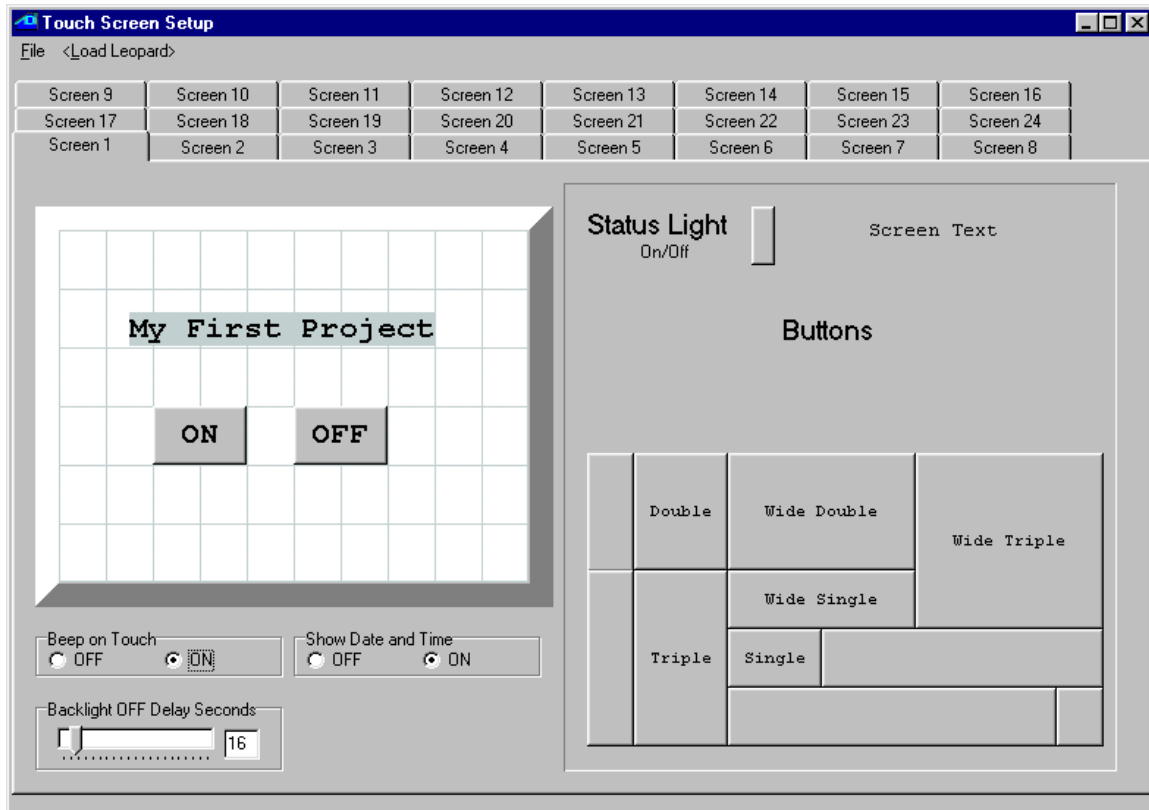


Fig. 16

Since this screen definition is part of a project, there is no need to save it separately, so just close the window by clicking on the “x” in the upper right. Click on **OK** when you see the confirmation window that warns you about closing the screen editor window without first saving. You will be brought back to the main window. This would be a good time to save our work in progress so click on **Project** then **Save Project**.

1.6.4 Writing the Program Code

Now we reach the point where we write the actual program code for our project. Under the **Program Text** heading, double click on line #1. You will get the control wizard screen as shown in fig. 17. Select the **IF** instruction type and then **Touch Button Pressed** instruction (use the vertical scroll bar to find this instruction.). The window will update its available controls to show the ones that apply to a Touch Button. Leave the **Module** choice as being **Leopard** since this is our only module. Since we noted that the ON button was object #2 in the touchscreen editor, select **Touch Object #2** in the **Button** list box. Once all these selections have been made click on the large **Enter Touch Button** button at the bottom of the control wizard and you will see your newly created program line in the main window.

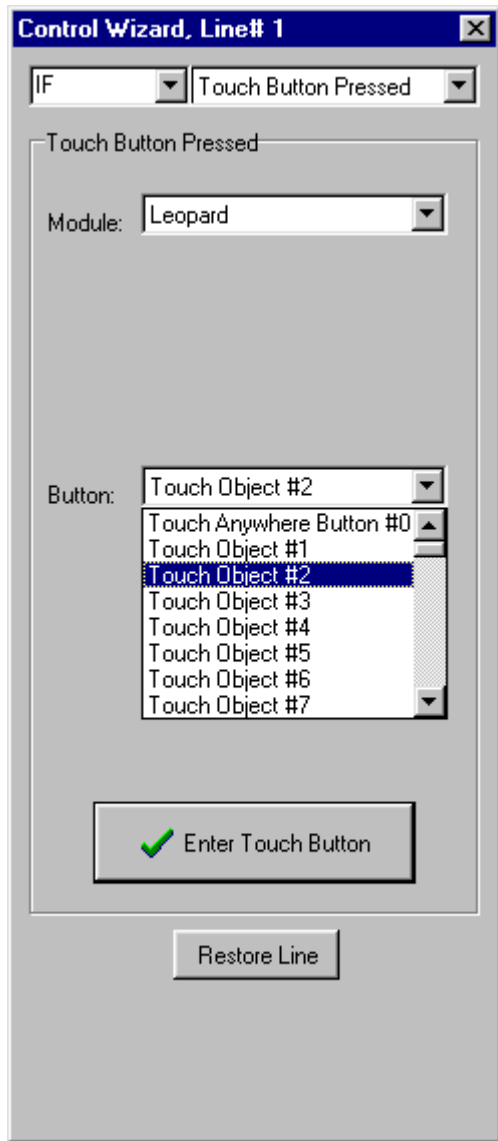


Fig. 17

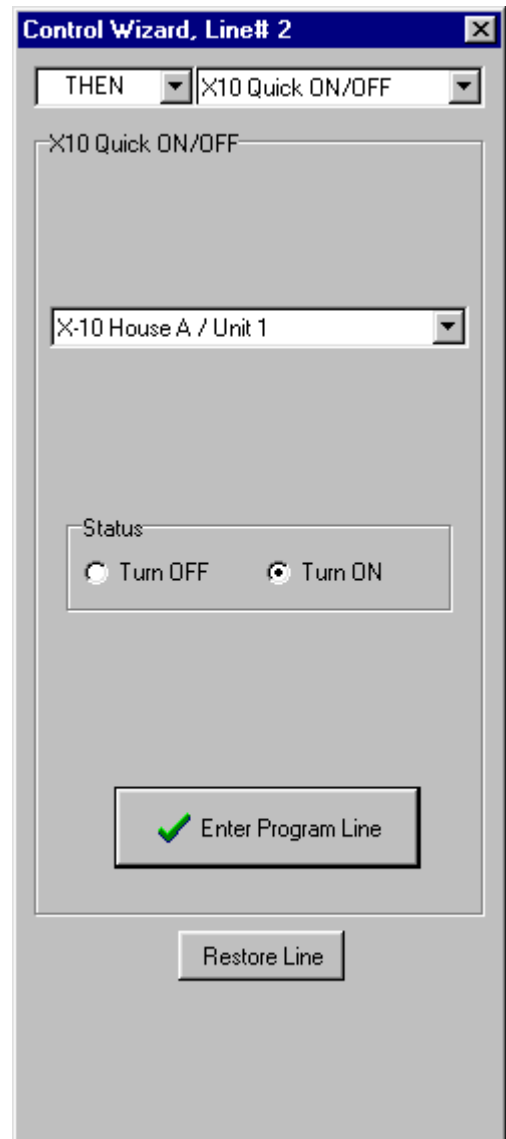


Fig. 18

Now double click on line 2 and create the corresponding action statement. Once again you will get the control wizard screen as shown in fig. 18. Select the **THEN** instruction type and then the **X10 Quick ON/OFF** instruction (use the vertical scroll bar to find this instruction.). The window will update its available controls to show the ones that apply to an X10 Quick ON/OFF instruction. Choose an X10 house code/ unit code combination that corresponds to a light, lamp or whatever device that you want to try your first program with. Then click on the **Turn ON** radio button. Once these selections have been made click on the large **Enter Program Line** button at the bottom of the control wizard and you will again see your newly created program line in the main window.

Using the same technique learned with the first two lines, now create the next two lines that will allow us to turn the device OFF. Just double click on the line you want to create or edit and make the appropriate choices. The difference between the first pair of lines and the second pair is that you will now be looking for **button #3** being pressed and the resulting action will be to trigger an X10 **Turn OFF** command. Make sure you select the same X10 house and unit codes as in the first two lines!

Now we need to do some screen control to make our simple program work correctly. Remember when you were doing the touch screen layout? The tab at the top showed that you were editing screen #1 (see fig. 16). You can have up to 24 screens each containing any combination of buttons, text objects or status lights (see the reference section of the manual to learn how to use these). Displaying one screen or another is done under program control and can be in any way you like, like pressing a button, receiving an X10 command, by time of day, etc. Since our simple program only has one screen, we want it to be displayed at all times. When the Leopard II is first powered up or a new program is loaded into it, it displays a blank screen and the special variable that specifies the desired screen # is set to 255. We thus want a program segment that looks for the screen variable **not** being equal to 1 and if so, set it to 1. Edit the next two lines to test and act upon variable #63 (called the screen variable) to do just that. Your control wizard windows should look like figures 19 and 20 for the **IF** and **THEN** statements.

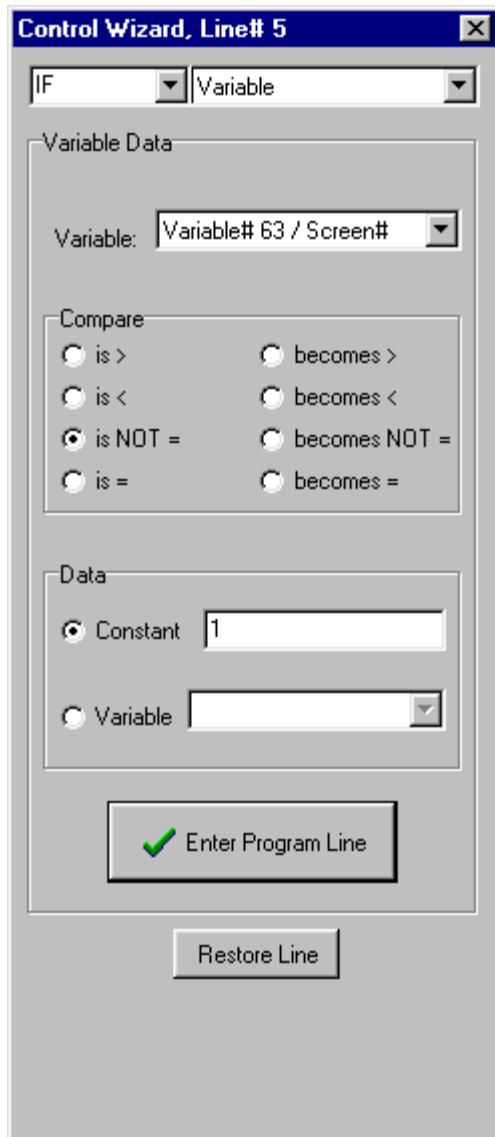


Fig. 19

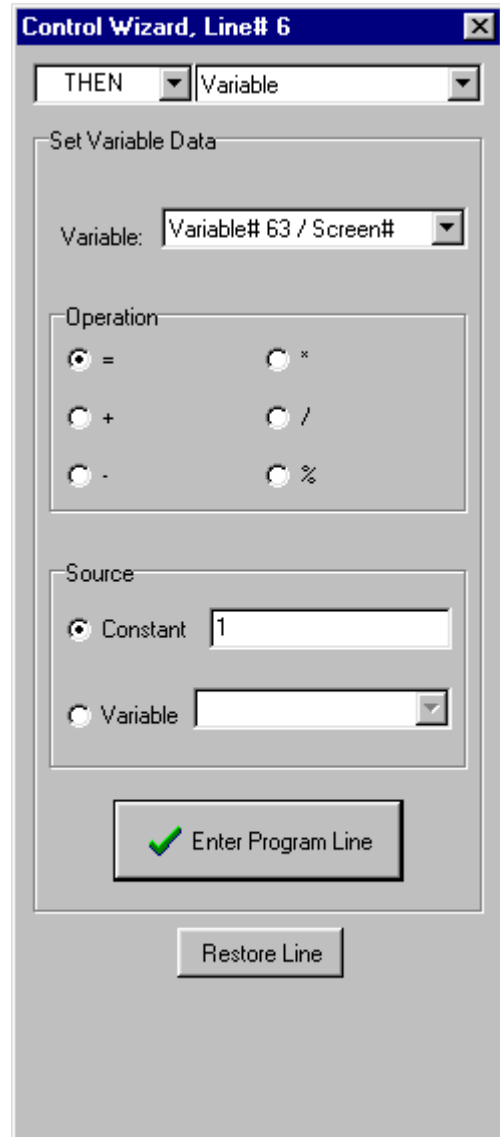


Fig. 20

We're almost done! We have the test and action statements to turn the device ON, another test and action statement pair to turn it OFF, and a program segment to make sure we're displaying screen #1. All we need now is an **END** statement. Double click on the next line and in the control wizard, select the **END** command type. There are no other options to this command so you can now close the command wizard by clicking on the "x" in the upper right hand corner. You will see an **End Program** line on the main screen. Your program is now complete and should look a lot like the one shown in fig. 21 with the possible exception that the house and unit codes for the X10 device will be the ones you chose. Note that you can also add comments for each line as shown in fig. 21. To do this, just click the line you want to comment on under the **Comments** heading and enter your comments. Now, this would be a good time to save your project, so click on **Project** then **Save Project**.

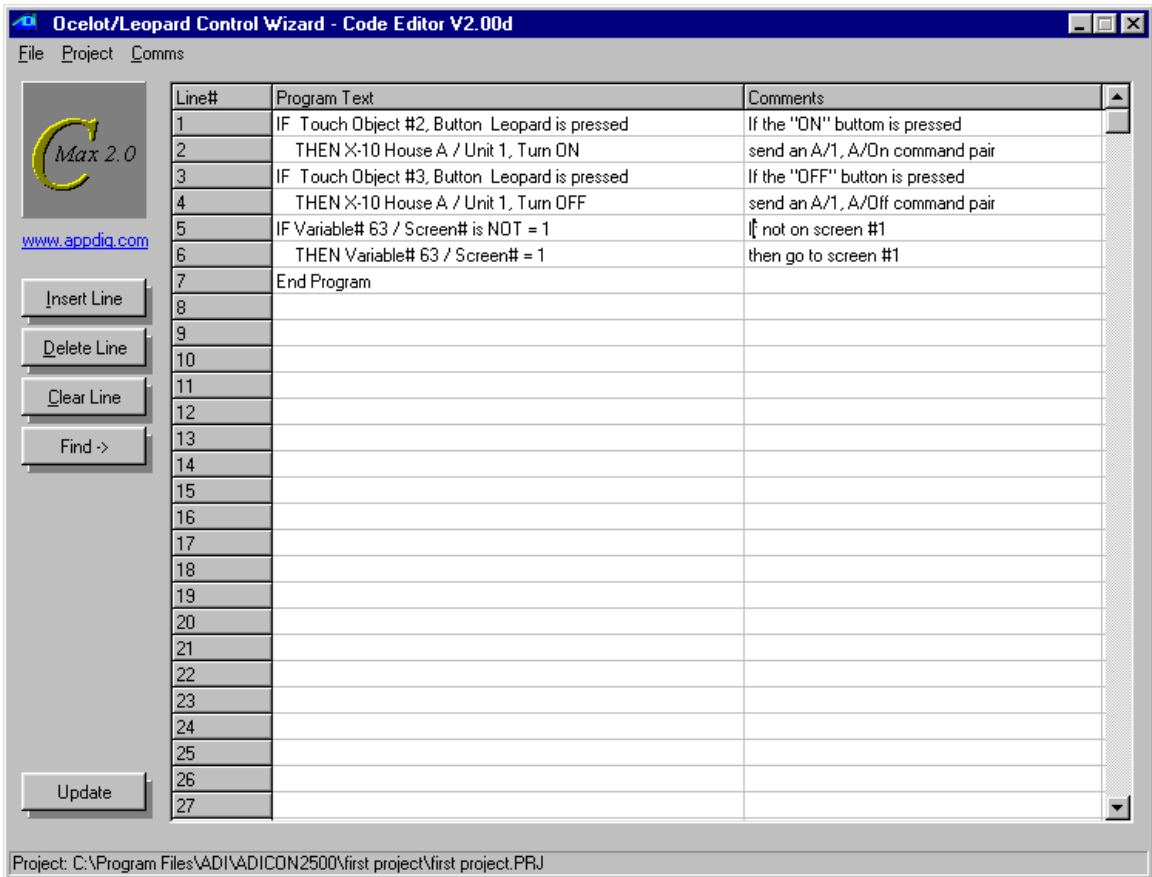


Fig. 21

The only thing left to do at this point is to get the screen definition and program code into the Leopard II and actually try it. To do this, click on **Project** then **Download Project**. You will see a project downloading window like the one in fig. 22. Using the check boxes, select the **Download Program** and **Download Touch Screen** options. This project does not use icons so there is no need to select the icons download. There will be no harm done if you download icons but the download will take more time if you do so.

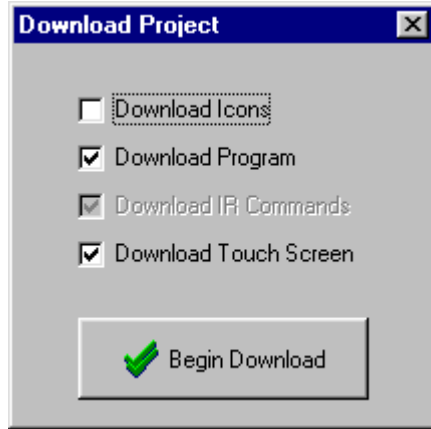


Fig. 22

Click on **Begin Download**. The controller access window will appear and show the progress of the operation (see fig. 23). Watch the status window messages and the progress bar to see the touch screen objects and then the program get downloaded (you will see the progress bar for each of these operations). When complete, a small window will appear (fig. 24) indicating that the controller is being restarted.

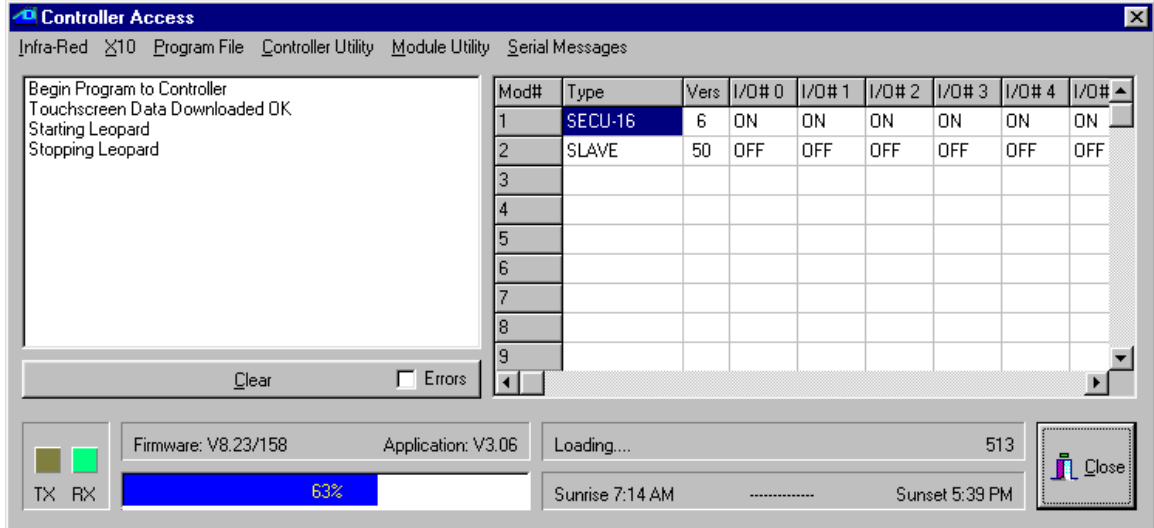


Fig. 23

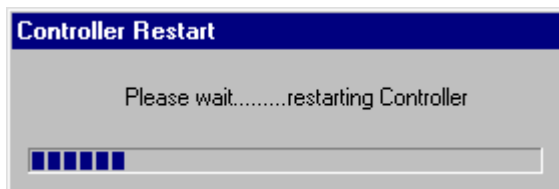


Fig. 24

Watch your Leopard II screen now. You will see it “reboot” and after a delay lasting a few seconds you should see the screen layout you created! Try pressing the ON and OFF buttons to control your X10 device. Congratulations, you have completed your first Leopard II project. Of course, your actual application will be much more complex but the basic steps involved in creating it will be very similar to the ones just covered.

The next manual section deals with the programming model used for the program logic. It is strongly suggested that you read it carefully and make sure you understand the general principles used to create an application.

2 Programming the Leopard II

2.1 The C-Max programming model

The C-Max utility that comes with the Leopard II serves as the program editor for the controller. The editor is the very first screen that you see when C-Max is started. Before we explain the available instruction set and start to create programs, we will look at the programming model that the Leopard II and other ADI controllers use to accomplish their tasks. Understanding this aspect of the controller is key to obtaining satisfactory results and realizing the full potential that the Leopard II has to offer. The average PC user familiar with languages like BASIC or Visual BASIC may find this model unfamiliar so the following paragraphs will explain the history behind the model used and the reasons for its adoption for the Leopard II.

2.2 Ladder Logic

The programming model used for the Leopard II is called “Ladder Logic”. This language *model* (as opposed to a language as such) has its roots in the industrial control field. The early control logic for industrial process control consisted mainly of switches, pushbuttons, relays, etc. and this is still used in some of the simpler applications. Many relay contacts and timers were used to accomplish the necessary sequencing and timing for the entire process. As machines grew in complexity, the number of relays and other mechanical components needed to do everything became rather large and the mechanical nature of these controls needed a lot of maintenance. Logic diagrams for these circuits often consisted of two long lines representing the power source (often “common” and 24 volts) for the controls and then each relay or button was illustrated as connecting across the two power lines. These diagrams thus resembled ladders and the schematic diagrams were called ladder logic diagrams. When computer technology advanced to the point of becoming fast and affordable enough to replace the logic part of the circuit with a computer program, the Programmable Logic Controller (PLC) was born.

In the mechanical circuits, an action like pressing a button usually triggers a series of events happening in quick succession, like closing a contact, which then activates a relay coil. The relay contacts need a few hundredths of a second to close before the next action can take place but generally, the action seems almost instantaneous in relation to the task that needs to be performed such as stopping a conveyor belt. When this functionality was transposed to a PLC, the computer needed a way to look for any one of several possible events and then respond in the appropriate way. Since a computer can only do one thing at a time, the solution is to have it scan all the possible sources for triggering events continuously (polling) and then have it perform the needed actions as quickly as possible. This is so that it could get back to looking out for any other possible triggering events. As long as the computer can react to a triggering event fast enough so that its always ready for the next event, it appears to be doing many things at once.

The Leopard II is in fact a PLC specifically designed with home automation in mind. The Ladder Logic programming model is particularly well suited to such an application because it allows the controller to acquire and maintain general status information about certain aspects of the home like temperatures, HVAC status, alarm system armed/disarmed, etc.. The program can then make logical decisions based on any or all of these facts. For example, you could have a short program segment that just sets a variable to indicate that the alarm system is armed. This variable can then be used anywhere else in the code where knowing the alarm system is part of the decision process, such as using motion sensors to turn lights on automatically if the home is occupied, but to trigger an alarm if the alarm system is armed. This allows your motion sensors to do two totally separate yet useful functions by virtue of program logic. The various expansion modules available gives the controller the ability to acquire information based on many types of inputs such as contact closures, analog voltages, environmental data, etc.

2.3 Program Flow Rules

To successfully create your home automation application it is important that you understand the rules that govern program flow. The following paragraphs will explain, by way of examples, how the logic tests function and how you can combine them to create complex logic decisions based on a few well-established rules.

In ladder logic systems, the entire program runs as one long loop, *never* stopping, waiting, or looping at one spot. In fact, pure ladder logic is nothing but IF/THEN statements looking for conditions and then updating

flags or actually activating outputs. Here is a sample C-Max program to illustrate this. The program looks for an A/1 ON command, and sends a B/1 On command 3 seconds later (note: The program listings shown were produced using the “print to file” utility in C-Max. This is explained in the utilities section of the manual):

Example #1

STATEMENT	COMMENT
0001 IF X10 House A/ Unit 1, from ON Command Pair	//If X-10 A-1 ON received
0002 THEN Timer #0 = 1	//start timer 0 (set it to 1)
0003 IF Timer #0 becomes > 3	//the first time timer 0 reaches 4
0004 THEN Timer #0 = 0	//stop the timer
0005 THEN X10 House B/ Unit 1, Turn ON	//and send an X-10 B-1 ON command
0006 END	

Look at lines 1 and 2. The A-1 ON command simply starts a timer (setting a timer to a non zero value causes to increment from that value upwards once per second). The code segment from lines 3 to 5 looks at the timer on-the-fly as the program loops to see if it is time for the next step, and will send the B/1 On command when it’s condition (the timer having become greater then 3) tests true. Don’t forget that the program loops continuously, so line 3 will test false several times before finally testing true. Each loop through a program is called a *pass* so we can say that several passes will be executed until line 3 tests true. Also note that the timer will increment on it’s own once per second and that no program lines are needed to accomplish that. When line 3 finally tests true, the X-10 output command will be put in a transmit buffer, freeing the program right away to keep looping. Thus functions like sending X-10 commands, incrementing timers, etc. are done “behind the scenes” to allow the looping to never stop. You can see that the program is actually multitasking when it becomes a long loop with many segments, each one looking for it’s condition to being true during any given pass.

An example of multitasking is shown in Example# 2. This program checks if an X-10 A-1 ON command is received 3 times within 10 seconds of receiving the first one, and if so sends a B-1 ON. If you simply look at the program now, you may find it difficult to grasp the entire logic; it just appears as somewhat related logic tests. If we break down the desired task into logic rules, then it becomes much more “natural” looking. Here are the rules:

- 1- If we receive an A-1 ON and we had not yet started to time the 10 second interval, then count the command as being the first one and start timing.
- 2- If we receive an A-1 ON and we were already timing the 10 second interval, then just count the command as one more.
- 3- If we have reached a count of three A-1 ON commands, then stop timing and send the B-1 ON command since the target count was attained.
- 4- If the timer shows that 10 seconds have elapsed since the first command, then the time has effectively run out and we stop timing and reset the count to zero.

Now look at the program code:

Example #2

```
0001 IF X10 House A/ Unit 1, from ON Command Pair //If X-10 A-1 ON received
0002 AND Timer# 0 = 0 //but timer was not yet started
0003 THEN Variable# 0 = 1 //then start with count of 1
0004 THEN Timer# 0 = 1 //and start timer
0005 IF X10 House A/ Unit 1, from ON Command Pair //If X-10 A-1 ON received
0006 AND Timer# 0 > 0 //and timer is already started
0007 THEN Variable# 0 + 1 //then increment count
0008 IF Variable# 0 > 2 //if count has reached 3
0009 THEN Variable# 0 = 0 //then reset count
0010 THEN Timer# 0 = 0 //and stop timer
0011 THEN X10 House B/ Unit 1, Turn ON //and send X-10 B-1 ON command
0012 IF Timer# 0 becomes > 10 //if 10 seconds have elapsed
0013 THEN Timer# 0 = 0 //then stop timer
0014 THEN Variable# 0 = 0 //and reset count
0015 End Program
```

Notice how each program segment (test commands followed by action commands) corresponds to one rule. By breaking the problem down to a set of rules and conditions, coding comes more naturally. In essence, the program continuously evaluates the conditions against the rules and immediately applies any required action or updates.

2.4 Precedence of Logic Tests

Example #2 showed how an AND statement can be used for conditional logic. Both AND and OR statements provide the means for making simple or complex logic decisions. It is important to understand the way that these statements work when multiple conditions are tested. Not knowing the *ordering* rules can cause unexpected program behavior. The important thing to remember is that statements are evaluated as true or false on a *line by line* basis. If you are experienced with conditional logic you may have noticed that there are no parenthesis to force evaluation precedence. This makes the order of the logic tests very important. The three commands that perform logic tests are: IF, AND, and OR. The rules that govern each of these commands are:

IF – An IF statement assumes that any preceding conditions are false, so in fact it starts an entirely new logic test sequence. Its end result will be true only if the condition it tests is true.

AND – The end result of an AND statement will be true if the condition it tests is true **and** the previous logic test just before it also had an end result of true.

OR – The end result of an OR statement will be true if the condition it tests is true **or** the previous logic test just before it also had an end result of true.

A previous true or false end result is always carried forward to the next program line until it gets cleared by an IF statement.

The best way to illustrate this is again with examples. Here are two versions of a sample program (examples #3a and #3b) that is meant to turn on B/1 if either an A/1 on is received or button #1 is pressed on the Leopard screen. In both cases we only want the command to be executed if the time of day is past sunset:

Example #3a

```
0001 IF Time of Day > Sunset offset 0 minutes
0002 AND X10 House A/ Unit 1, from ON Command Pair
0003 OR Touch Object #1, Button Leopard is pressed
0004 THEN X-10 House B/ Unit 1, Turn ON
```

Example #3b

```
0001 IF X10 House A/ Unit 1, from ON Command Pair
0002 OR Touch Object #1, Button Leopard is pressed
0003 AND Time of Day > Sunset offset 0 minutes
0004 THEN X-10 House B/ Unit 1, Turn ON
```

Example #3a shows an example of a potential programming pitfall (known as the “OR trap”...). If your intent was to allow the A/I ON command or the touch button to work only if past sunset, then you’ll have the unexpected behavior that the touch button will work at any time; regardless of sunset. This is because any OR statement effectively starts a new evaluation sequence, and the THEN statement(s) will be executed as long as the evaluation sequence still tests true by the time THEN is reached. The proper way to code this is shown in example 3b. The important thing to remember is: If you have one of several possible OR conditions plus some mandatory AND conditions, *put the OR conditions FIRST!*

The “line skipping” instruction in C-Max provides another, more flexible method of altering program flow and effectively allows the nesting of conditional statements. Here is example #3 rewritten to use this feature (#3c). Notice how the editor line numbers are by the “skip to” instruction. Also note that you do not need to use THEN statements to have an ELSE, nor do they need to be in a specific order. In the example below, a single ELSE specifies that we want to skip over the next 3 lines if the time of day test is false.

Example #3c

```
0001 IF Time of Day > Sunset offset 0 minutes
0002 ELSE Skip to line 6
0003 IF X-10 A/I On Command Pair
0004 OR Touch Object #1, Button Leopard is pressed
0005 THEN X-10 House B/ Unit 1, Turn ON
0006 End Program
```

This makes for a more logical flow (the overriding “past sunset” condition is evaluated first) and can save a considerable quantity of program lines if a repetitive condition must be tested. For example, you may have a dozen or more different programmed activities that simulate an occupied house, and that depend on the alarm system being armed to be applicable. Rather than having a dozen AND statements to check the alarm status along with each programmed time, you can test for the armed condition first and skip all the timed commands if the alarm-armed status test is false.

2.5 Single and Continuous Events

As explained at the beginning this tutorial, the ladder logic nature of a C-Max program means that all the program lines are repeatedly evaluated several times per second in a continuous loop. This means that a way is needed to detect certain types of events only the first time they happen to avoid having multiple occurrences of the resulting action. For example, you would not want the controller to send out five copies of an X-10 On command to a light just because you held your finger on the Leopard button for a full second. In most commands involving the manipulation of data like timers, variables and constants, the word “becomes” is used to designate this type of behavior. For example, the command “IF Time of Day becomes = 11:30” will test true only the first time the time is 11:30, even though that time value technically lasts a full minute. Other instruction types involving I/O like X-10 and expansion module inputs will offer “Is ON” and “Turns ON” type choices. Here, the “Turns” option will test true only the first time that the state changes to on as opposed to already being On according to the internal status table that the controller maintains. In the case of X-10 status change there is a third option; looking for the command pair itself, allowing you to detect individual occurrences of the command even if the status was already in that state beforehand. Some events like Touch Button presses and received IR commands are by their nature single events and do not present a choice. For a “becomes” type of test to test true more than once, it has to have

become false first. For example, for a “IF Variable #1 becomes = 2” command to test true a second time, the value will have to change from being equal to 2 to becoming anything else but 2, and then 2 again.

The ability to test for continuous or single events creates useful possibilities and also some potential pitfalls. Consider the case where you would want to turn off your water heater with an X-10 command if the time is past 9:00 PM and the water is already hot enough (detected by a thermostat connected to a SECU16 expansion module input that turns off when water is hot enough). Here are 4 possible ways you could code this test in C-Max (Examples #4-a through #4-d). The X-10 output command has been omitted for brevity:

Examples# 4-a to 4-d

```
0001 IF Time of Day becomes = 21:00 // Ex. 4-a
0002 AND Module #1 -SECU16 Input-#0 Is OFF

0001 IF Time of Day is > 20:59 // Ex. 4-b
0002 AND Module #1 -SECU16 Input-#0 Turns OFF

0001 IF Time of Day becomes = 21:00 // Ex. 4-c
0002 AND Module #1 -SECU16 Input-#0 Turns OFF

0001 IF Time of Day is > 20:59 // Ex. 4-d
0002 AND Module #1 -SECU16 Input-#0 Is OFF
```

In example #4-a, the exact time is the single event and the input has to already be OFF for the program segment to test true. Example #4-b reverses things so that as long as the time is between 9:00 PM and midnight; as soon as the thermostat turns off, the segment will test true. In a situation where the water may be hot enough before or after 9:00 PM, you would want to use *both* program segments 4-a and 4-b to satisfy the condition of turning off the water heater only if both requirements are true but cannot know which condition will occur first. The next two examples illustrate some programming pitfalls. Example #4-c looks for two single events to test true, which is almost impossible to achieve unless the thermostat happens to turn off exactly at the moment when the time becomes 9:00 PM.. Example #4-d shows the worst possible mistake; transmitting a single command based on two continuous events. Your house wiring would be flooded with a continuous stream of X-10 signals as long as the thermostat is off after 9:00 PM. This could actually be ok if you were only setting a variable or other internal value to remember the status, but any program segments sending out individual commands like X-10, IR, or module output changes needs to be verified to avoid this, or else your controller may even appear to lock up.

2.6 Timing is everything

Writing predictable programs requires that you know when events like timer increments, received X10 commands, etc. occur and become available to the running program. Many times, the order in which you will place your program segments will be important, based on the fact of which commands will detect an event first. This is so important that each command type in the reference section of the manual explains it's timing in relation to the when they occur and when they become valid in the code. It is best to look at the reference section for each specific case but here is a summary classified by how the various command types work:

X10 input

IR input

Touch button presses on a Leopard

I/O latched reads ("IF Module/Point" commands)

I/O analog reads ("IF Module/Parameter" commands)

The main program will look at each one of these queues (or cached value for I/O reads) between program passes and if there is an input for any of these, that value will be true for one entire pass through the program. The I/O values stay as they are for all passes until the bus routine updates them from a new reading, but still only starts to give the new value between passes.

Serial (input and output) Leopard screen updates (eg: displayed variables)

As soon as the complete serial read or write command is received or whenever the screen processor asks for an update, and can occur between any two-program lines.

X10 output IR output I/O value sets (analog and I/O points)

These are put in an output queue and get sent out as soon as their respective processors or routines can send them, irrespective of the running C-Max program which is free to continue looping simultaneously.

Timers

Timers roll over to the next value between passes (unless modified right in the program code). Thus if your program loops 3 times per second, you will get the same value during 3 passes, then the next for 3 passes, etc. Furthermore, timers are aligned with the system clock, so timers all roll over to the next value together, when the real time clock advances to a new "second". This means that if you happen to start a timer just at the very end of a second and the clock rolls over for the next pass, then the timer will have advanced to the next value in that pass. Because of this, timers are only precise to +/- 1 second and short intervals are less precise (% wise) because of this.

Time and Date

Like the timers which they are related to the system clock as explained above, these roll over between passes, so the lowest numbered lines always see a new time or date first.

2.7 C-Max Variables

C-Max variables are integers with values ranging from 0 to 65535. This apparently odd range is because they are internally represented by 16 bit binary memory locations. In a binary system, each bit (from right to left) represents an increasing value of a power of two. Thus the first bit is the 1's, then the 2's, the 4's, the 8's and so on until we get to the 16th bit which is the 32768's. This is just like the more familiar decimal system where we have the 1's, the 10's, the 100's, etc. The maximum value that can be represented is when we have each bit set. Therefore if we add $1 + 2 + 4 + 8 + \dots + 32768 = 65535$.

You will notice that in C-Max, there are no negative numbers. The above description of your controller's variables is called an "unsigned 16 bit integer". If you try to make a variable go below zero by subtracting or decrementing it past zero, you will see noticed that it "rolls over" to 65535 and then continues to decrement downwards. This is very similar to a mechanical car odometer that counts up to 99999 and then rolls over to 00000. If you then turn it backwards it will show 99999 again and keep decreasing. In binary number usage, there is a convention that allows us to consider a 16 bit number as a "signed 16 bit integer". In this case, the highest bit indicates whether the number is positive or negative (1 = negative, 0 = positive) and the remaining 15 bits give us the number's value. We still have the same total range but it is now "shifted" to represent numbers from -32768 to 32767. In that system, called "two's complement", the bit pattern for 65535 is equal to "-1", $65534 = -2$ and so on. This means that the rolling over of your controller's variables from 0 to 65535 when subtracting or decrementing is already the correct behavior for signed 16 bit integers. Even though you cannot see negative numbers in the C-Max editor, you can still display them on the Leopard II's screen by using a formatting option that instructs it to interpret it as a signed 16 bit value. See the chapter on Variable Formatting Options to learn how to do this.

Although you can interpret and display a negative number on your Leopard screen or in an ASCII string, they are still positive-only in the C-Max instruction logic. This can cause a few surprises if you are not careful with "greater than/less than" type instructions. For example, if you are looking for a temperature as being between -10 and +10 deg F for a true condition, you will need to consider it logically as "IF < 11 OR

> 65525". You cannot use an AND because you are looking for a number that is possibly at either end of the positive number scale. If you want to enter a negative number in the editor, type it in by starting with the "-" sign and the editor will automatically convert it to the signed equivalent (eg. Enter "-11" and you will see "65525"). Some instructions like **IF Compare Bobcat Data** expect the numbers to range from -100 to 155 and will display them directly as negative in the editor.

3 C-Max Command Reference

This section of the manual explains the detailed options available for each type of C-Max command. These descriptions are current as of C-Max 2.00e and may evolve in subsequent versions to add new commands or modify/enhance existing ones. Make sure you read any release notes included with a new version to learn about the latest changes.

C-Max commands are of two different types: test statements and action statements. The test statements evaluate the conditions specified in the command options and determine if the statement is true or false. A test statement will be prefixed by one of: IF, AND, or OR; depending on the interaction desired with other test statements before and after it. See the section on **Programming the Leopard II** to learn how test statements interact to create complex logic.

Action statements instruct the controller to do something, like send I/O commands, set variables and timers, display information of the Leopard II screen, etc. . Action statements are prefixed with either THEN or ELSE. A THEN statement will execute if the test conditions prior to it are true, while an ELSE statement looks for the conditions to be false. The END statement is a standalone action statement that executes in all cases (ie: whether the test condition is true or false).

The following pages describe each statement type along with the options available. First the test statements will be described, each one listed in the order that they appear (from the top, down) in the C-Max Control Wizard. Following that are the action statements, also arranged by Control Wizard order. For brevity, only IF and THEN will be shown as the prefixes for the commands although the syntax remains the same if the other prefixes are used. The *description* section describes the various options and behavior of the command. A short *example* of each command usage is shown. The *timing* paragraph describes what timing considerations apply to this command, important for predictable operation. The *work variable* paragraph (test statements only) describes what data will be captured if a **Load Data to Variable** command follows the test command being described. Finally you will see *notes*, explaining any special considerations pertaining to this command.

IF Module / Point *Module#, I/O point, I/O Status*

Description: Tests the status of an expansion module I/O point. An I/O point is either an input such as the ones found on SECU16 or SECU16I modules or output relays like the ones on the SECU16 or RLY8XA. Module inputs and outputs are determined to be either *on* or *off* by the module's internal logic (see module documentation). The Leopard II maintains an internal status table for each possible I/O point. The **IF Module / Point** command consults this table to see if the I/O point is currently *on* or *off* (Is OFF, Is ON) or if the transition from one state to the other has just occurred (Turns OFF, Turns ON). This latter option simplifies the creating of a single triggering action when only detecting the change is important.

Example:

```
0001 - IF Module #1 -SECU16 Input #0 Turns ON // If switch on input 0 closes
0002 - THEN X-10 House J / Unit 1, Turn ON // turn on light at J/1
```

Timing: The controller's bus routine interrogates the modules continuously, independently from the running C-Max program. Whenever a transition from *on* to *off* (or vice versa) occurs, the "Turns" action will be true for one complete pass through the program, at the beginning of the pass following the one during which the change actually occurred.

Work Variable: None. Always 0.

Notes: Your expansion module must be defined in the System Map or else the control wizard will not show you the I/O point and I/O status options. See the manual section on **Projects** to learn how to create/maintain the System Map.

IF Module / Param *Module#, Parameter#, Comparison, Data*

Description: Compares the value of an analog input (module) or a slave variable (slave controller) against the specified data; either a constant or the contents of a variable. Analog module inputs such as the ones available on the SECU16, SECU16I or some Bobcat modules return a value ranging from 0 to 255. Consult the module's documentation to see the available values and corresponding parameter numbers. When the specified *module#* is a slave controller, this command will obtain the contents of the slave's variable that corresponds to the parameter number, and the returned value can range from 0 to 65535. The requested slave *parameter#* can range from 0 to 51.

Example:

```
0001 - IF Module #1 -Slave Leopard Variable #4 is = 55 // If slave's var. 4 = 55
0002 - THEN Load Data to: Variable #7 // capture value in local var. 7
```

Timing: The controller's bus routine interrogates the modules continuously, independently from the running C-Max program. The latest value for a module/parameter pair is always cached in the controller's memory so that the program line can be executed immediately. This means that depending on the length of the program and the number of expansion modules, the program may make one or more passes through the code before a new version of the analog value is obtained. Whenever a new, updated parameter value does become available, it will be presented at the beginning of the next pass through the program. This allows for predictable "becomes" behavior whereby the lower numbered program lines always see a value change first.

Work Variable: Captures the analog value of the parameter being tested in the work variable.

Notes: Your expansion module or slave controller must be defined in the System Map or else the Control Wizard will not show you the parameter, comparison, and data options. See the manual section on **Projects** to learn how to create/maintain the System Map.

The large potential number of modules and number of parameters per module means that the master cannot track all of them; too much time would elapse between updates. Instead, the controller establishes a table of module #'s and parameter #'s to keep track of. The table is built up during the first pass following the starting up of the program and any **IF Module/Param** commands get their module and parameter number added to the parameter tracking list. This list has a maximum of 40 entries (as of C-Max 2.00e). This means that if you have more than 40 such commands in your program code, the 41st and any subsequent such lines will be ignored. Each **IF Module/Param command** adds an entry to the list, even if the same module and parameter number are already entered from a previous line. This means that if you will be making several comparisons against the same module/parameter pair, it might be worthwhile to use one **IF Module/Param** early in your program and capture the value in a local variable (using the work variable). You can then make your multiple comparisons against the local variable instead.

Master and slave controllers have *internal* parameters that are set using C-Max to configure such things as their Adnet address and other options. Reading a slave parameter with the **IF Module/Param** will *not* read these values, but will instead return the slave's corresponding variable number as described above.

IF Receive Single X10 House code, Unit or Command code

Description: Looks for the reception of a single X10 command consisting of a *house code* and either a *unit code* or *command code*. The house code is selected using the radio buttons while the unit or command code is selected from the list box. For the command codes, the list box also shows the numerical equivalent in parenthesis. To look for X10 command *pairs*, see **IF X10 Status/Cmnd Pair**.

Example:

```
0001 - IF Receive X10, B - 6                // If module at B/6 is being addressed
0002 -     THEN Timer #3 = 1                // Then start timer 3
0003 - IF Receive X10, B - Status Request (31) // If Status Request is received
0004 -     AND Timer #3 is > 0              // and timer is running
0005 -     THEN Transmit X10, B - Status ON (29) // reply with Status ON
0006 -     THEN Timer #3 = 0                // and stop the timer
0007 - IF Timer #3 becomes > 4              // if timer exceeds 3 seconds
0008 -     THEN Timer #3 = 0                // then just stop the timer
```

Timing: X10 commands are received by a separate processor in the Leopard II. Whenever a complete X10 command is received, it is placed in the X10 input buffer and at the beginning of every program pass the buffer is checked for any X10 commands in it. If there is one or more, the oldest one (ie: the one that was received first) is made available as the current incoming X10 command for comparison by this command throughout the pass. Any other buffered commands will similarly be read on subsequent passes.

Work Variable: The packed-format buffer value of the current X10 command will be in the work variable. The command is packed in a way that allows a complete X10 command *pair* to be held in a 16-bit variable. This is because the X10 input buffer is also used by the **IF X10 Status/Cmnd Pair** command, which can look for a command pair. You can parse the 16-bit variable into two individual 8-bit fields using the divide and modulo math operators. The fields break down as follows (values are shown in hexadecimal):

Highest 8 bits = House code.

00 = House code A
01 = House code B
...
0F = House code P

Lowest 8 bits = Command/Unit Code:

00 = unit code 1
01 = unit code 2
...
0F = unit code 16
10 = All Units OFF
11 = All Lights ON
14 = Dim
15 = Bright
16 = All Lights OFF
17 = Extended Code
18 = Hail Req.
19 = Hail Ack.
1A = Preset Dim 0
1B = Ext. Data
1C = Status ON
1D = Status OFF
1E = Status REQUEST
1F = Preset Dim 1
40 = unit 1 OFF
41 = unit 2 OFF
...
4F = unit 16 OFF
80 = unit 1 ON
81 = unit 2 ON
...
8F = unit 16 OFF

When there are no X10 commands in the input buffer, the value returned is 6363 hex. Note that all the values listed above are in hex. The variables normally display in decimal so be sure to make the appropriate conversion.

Notes: This command is useful when you're looking for raw X10 or for some of the more unusual X10 commands. Since the most common X10 command usage is with command pairs (eg: B/6 followed by B/ON to turn the device with address B/6 On) the **IF X10 Status/Cmnd Pair** command is more useful to look for those. Since this command is usually used in more unusual circumstances, System Map names are not used either, given that house code/unit code pairs might not be referring to the device at that address but to things like a preset dim level instead. You will notice that there are no standalone ON or OFF commands in the work variable table. This is because receiving a command pair such as B/6, B/ON will actually return (in the packed variable) 0105 hex (B/6) followed by 0185 hex (B/6 ON). In other words, the previously addressed unit code is retained and encoded along with the subsequent B/ON command to put the entire B/6 ON command pair in the input buffer location. This is what allows the **IF X10 Status/Cmnd Pair** command to look for a command pair while needing only one input buffer value.

The program example above shows one use for this instruction. Requesting X10 status is a command pair operation whereby you send the house code/unit code followed by the house code/Status Request command. Since C-Max doesn't have a "Status Request command pair" option under the **IF X10 Status/Cmnd Pair** command, you can make your own equivalent by looking for the unit being addressed (line 1). If it is, start a timer that will give the program 3 seconds to receive the B/Status Request command. Lines 3 to 6 look for the B/Status Request command to be received while the timer is running. If it is, it returns the ON status and stops the timer. If the B/Status Request command is not received within 3 seconds, lines 7 and 8 cause the waiting period to simply time out.

The Leopard II will receive any X10 commands it transmits itself, and will process them like commands from any other source.

The Leopard II cannot transmit or receive IR and X10 simultaneously. You therefore cannot create a routine that looks for a stream of IR commands to perform X10 simultaneous dimming, or an X10 dimming stream to perform an IR function like adjusting a volume level. You need to stop sending one type of data to let it transmit the other.

IF Compare Bobcat Data *Module#, ,Comparison, Data*

Description: Compares the analog value of a Bobcat against the specified data; either a constant or the contents of a variable. Bobcat modules return a value ranging from -100 to 155 (see the chapter earlier in this section on C-Max and negative numbers to learn how to process them in program code). The Bobcat's documentation will explain the meaning of the values returned, and this varies by Bobcat type.

Example:

```
0001 - IF Module #1      -BOBCAT-T becomes < 32    // If temperature goes below freezing
0002 -     THEN X-10 House E / Unit 8, Turn ON      // turn on the heat.
0003 - IF Module #1      -BOBCAT-T becomes > 34    // If temp. is now 3 degrees above freezing
0004 -     THEN X-10 House E / Unit 8, Turn OFF    // turn off the heat
```

Timing: The controller's bus routine interrogates the modules continuously, independently from the running C-Max program. The latest value for Bobcat is always cached in the controller's memory so that the program line can be executed immediately. This means that depending on the length of the program and the number of expansion modules, the program may make one or more passes through the code before a new version of the analog value is obtained. Whenever a new, updated Bobcat value does become available, it will be presented at the beginning of the next pass through the program. This allows for predictable "becomes" behavior whereby the lower numbered program lines always see a value change first.

Work Variable: Captures the analog value of the parameter being tested in the work variable.

Notes: Your Bobcat must be defined in the System Map or else the Control Wizard will not show you the comparison and data options. See the manual section on **Projects** to learn how to create/maintain the System Map. Capturing the Bobcat value with the work variable can be useful if your intention is not just to compare the value but actually capture it in a variable for logging it or displaying it on the Leopard II screen.

IF Timer (seconds), Timer#, Comparison, Data

Description: Compares the current value of a Timer against the specified data; either a constant or the contents of a variable. A Timer set to a value other than 0 will increment once per second until stopped by either setting it 0 by program code or if it counts up to 65535 after which it rolls over to 0 and stops by itself.

Example:

```
0001 - IF Timer #0 becomes > 10           // If timer becomes equal to 11 or more
0002 -     THEN Timer #0 = 0             // then stop the timer
0003 -     THEN Leopard, Zone 0 IR# 45    // and transmit IR code # 45
```

Timing: Timers can be set to any value by program code and will stay at that value for the remainder of that program pass (unless modified again by another instruction later in the same pass). Between every pass, the controller verifies if the controller's real time clock has advanced to the next second. If it has, all running timers are incremented by 1 at the same time. This means that timer increments are aligned with the system's real time clock and always occur between program passes. This allows for predictable "becomes" behavior whereby the lower numbered program lines always see a value change first.

Work Variable: Captures the current value of the Timer in the work variable.

Notes: The timing considerations (as explained above) are the most important things to note about this instruction. Since the program looping runs independently of the clock (a shorter program will loop more times per second than a long one), the time between a timer being set to a value other than 0 and the time it first increments can be very short. This will happen if the real time clock was due to roll over to the next second during the program pass in which the Timer's value was set. Because of this, a timed interval is precise to within a second. A very long program (2000 lines or more) could take more than a second to complete a loop, which means that a timer could theoretically increment by 2 between two consecutive passes. If that risk is present, it is better to avoid looking for a Timer being equal to a value and instead look for it to becoming greater than the desired value - 1 (as shown in the example above), which will still test true even if it was to increment by 2.

IF Variable, Variable#, Comparison, Data

Description: Compares the current value of a Variable against the specified data; either a constant or the contents of another Variable.

Example:

```
0001 - IF Variable #3 becomes = 1         // If variable becomes = 1
0002 -     THEN Send Module #1 -SPEAK-EZ Audio Message #4 // then say message #4
```

Timing: Variables are set by program code immediately and will stay at that value until modified again by another instruction later during program execution. If the Leopard II is connected to an external computer or device that sets variables using the serial protocol, the change can occur between any two program lines, as soon as the serial protocol command is received. When a variable's value changes, a "becomes" test for it will correctly see that change for an entire *loop* through the program. This means that if line # 75 changes Variable #3 from 0 to 1; a line looking for variable #3 becoming = 1 will test true whether that line is after line# 75 during the same pass or if the line is before line #75 during the following pass. This allows for predictable "becomes" behavior regardless of relative line position. The "becomes" behavior is accomplished in the following manner: At every pass, the specified logic test is done and a one bit flag is saved in memory indicating if the test was true or not. On subsequent passes, the logic test is done again and the true or false result is compared against the status of the saved one bit flag. If the current logic test is true but the one bit flag indicates that in the previous pass the logic test was false, then it is determined that the test has "become" true and the statement itself is deemed to be true. If the logic test is true but the flag indicates that it was also true in the previous pass, then the statement will be deemed false since the logic test hasn't just "become" true, but already was.

Work Variable: Captures the current value of the Variable in the work variable.

Notes: Among the configuration parameters that govern the Leopard II's operation, parameter# 22 sets the limit for the non-volatile variables. This means that the values for variable numbers at or above the number set in parameter# 22 will be retained even during a power failure, while the variables below that number will always be reset to 0 upon a power up. Having variables that predictably reset to 0 upon a restart can be useful to guarantee detection of a power interruption and to start in a known state. Having non-volatile variables can also be useful to retain certain things like user modified setpoints for temperatures or times even if the power fails.

When scrolling through the list of variables that may be tested, you will see that the variables go from 0 to 127, and then continues with a list of "Data for Module #x" where x goes from 0 to 127. These are referred to as the *extended variables* and correspond to the raw data value (0 to 255) of Bobcat modules like the temperature Bobcat, humidity Bobcat, etc. For non-Bobcat modules, these variables will not contain any valid data.

IF Time of Day, Comparison, Data

Description: Compares the current Time of Day against the specified data; either a constant, a variable, or the current day's sunrise or sunset time (with an offset of up to 120 minutes). The time is in military (24 hour) format ie: 6:00 PM = 18:00 and would be entered as 1800 .

Example:

```
0001 - IF Time of Day becomes = 18:00 // At 6:00 PM
0002 - THEN X-10 House F / Unit 7, Turn OFF // stop the pool filter
0003 - IF Time of Day becomes = Sunset offset 30 minutes // At half hour past sunset
0004 - THEN X-10 House D / Unit 12, Turn ON // turn on porch light
```

Timing: The Time of Day is obtained from the Leopard II's internal real time clock (system clock). The sunrise and sunset times as well as daylight savings or standard time are calculated, based on the daylight savings time check box and longitude/latitude values entered in the C-Max **Comms Setup** screen. Whenever the time advances to the next minute, the new time value will be valid beginning with the next pass. This allows for predictable "becomes" behavior whereby the lower numbered program lines always see a new time first.

Work Variable: When the comparison involves a constant or a variable's contents, the current value of the Time of Day* is captured in the work variable. If the comparison is against sunrise or sunset, then the sunrise or sunset time* for that day will be captured in the work variable. The current day's sunrise or sunset time is available as soon as the day begins, at midnight. This allows you to use it for calculations ahead of the actual sunrise/sunset times, or for display purposes.

* The Time of Day and sunrise or sunset time is captured as a "minutes after midnight" value. For example, 6:00 PM which is normally 1800 in military time will be returned as 1080 (18 x 60). You can use division and modulo to convert it to military time as follows (military time will appear in variable# 1):

```
0001 - IF Time of Day is > 00:00 // look at Time of Day
0002 - THEN Load Data to: Variable #0 // and capture in Var# 0
0003 - THEN Variable #1 = Variable #0 // and also copy to Var# 1
0004 - THEN Variable #1 / 60 // calculate hours
0005 - THEN Variable #1 * 100 // convert hours to HHxx
0006 - THEN Variable #0 % 60 // calculate "minutes" value
0007 - THEN Variable #1 + Variable #0 // add to hours: HHMM
0008 - ELSE Variable #1 = 0 // if midnight, then time = 0
```

Notes: Since the Time of Day is a number that increases in value from midnight to 11:59 PM, numerical comparisons need to take that into account for proper behavior. For example, if you want to write a program segment that tests true for nighttime (after sunset and before sunrise), you need to use an OR statement since you're looking for a value that can be at both ends of the Time of Day scale:

```
0001 - IF Time of Day is > Sunset offset 0 minutes // If past sunset
0002 - OR Time of Day is < Sunrise offset 0 minutes // or before sunrise (nighttime)
```

While a test for daytime uses an AND statement:

```
0001 - IF Time of Day is > Sunrise offset 0 minutes // If past sunrise
0002 - AND Time of Day is < Sunset offset 0 minutes // and before sunset (daytime)
```

Be careful about scheduling any critical tasks between 1:00 and 3:00 AM because this time period falls in the adjustment range when the time switches from standard time to daylight savings time or vice versa. For example, when going from daylight savings time to standard time in the Fall, the time goes from 2:00 AM to 1:00 AM and an event scheduled between 1:00 AM and 2:00 AM will be executed twice. Similarly, in the Spring the time changes from 2:00 AM to 3:00 AM and an event scheduled between these two times will not be executed.

IF Month, Comparison, Data

Description: Compares the current month against the specified data; either a constant or a variable. Months are numbered from 1 (January) to 12 (December)

Example:

```
0001 - IF Month is = December (12) // If in December
0001 - AND Time of Day becomes = Sunset offset 0 minutes // and at sunset
0002 - THEN X-10 House E / Unit 3, Turn ON // turn on Christmas lights
```

Timing: The month is obtained from the Leopard II's internal real time clock (system clock). The month will advance to the next month at midnight and the new month value will be valid beginning with the next pass. This allows for predictable "becomes" behavior whereby the lower numbered program lines always see a new month first.

Work Variable: The current month number will be captured in the work variable.

Notes: None.

IF Day of Month, Comparison, Data

Description: Compares the current day of month against the specified data; either a constant or a variable. Days of the month are numbered from 1 to the last day of the current month.

Example:

```
0001 - IF Month is = December (12) // If in December
0001 - AND Time of Day becomes = Sunset offset 0 minutes // and at sunset
0002 - THEN X-10 House E / Unit 3, Turn ON // turn on Christmas lights
```

Timing: The day of month is obtained from the Leopard II's internal real time clock (system clock). The day of month will advance to the next day at midnight and the new day of month value will be valid beginning with the next pass. This allows for predictable "becomes" behavior whereby the lower numbered program lines always see a new day of month first.

Work Variable: The current day of month number will be captured in the work variable.

Notes: The number of days for any given month is automatically calculated by the system clock. Adjustments for leap years are also automatic, adding Feb. 29th as necessary. Sometimes it is necessary to determine if the day of the month is odd or even for things like lawn watering routines, etc. You can use the following routine to create an odd/even flag that can then be used elsewhere in your program:

```
0001 - IF Day of Month is > 0 // always true...
0002 - THEN Load Data to: Variable #0 // store in Variable #0
0003 - THEN Variable #0 % 2 // get modulo 2 (0 = even, 1 = odd)
```

Variable #0 will be equal to 0 on even numbered days and equal to 1 on odd numbered days.

IF Day of Week, Comparison, Data

Description: Compares the current day of the week against the specified data; either a constant or a variable. Days of the week are numbered from 0 to 6:

0 = Sunday
1 = Monday
2 = Tuesday
3 = Wednesday
4 = Thursday
5 = Friday
6 = Saturday

Example:

```
0001 - IF Day of Week is = Wednesday (3) // If today is Wednesday
0002 - AND Time of Day becomes = 06:00 // at 6:00 AM
0003 - THEN Send Module #3 -BOBCAT-A Message 34 w/ Variable #0 // Garbage day reminder...
```

Timing: The day of week is obtained from the Leopard II's internal real time clock (system clock). The day of week will advance to the next day at midnight and the new day of week value will be valid beginning with the next pass. This allows for predictable "becomes" behavior whereby the lower numbered program lines always see a new day of the week first.

Work Variable: The current day of week number will be captured in the work variable.

Notes: None.

IF Year, Comparison, Data

Description: Compares the current year against the specified data; either a constant or a variable. Years are entered as 4 digits.

Example:

```
0001 - IF Year is = 2003 // If year is 2003
0002 - THEN Load Data to: Variable #90 // capture work variable
0003 - ELSE Load Data to: Variable #90 // no matter what year
```

Timing: The year is obtained from the Leopard II's internal real time clock (system clock). The year will advance to the next year at midnight and the new year value will be valid beginning with the next pass. This allows for predictable "becomes" behavior whereby the lower numbered program lines always see a new year first.

Work Variable: The current year number will be captured in the work variable as a two significant digit value (ie: 2003 = 03 so the variable will contain 3).

Notes: Performing actions based on the year value is quite rare, but capturing the year value in the work variable (as shown in the example above) can be useful for event logging routines where you want to include the date and time in formatted strings sent from a serial device like the Leopard II's RS-232 port or a serial Bobcat.

IF Date (mm/dd/yy), Comparison, Date

Description: Compares the current date against the date selected from the calendar. Clicking on the date list box will display a calendar allowing you to choose the date you want. On each side of the month-year name you will see an arrowhead button allowing you to go to the previous or next month. When you scroll past the first or last month of a year, the year will adjust and you will be able to keep scrolling through the previous or next year as applicable. Once

you have the desired month and year displayed, click on the day of the month that you want. As a convenience, the current day's date will be encircled in red when the current month is selected.

Example:

```
0001 - IF Date is = 06/05/03 // If June the 5th
0002 - THEN Send Module #3 -BOBCAT-A Message 0 w/ Variable #5 // display birthday message
```

Timing: The date is obtained from the Leopard II's internal real time clock (system clock). The date will advance to the next date at midnight and the new date value will be valid beginning with the next pass. This allows for predictable "becomes" behavior whereby the lower numbered program lines always see a new date first.

Work Variable: None. Always 0.

Notes: None.

IF Receive IR Module, IR

Description: Looks for the reception of a recognized Infrared (IR) code from the master or a slave controller, as specified in the control wizard list boxes.

Example:

```
0001 - IF Receive IR IR #19, from Leopard // If "TV On" IR code is received
0002 - THEN X-10 House N / Unit 8, Turn ON // turn on lamp
0003 - THEN Transmit X10, N - Dim (21), 5 time(s) // and dim it
```

Timing: IR codes are received by a separate processor in the Leopard II. Whenever a recognized IR code is received, it is placed in the IR input buffer and at the beginning of every program pass, the buffer is checked for any received IR codes in it. If there is one or more, the oldest one (ie: the one that was received first) is made available as the current incoming IR code for comparison by this command throughout the pass. Any other buffered commands will similarly be read on subsequent passes.

Work Variable: The packed buffer value of the current IR code will be in the work variable. The code is held in a 16 bit variable. The upper 8 bits contain the module number (adnet address) of the controller that received the code while the lower 8 bits contains the code number itself. If there are no recognized IR codes in the input buffer, a value of 65535 is returned. You can parse the 16 bit variable into two individual 8 bit fields using the divide and modulo math operators:

```
0001 - IF Receive IR IR #0, from Leopard // Look for an IR code
0002 - THEN Load Data to: Variable #1 // capture in variable #1
0003 - ELSE Load Data to: Variable #1 // in either case
0004 - IF Variable #1 becomes NOT = 65535 // If there is a code in input queue
0005 - THEN Variable #2 = Variable #1 // copy code to variable #2
0006 - THEN Variable #2 / 256 // get module number in var #2
0007 - THEN Variable #1 % 256 // get code number in var #1
```

Notes: Any slave controller(s) must be defined in the System Map or else the Control Wizard will not show you the module list properly to then allow you to select the IR code. Consult the C-Max utilities section of the manual to see how to learn and/or load IR codes into your controller(s). Among the configuration parameters that govern the Leopard II's operation, parameter# 20 sets the highest IR code for which the controller will try to find a match whenever an IR signal is received at the controller's IR sensor. Up to 256 IR codes can be compared against a received code before the controller decides that none of them match. It takes the controller a certain amount of time to go through all the codes so the best performance will be obtained when the controller has to search through the least amount of codes before giving up. Because of this, often recognized codes should ideally be learned in the lower code locations and parameter 20 should be set to the lowest number possible (ie: no higher than the actual amount of codes that need to be recognized).

The Leopard II cannot transmit or receive IR and X10 simultaneously. You therefore cannot create a routine that looks for a stream of IR commands to perform X10 dimming, or an X10 dimming stream to perform an IR function like adjusting a volume level. You need to stop sending one type of data to let it transmit the other.

IF X10 Status/Cmnd Pair *X10 house/unit code, X10 Status*

Description: Looks for an X10 status or command pair. For X10 status, can look at the present status for the specified house code/unit code in the X10 status table (see *notes* below) or for a change in status. The instruction can also look for the reception of an actual *On* or *Off* command pair from the powerline. To look for *individual* X10 commands, see **IF Receive Single X10**.

Examples:

```
0001 - IF X-10 House J / Unit 4, Is OFF // If air conditioner is Off
0002 - AND Module #1 -BOBCAT-T becomes > 76 // and temp goes above 76
0003 - THEN X-10 House J / Unit 4, Turn ON // turn it on.
0004 - //
0005 - IF X-10 House L / Unit 2, ON Command Pair // If On command pair for lamp
0006 - AND Time of Day is > Sunrise offset 0 minutes // and after sunrise
0007 - AND Time of Day is < Sunset offset 0 minutes // and not sunset yet
0008 - THEN Variable #4 = 1 // set lamp request flag to 1
```

Timing: X10 commands are received by a separate processor in the Leopard II. Whenever a complete X10 command is received, it is placed in the X10 input buffer and at the beginning of every program pass the buffer is checked for any X10 commands in it. If there is one or more, the oldest one (ie: the one that was received first) is made available as the current incoming X10 command for comparison by this command throughout the pass. Any other buffered commands will similarly be read on subsequent passes.

Work Variable: The packed-format buffer value of the current X10 command will be in the work variable. See the description of the **IF Receive Single X10** command to see the values that are returned.

Notes: The difference between looking at the status table and for the actual commands can be subtle, but important. The Leopard II maintains an in-memory status table of all 256 X10 addresses, and memorizes that the device is either *on* or *off*. This can be used as a flag in making logic decisions, avoiding the need to use variables to keep track of X10 device status. The *on* or *off* status of a device will obviously be updated with the reception of the corresponding command pair for the given house and unit code, but also with the reception of a *Status ON* or *Status OFF* in response to a *Status Request* command. These commands do not have to be generated by the Leopard II itself, but can come from any X10 device (the Leopard II analyzes all X10 traffic on the powerline). The status transmitting device might also be configured to transmit it's own status periodically. X10 *Dim* or *Bright* commands do not affect the status table entry of an X10 device. Looking for the actual command pair allows you to ignore the fact that a device is already *on* or *off*, letting you look for the significance of receiving the command multiple times, or when the device has local on/off control and it's status cannot be reliably determined just from powerline commands.

The Leopard II will receive any X10 commands it transmits itself, and will process them like commands from any other source.

The Leopard II cannot transmit or receive IR and X10 simultaneously. You therefore cannot create a routine that looks for a stream of IR commands to perform simultaneous X10 dimming, or an X10 dimming stream to perform an IR function like adjusting a volume level. You need to stop sending one type of data to let it transmit the other.

IF I/O Error Occurs

Description: Traps Adnet bus I/O Errors (if the master can no longer access an expansion module that was present when the master was first powered up or restarted)

Example:


```

0001 - IF I/O Error Occurs //if I/O error
0002 - THEN Load Data to: Variable #0 //put Adnet mod# in variable# 0

```

Work Variable: Returns the Adnet address of the unresponsive module. Returns the highest affected module address if several modules become unresponsive.

Notes: You can use this instruction to log the error through the serial port or display an error message on the screen if one or more modules fail to respond to polling from the master controller. This can be useful for critical applications like reading temperatures or liquid levels, where a failure to keep control could have materially harmful consequences. If more than one module fails, then the highest numbered one (address-wise) will be the one reported in the work variable. This could happen due to a wiring or power fault affecting several modules at once.

You need to verify and possibly adjust parameter 9 in your master to specify how many consecutive unsuccessful attempts must be made before the error condition is trapped (a value between 3 and 5 should be good). Note that if the module reappears on the bus following such an error, it will then again be accessed, without the master needing to be rebooted. You may also want to adjust parameter 8 to reduce the time wasted waiting for an unresponsive module, a value between 5 and 10 for this parameter should be sufficient.

IF Touch Button Pressed *Module, Button*

Description: Looks for a touch button (as defined on the screen) having been pressed on the master Leopard or a slave Leopard controller, as specified in the control wizard list boxes.

Example:

```

0001 - IF Touch Object #2, Button Leopard is pressed // if button 2 is pressed
0002 - THEN Variable# 63 / Screen# = 5 // display screen # 5

```

Timing: Touch button presses are detected by a separate processor in the Leopard II. Whenever a touch button object is pressed on the screen, its object number is placed in the touch button input buffer and at the beginning of every program pass, the buffer is checked for any touch button codes in it. If there is one or more, the oldest one (i.e.: the one that was pressed first) is made available as the current touch button object number for comparison by this command throughout the pass. Any other buffered button presses will similarly be read on subsequent passes.

Work Variable: The packed buffer value of the current touch button code will be in the work variable. The code is held in a 16 bit variable. The upper 8 bits contain the module number (Adnet address) of the controller on which the button was pressed while the lower 8 bits contains the button's object number itself. If there are no touch button codes in the input buffer, a value of 65535 is returned. You can parse the 16 bit variable into two individual 8 bit fields using the divide and modulo math operators:

```

0001 - IF Touch Object #1, Button Leopard is pressed // if any button is pressed
0002 - THEN Load Data to: Variable #1 // capture in variable #1
0003 - ELSE Load Data to: Variable #1 // in either case
0004 - IF Variable #1 becomes NOT = 65535 // If there is a button code in input queue
0005 - THEN Variable #2 = Variable #1 // copy code to variable #2
0006 - THEN Variable #2 / 256 // get module number in var #2
0007 - THEN Variable #1 % 256 // get button number in var #1

```

Notes: Any slave controller(s) must be defined in the System Map or else the Control Wizard will not show you the module list properly to then allow you to select the touch button code. Consult the C-Max user guide section of the manual to see how to create touch screen layouts for your controller(s).

Touch button #0 is defined as the “touch anywhere” button. You will get this code whenever you press any area on the screen that has a valid button defined, or press anywhere on screen #0; the bitmap display screen. Touching a valid button will in fact always queue two button press events; a “0” for the “touch anywhere” event followed by that actual touch button object number. These two values will be returned on two consecutive passes through the

program. If the screen is displaying screen #0, you will get the object “0” event followed by touch object #255 being pressed. This allows you to distinguish between screen 0 being pressed and any other button being pressed.

For touch button presses being detected on the master as coming from a slave Leopard, you will *not* get “touch anywhere” events; only the actual button object number.

THEN Module / Point *Module#, I/O point, I/O command*

Description: Sets an expansion module I/O point On or Off. An I/O point is either an input such as the ones found on SECU16 or SECU16I modules or output relays like the ones on the SECU16 or RLY8XA. Only output relays can be controlled by this instruction.

Example:

```
0001 - IF Time of Day becomes = 05:00           // At 5:00 AM
0002 -     THEN Module #5  -RELAY-08 Relay #2 Turns ON // turn on sprinklers
```

Timing: The controller’s bus routine interrogates the modules continuously and transmits updates as needed, independently from the running C-Max program. Any I/O output commands are queued for transmission and are inserted between the module status interrogations, to maintain timely detection of input events. This means that a certain delay may be observed between the time the command is executed and the module output actually responds. This delay can be up to a second or so. Because of this, very short timing intervals for outputs cannot be reliably produced. If there is a need for short, momentary-type contact activation, please inquire about the availability of momentary closure mode modules. You must also avoid program logic where module output commands are issued on every pass. This will overrun the output buffer and can cause the controller to appear to lock up. Use single triggering logic tests and/or timers to issue module output commands only when needed.

Notes: Your expansion module must be defined in the System Map or else the control wizard will not show you the I/O point and I/O command options. See the manual section on **Projects** to learn how to create/maintain the System Map.

THEN Set Slave Variable *Module#, Variable#, Data*

Description: Updates the value of a slave variable (in a slave controller) with the specified data; either a constant or the contents of a variable. The specified slave variable# can range from 0 to 63.

Example:

```
0001 - IF Touch Object #18, Button Leopard is pressed // If temperature + button is pressed
0002 -     THEN Variable #10 + 1                       // increment setpoint variable
0003 -     THEN Module #3  -Slave Leopard Variable #10 to: Variable #10 // and update slave
```

Timing: The controller’s bus routine interrogates the modules continuously and transmits updates as needed, independently from the running C-Max program. Any slave variable update commands are queued for transmission and are inserted between the module status interrogations, to maintain timely detection of input events. This means that a certain delay may be observed between the time the command is executed and the slave variable is actually updated. This delay can be up to a second or so. You must also avoid program logic where slave variable updates are issued on every pass. This will overrun the output buffer and can cause the controller to appear to lock up. Use single triggering logic tests and/or timers to issue slave variable update commands only when needed.

Notes: Your slave controller must be defined in the System Map or else the control wizard will not show you the variable# command options. See the manual section on **Projects** to learn how to create/maintain the System Map

THEN Load Data to Variable *Variable#*

Description: Copies the contents of the work variable to the specified variable.

Examples:

```
0001 - IF Module #4 -SECU16 Analog #1 is < 256 // read analog input
0002 - THEN Load Data to: Variable #3 // and capture to variable #3
0003 - IF Touch Object #1, Button Leopard is pressed // if a touch button is pressed
0004 - THEN Load Data to: Variable #4 // save in variable #4
0005 - ELSE Load Data to: Variable #4 // in all cases
```

Timing: The value is copied immediately.

Notes: This instruction allows the tested resource's present value (from the preceding test statement) to be copied to a variable for use in the program, display, etc. Whenever a resource like the X10 input command queue, Touch button press queue, a Timer, etc. is referenced by an **IF** statement, the specified resource is loaded into the work variable and then compared against the specified value in the statement; usually a constant or the contents of a variable. The test statement then sets the *true/false* condition according to the result of the logic test. The **Load Data to Variable** command allows you to capture the current value of the resource that was just tested. This greatly enhances the capabilities of the test statements because you can get the value itself, instead of merely testing it for a true or false condition. A good example of this is a SECU16 analog input. You certainly don't want to use 256 IF/THEN statements to test for every possible value. With this statement, you can make one comparison (as shown in the first example above) just to access the value and then store it in a variable.

Since this is an action statement, its execution is still dependent upon the *true/false* status of the test statement preceding it. Because analog input values for an analog SECU16 input always range from 0 to 255, the first example above will always test true and line 2 will always be executed. Other resource types like the touch button queue (second example above) force you to choose the test value from a list and therefore do not allow a test that will be guaranteed to test true at all times. In such cases, use a THEN and an ELSE statement as shown in lines 4 and 5 of the example to insure the capture of the work variable every time.

The actual significance of the work variable's value varies with the type of resource being tested. Read the reference manual entry for each type of test statement to see what the work variable will return. This is explained in detail under the *work variable* paragraph for each test statement.

THEN Transmit Single X10 *House code, Unit or Command code*

Description: Transmits a single X10 command consisting of a *house code* and either a *unit code* or *command code*. The house code is selected using the radio buttons while the unit or command code is selected from the list box. For the command codes, the list box also shows the numerical equivalent in parenthesis.

Example:

```
0001 - IF Touch Object #6, Button Leopard is pressed // turn on kitchen lights
0002 - THEN Transmit X10, K - 1 // address overhead lights
0003 - THEN Transmit X10, K - 3 // then countertop lights
0004 - THEN Transmit X10, K - ON (19) // turn them on
```

Timing: X10 commands are transmitted by a separate processor in the Leopard II. Whenever a statement to transmit an X10 command is executed, the command is placed in the X10 output buffer and program execution resumes immediately. The queued command will be transmitted as soon as any previously queued commands have finished transmitting, independently of the running program.

Notes: This command is useful for sending raw X10 commands. Since the most common X10 command usage is with command pairs (eg: B/6 followed by B/ON to turn the device with address B/6 On) the **X10 Quick ON/OFF** command is more useful to send those. The **Transmit Single X10** is mostly used to: 1- Address several modules before issuing a command that will concern all of them, like shown in the above example. This can include

coordinated dimming too. 2 - Issue commands like Status Requests. 3 – Issue or respond a preset dim level. 4 – Any other situation where you want to issue specific X10 commands.

Because this command is sometimes used in more unusual circumstances, System Map names are not used, given that house code/unit code pairs might not be referring to the device at that address but to things like a preset dim level instead.

The Leopard II will receive any X10 commands it transmits itself, and will process them like commands from any other source.

The Leopard II cannot transmit or receive IR and X10 simultaneously. You therefore cannot create a routine that looks for a stream of IR commands to perform X10 simultaneous dimming, or an X10 dimming stream to perform an IR function like adjusting a volume level. You need to stop sending one type of data to let it transmit the other.

THEN Timer (seconds) *Timer#, Data*

Description: Sets a Timer to the specified data; either a constant or the contents of a variable. A Timer set to a value other than 0 will increment once per second until stopped by either setting it 0 with program code or if it counts up to 65535 after which it rolls over to 0 and stops by itself.

Example:

```
0001 - IF Time of Day is > Sunset offset 20 minutes // if time is past sunset
0002 - OR Time of Day is < Sunrise offset 0 minutes // and before sunrise (night)
0003 - AND Module #1 -SECU16 Input #2 Turns ON // and PIR detects movement
0004 - THEN X-10 House H / Unit 2, Turn ON // turn on hall light
0005 - THEN Timer #4 = 1 // and start the timer
0006 - IF Timer #4 becomes > 900 // if 15 minutes have elapsed
0007 - THEN Timer #4 = 0 // then stop timer
0008 - THEN X-10 House H / Unit 2, Turn OFF // and turn off hall light
```

Timing: Timers can be set to any value by program code and will stay at that value for the remainder of that program pass (unless modified again by another instruction later in the same pass). Between every pass, the controller verifies if the controller's real time clock has advanced to the next second. If it has, all running timers are incremented by 1 at the same time. This means that timer increments are aligned with the system's real time clock and always occur between program passes. This allows for predictable "becomes" behavior whereby the lower numbered program lines always see a value change first.

Notes: The timing considerations (as explained above) are the most important things to note about this instruction. Since the program looping runs independently of the clock (a shorter program will loop more times per second than a long one), the time between a timer being set to a value other than 0 and the time it first increments can be very short. This will happen if the real time clock was due to roll over to the next second during the program pass in which the Timer's value was set. Because of this, a timed interval is precise to within a second. A very long program (2000 lines or more) could take more than a second to complete a loop, which means that a timer could theoretically increment by 2 between two consecutive passes. If that risk is present, it is better to avoid looking for a Timer being equal to a value and instead look for it to becoming greater than the desired value – 1 (as shown in the example above), which will still test true even if it was to increment by 2.

THEN Variable *Variable#, Operation, Source data (operand)*

Description: Performs the specified mathematical or assignment operation to the variable using the source data (a constant or the contents of the specified variable) as the operand and puts the result in the variable.

Example: (convert temperature value from Fahrenheit to Celsius degrees)

```
0001 - IF Module #2 -BOBCAT-T is > -100 // Read temperature Bobcat
0002 - THEN Load Data to: Variable #2 // and capture value in variable 3
0003 - THEN Variable #2 - 32 // subtract 32
0004 - THEN Variable #2 * 5 // multiply by 5
```

```

0005 - THEN Variable #2 / 9 // divide by 9
0006 - THEN Variable #3 = Variable #2 // copy to display variable (now in Celsius)

```

Timing: Variables are set to the new value immediately. See the reference page for the **IF Variable** command to see how a variable read by a device using the serial protocol or a variable linked to a screen text object can return intermediate values because of their timing considerations. If this is a concern, it is best to perform the calculations using a temporary variable and then copy the final result to the variable that will be read by the external device or referenced on the screen.

Notes: You can perform straight assignment (=), the 4 regular math operations (add, subtract, multiply, divide) and modulo (the “%” operator). Modulo is the remainder portion after an integer division. For example: 23 % 3 = 2 because 23 divided by 3 = 7 with a remainder of 2. This is useful for base conversions such as converting from minutes to sexagesimal (base 60) time. The **IF Time of Day** command reference page has an example of using modulo to do this.

Among the configuration parameters that govern the Leopard II’s operation, parameter# 22 sets the limit for the non-volatile variables. This means that the values for variable numbers at or above the number set in parameter# 22 will be retained even during a power failure, while the variables below that number will always be reset to 0 upon a power up. Having variables that predictably reset to 0 upon a restart can be useful to guarantee detection of a power interruption and to start in a known state. Having non-volatile variables can also be useful to retain certain things like user modified setpoints for temperatures or times even if the power fails.

When scrolling through the list of variables that may be selected as the source data, you will see that the variables go from 0 to 127, and then continues with a list of “Data for Module #x” where x goes from 0 to 127. These are referred to as the *extended variables* and correspond to the raw data value (0 to 255) of Bobcat modules like the temperature Bobcat, humidity Bobcat, etc. For non-Bobcat modules, these variables will not contain any valid data.

THEN Skip to (Program Jump) Line#

Description: Instructs the command interpreter to continue program execution at the specified line number.

Example:

```

0001 - IF Touch Object #3, Button Leopard is pressed // If toggle button is pressed
0002 - AND X-10 House A / Unit 3, Is OFF // and light is off
0003 - THEN X-10 House A / Unit 3, Turn ON // then turn light on
0004 - THEN Skip to line 7 // and skip rest of routine
0005 - IF Touch Object #3, Button Leopard is pressed // If toggle button is pressed
0006 - THEN X-10 House A / Unit 3, Turn OFF // then turn light off
0007 - IF ...

```

Timing: Immediate.

Notes: This is the only command (other than the **END** statement) that causes the program flow to continue with a program line other than the following one. You can only jump forward in the program. Using this command to skip statements can greatly simplify program coding because you can create “nested” logic. For example: suppose you have an alarm system that sets a variable as an “away mode” flag. You want to simulate a lived-in look for the home when the system is armed so you would normally have many program segments looking for the Time of Day being equal to a certain value *and* the away mode flag being set to turn lights, etc. on and off. Using the Program Jump instruction, you can look for the flag being set and if not, skip the entire lived-in routine. The lived-in routine would only need to look for the Time of Day, without needing all the AND statements to also test the flag each time.

The example above shows another common use for the Program Jump instruction. Creating a toggle button is an order sensitive task because a touch button press is true for an entire pass through the program, so the first instance would toggle the output and the second one toggle it right back. In our example, we avoid this by skipping over the second toggle if the first one was executed. This also saves us a logic test on the status of X10 device A/3 because the only way line 6 can be reached is if line 5 is true and line 2 is not true.

Note that although the program editor shows the line with a destination line number, internally it stores the destination as an offset from the current line. This means that if you cut/copy and paste a program segment containing Program Jump instructions, the pasted segment will display the newly calculated destination line number(s), again as offsets from where they are. For example, you copy lines 10 to 17 and line 12 is a Program Jump that skips to line 16. If you paste the segment beginning at line 30, the Program Jump line will be at line 32 and will show that it skips to line 36. The editor also automatically adjusts the offset if any program lines are inserted or deleted between a Program Jump instruction and its destination line.

Keep in mind that the current *true/false* status of the preceding test statements stays intact when a jump occurs, so you cannot selectively skip some THEN statements but execute others by using this command. Ideally, a program should always jump to an IF statement.

THEN Set Displayed Icon *Object#, Icon#*

Description: Dynamically selects an icon to be displayed on a touch button. The object# must refer to a touch button, any other screen object type will ignore the command. The icon# can be selected from the list of System Map named icons or by the contents of a variable.

Example:

```
0001 - IF Module #1 -SECU16 Analog #3 is > 64 // if tank is one quarter full or more
0002 - THEN Touch Object #6 displays Icon# Icon #1 // show the "full" picture
0003 - ELSE Touch Object #6 displays Icon# Icon #2 // else show the "low level" picture
```

Timing: Immediate.

Notes: Although the object must be a touch button, nothing prevents you from using a touch button simply as a display-only object if you wish to do so. You can use this command to display detailed status information in the form of an icon, or to reflect the current status of a button that triggers a toggling action. Note that you cannot change an icon on a slave Leopard from the master controller with this command, this can only be done on the local controller. Read the application note on creating icons elsewhere in this manual to see the entire procedure involved in creating and using icons.

THEN Send X10 Thermostat (RCS) *House code, Function, Data*

Description: Sends X10 command to an RCS thermostat. Some functions like specifying the setpoint require a data value.

Example:

```
0001 - IF Time of Day becomes = 16:00 // before coming home from work
0002 - THEN Set RCS X10 B Setpoint 72 // set temperature to 72 degrees
```

Timing: The command is put in the X10 output queue and program execution resumes immediately.

Notes: The RCS thermostat uses the original format *X10 preset dim* command to exchange data to and from a controller like the Leopard II. A given RCS thermostat uses an entire X10 house code for itself. This means that up to 16 RCS thermostats can be independently controlled. The use of the preset dim commands allows the thermostat to send and receive multiple analog values such as the current temperature even though it only has one house code to do so. The value of the dim level can be any one of 32 values for a single unit code, allowing a total of 512 different possible commands or values to be transmitted between the thermostat and the controller. See the RCS thermostat documentation to learn about all the possible commands and values.

Requesting the temperature from a thermostat (or getting the value automatically by virtue of the thermostat's "Auto Send" mode) causes one of many possible preset dim commands to be returned/sent by the thermostat. The Leopard II can be set to automatically look for these responses and store the equivalent temperature value in a pre-designated variable. To do this, set your controller's parameter # 21 to a value of 1. Once this is set, the Leopard II

will store the temperature from the thermostat at house code A in variable # 64, thermostat B in variable # 65, etc. Note that any X10 house codes that don't have RCS thermostats using them will not be using it's designated variable, so the designated variables for those house codes can be used as general-purpose variables.

THEN Send Bobcat Thermostat

Description: This command is for the support of a future product (Bobcat) designed to control non-X10 thermostats.

THEN X10 Preset Dim (PCS) Switch type, House/Unit Code, Dim Level

Description: Encodes and sends an X10 preset dim command to the PCS switch at the specified house and unit code. The house and unit codes will be replaced by the System Map name if one has been entered. The dim level is expressed as a percentage of full brightness and can be selected either directly (using the slider) or by the contents of the specified variable.

Example:

```
0001 - IF Touch Object #4, Button Leopard is pressed // if button 4 is pressed
0002 - THEN Set X-10 House J / Unit 7, to: 45 % // set lights to 45 % brightness
```

Timing: The command is put in the X10 output queue and program execution resumes immediately.

Notes: This command is also used as the **THEN X10 Preset Dim (Leviton)** command. The *switch type* radio button determines which dim level encoding type will be done. The preset dim level can be one of 32 levels for PCS switches, and one of 64 levels for Leviton switches. In either case, the level closest to the specified percentage will be encoded and sent.

THEN X10 Quick ON/OFF House/Unit Code, Command

Description: Transmits an X10 command pair to turn a device On or Off. The house and unit codes will be replaced by the System Map name if one has been entered.

Example:

```
0001 - IF Touch Object #54, Button Leopard is pressed // if button 54 is pressed
0002 - THEN X-10 House B / Unit 4, Turn ON // turn on basement light
```

Timing: The command pair is put in the X10 output queue and program execution resumes immediately.

Notes: This command is provided as a convenient way to turn X10 devices on an off using a single program line, even though two individual X10 commands are actually sent. Without it, you would need to use two consecutive **THEN Transmit Single X10** commands to accomplish the same task.

THEN Send Page Message #

Description: Sends the specified message # to an alphanumeric pager.

Example:

```
0001 - IF Touch Object #54, Button Leopard is pressed // if button 54 is pressed
0002 - THEN X-10 House B / Unit 4, Turn ON // turn on basement light
```

Timing: The sequence to call the pager is initiated and program execution resumes immediately.

Notes: Calling a pager requires the use of the Adicon modem. The modem must first be configured in C-Max. See the C-Max configuration section to see how to set up the pager. The User PIN and up to 15 pager messages are

configured in the controller access utilities. Refer to the manual section on using C-Max to see how you can enter your pager message strings.

Description: Transmits an infrared command (IR) using the specified module and zone numbers. The IR code number can be specified either with it's system map name or by the contents of the specified variable.

Example:

```
0001 - IF Touch Object #49, Button Leopard is pressed // if button 49 is pressed
0002 - THEN Module #4 -SECU16-IR, IR Zone #5 IR# 56 // send ir #56 to output 5 of SECU16IR
```

Timing: The IR command is placed in the IR output queue and program execution resumes immediately.

Notes: IR commands can be transmitted by either: the master controller, a slave controller, or a SECU16IR module. For a master or slave controller, the zone will always be #0 and corresponds to that controller's built-in IR output. The SECU16IR module has 16 addressable outputs and the zone# corresponds to each one, allowing you to target individual IR controlled equipment when sending the commands. This gives you the ability to control similar types of equipment where the IR codes would otherwise interfere with each other. You also get individual signal amplification for each output, allowing many IR emitters to be used without loss of signal strength.

The IR system map names shown in the list box are always those of the current project. If you are triggering the transmission of an IR code for a slave controller, it's own IR codes might actually be different then the ones you see in the list box.

The Leopard II cannot transmit or receive IR and X10 simultaneously. You therefore cannot create a routine that looks for a stream of IR commands to perform X10 simultaneous dimming, or an X10 dimming stream to perform an IR function like adjusting a volume level. You need to stop sending one type of data to let it transmit the other.

THEN X10 Preset Dim (Leviton) *Switch type, House/Unit Code, Dim Level*

Description: Encodes and sends an X10 preset dim command to the Leviton switch at the specified house and unit code. The house and unit codes will be replaced by the System Map name if one has been entered. The dim level is expressed as a percentage of full brightness and can be selected either directly (using the slider) or by the contents of the specified variable.

This command is identical to the **THEN X10 Preset Dim (PCS)** command and actually uses the same control wizard screen. See the **THEN X10 Preset Dim (PCS)** command reference page for details on using this command.

THEN Transmit X10 Group (Leviton) *House Code, Group#, Set Command*

Description: Encodes and sends an X10 command to control a predefined Leviton switch group. The group is specified by the house code and the group number, as programmed in the switch at installation time. The set command can be to either have it go to its preset level, or to turn the group off.

Example:

```
0001 - IF Time of Day is > Sunset offset 0 minutes // at sunset
0002 - THEN Send House F-Group# 14, Group to PRESET // set grp 14 of h/c F to dim levels.
0003 - IF Time of Day becomes = 23:30 // at 11:30 PM
0004 - THEN Send House F-Group# 14, Group to OFF // turn that group off
```

Timing: The command is put in the X10 output queue and program execution resumes immediately.

Notes: These commands can only be used on Leviton's 16xxx and "Green Line" HC series switches. A Leviton group corresponds to a combination of a unit code and a learned dim level. A switch can belong to up to 4 groups. To use the **THEN Transmit X10 Group (Leviton)** command, you must first configure the switches to "teach" them their group numbers and dim levels. Normally this is done using a Leviton controller switch, but you can also use a

handy C-Max utility to do this too. The utility is under the “X10” utilities on the controller access screen; simply click on “Send Leviton X10” to set up and manage your Leviton groups. The use of this utility is covered in the C-Max section of this manual.

THEN Transmit Speak Easy Module#, Message#

Description: Instructs the specified Speak Easy module to play back the selected message#.

Example:

```
0001 - IF Module #2 -SECU16 Input #0 Turns ON           // if driveway's car detector trips
0002 - THEN Send Module #1 -SPEAK-EZ Audio Message #5 // announce imminent arrival
```

Timing: The command is sent to the speak easy module for playback and program execution resumes immediately. If a playback command is sent to a speak easy while a message is already playing back, the current message will be interrupted and playback of the new message will immediately begin.

Notes: The Speak Easy is an expansion module that can play back recorded messages or sounds. As explained in the module’s documentation, you may configure it to be able to record and play back many (up to 50) short messages, or a lesser number of longer messages. You can only play back messages under program control. To record messages, follow the procedure under the C-Max module utilities, described elsewhere in this manual.

THEN Transmit Ascii Message Module#, Embedded Variable#, Message#

Description: Instructs the specified module to transmit the selected ASCII message. The ASCII message can contain a formatting string allowing the specified variable#’s contents to be displayed within the message. The message number can be specified by either a constant or by the contents of a variable.

Example:

```
0001 - IF Module #1 -BOBCAT-T is > -100                // read temperature
0002 - THEN Load Data to: Variable #3                  // and capture in var #3
0003 - ELSE Load Data to: Variable #3                  // in all cases
0004 - IF Timer #5 becomes > 3600                     // once per hour
0005 - THEN Timer #5 = 1                               // restart the timer
0006 - THEN Send Module #2 -BOBCAT-A Message 6 w/ Variable #3 // and send the current temp.
```

Timing: The formatted ASCII string is placed in the serial output queue and program execution resumes immediately. For Serial Bobcats or slaves, the command is sent to the module (along with the contents of the embedded variable) and program execution resumes immediately.

Notes: ASCII messages can be transmitted by either: the master controller, a slave controller, or a Serial Bobcat. The use of an embedded variable is optional. If there is no formatting string within the ASCII message (ie: “%...””) then the embedded variable will be ignored.

A Serial Bobcat must be at release 6 or higher to support embedded variables.

Do not send ASCII messages to a controller’s serial port if that port is also being used by an external computer or device to exchange data using the ADI serial protocol. Doing so will cause a conflict and the external device will receive both types of serial messages, and could cause unpredictable behavior.

There are several formatting options available for embedded variables. These use the same format strings as the C language’s *printf* command. See the application note on formatted variables in this manual to see a complete list of the available formatting options along with usage examples.

ASCII strings are defined and loaded into your controller(s) or Serial Bobcat(s) using the C-Max controller utilities. See the manual section on using C-Max to learn how to do this.

END

Description: Terminates execution of the current program pass.

Example:

```
0001 - IF Touch Object #5, Button Leopard is pressed // if light-on button is bressed
0002 - THEN X-10 House G / Unit 3, Turn ON // turn on the light
0003 - End Program //
```

Timing: The program pass is immediately terminated and the command interpreter will begin the process of initiating a new pass.

Notes: The **END** statement is not a standard action statement, but rather a separate command type. This is because the command will execute regardless of the current true/false state (hence the absence of a *THEN* or *ELSE* prefix).

You can have more than one **END** statement in your program if you wish. Since the program interpreter executes every line in a program except those skipped over by **THEN Skip to** statements, the only way your program can get past any intermediate **END** statements is by having **Skip to** statements that cause the interpreter to not execute them. In that sense, you can think of intermediate **END** statements as a shortcut to skipping to the last line of your program.

4 C-Max User's Guide

This section of the manual describes the various utilities available in C-Max for programming and using your Leopard II. C-Max is a combination of program and screen editor, program and data loader, debugging tool, testing utility, etc. Each one of these functions will be described in detail. Although the program editor is the main screen and the first thing that you see when first start C-Max, we will begin with the **Project** sub menu since this is where you would start the creation of a new project and application program.

4.1 Projects

C-Max 2.0 introduces the concept of "projects" which aims to make file management easier by keeping the files (the ".pgm", ".tch", and any ".bmp" files for icons) for a given controller together in a directory. This allows a user to load and save all these related files together in one operation, saving time and avoiding frustration in having to locate separate files.

4.1.1 Creating a Project

From the main screen,(fig. 25), create a project by clicking on **Project → New Project**.

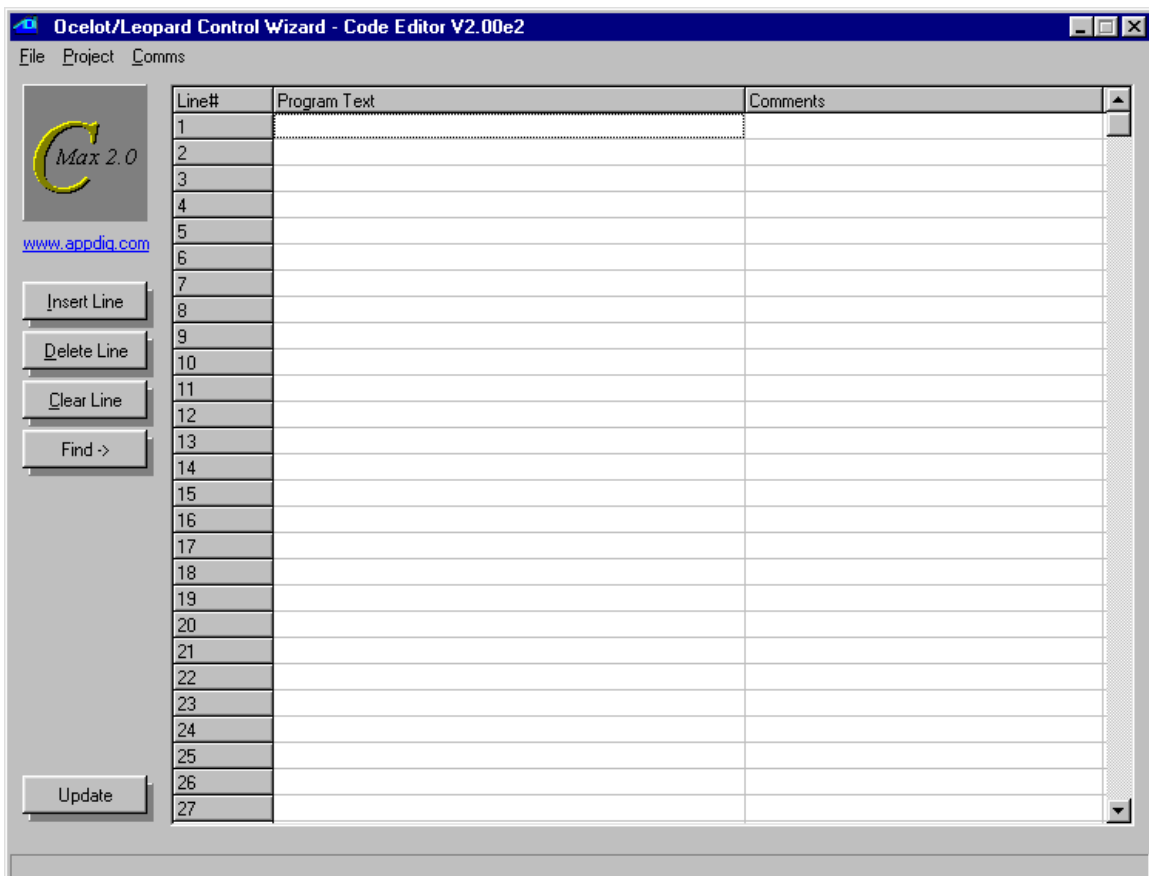


Fig. 25

You will see a file browsing window (fig 26).

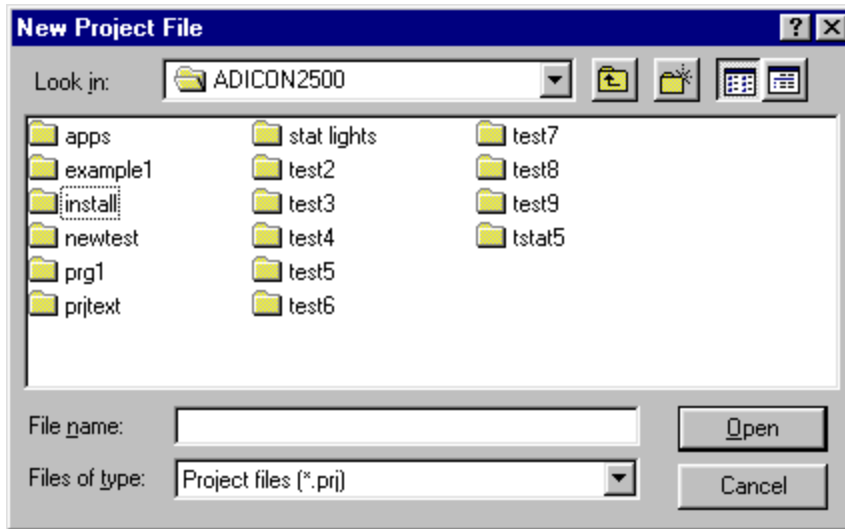


Fig. 26

Browse to a directory, or create a new one (recommended) where you want to keep the files associated with this project. It is best that you create a separate directory for each project, using the “new directory” control in the file browser (the small file folder with a sparkle). Once you have browsed to the desired directory, enter a name for your new project and click on "Open" to first save your project).

4.2 The System Map

The first thing you should do when you begin a new project is to create the System Map. The System Map allows you to identify any expansion modules or slave controllers that you may have, as well as permitting you to give your own names to most of the data items (variables, timers, X10 devices, etc). A bit of time invested in creating and maintaining it will make programming much easier, and your programs will be almost self-documenting. Start by creating a new Project as explained previously and then create/edit your System Map as follows:

Click on **Project → System Map**. You will see a new window with an expandable list starting with “Ocelot” at the top. (fig 27).

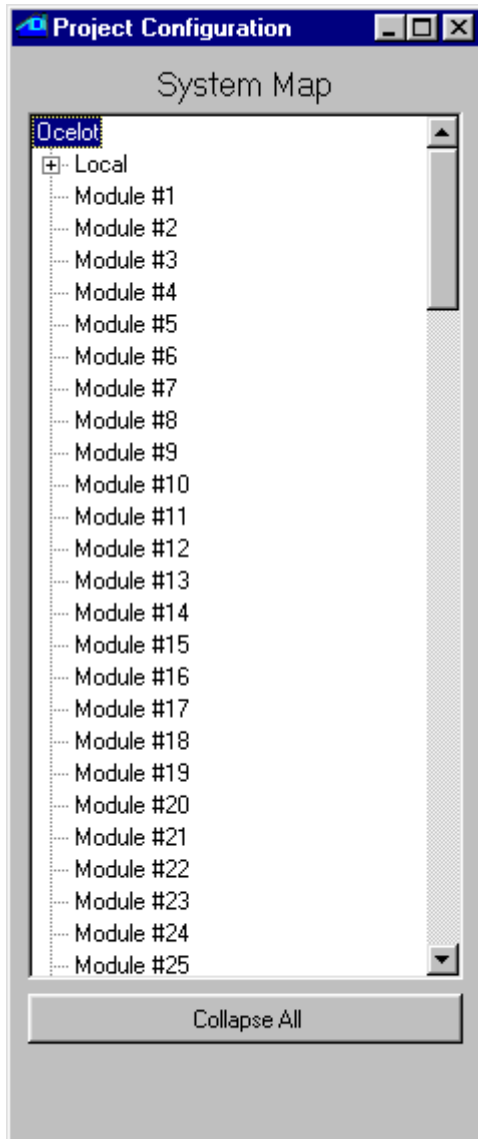


Fig. 27

If your controller is a Leopard II (or the original Leopard), right click on the word “Ocelot” and choose the “Leopard” controller type (once you have chosen Leopard, you cannot set it back to Ocelot). Then click on each module address that you have in your system and once again, right click to choose the type of module at that address. Note the Bobcat types: Bobcat-T = temperature, Bobcat-H = humidity, Bobcat-A = ASCII (serial), etc.

Now, you will notice that to the left of each module that you add (and also under the master controller on the line labeled “Local”) there is a small box with a “+” sign. Clicking on a “+” will expand that module’s list of resources, like I/O ports, variables, timers, touch buttons, etc. You can now edit each of those resource’s labels and give them meaningful names. If X-10 address B/9 is the hall light, then you can enter “Hall Light”. The process of editing resource names is similar to editing file names in Windows; click once to select the item, a second time to highlight the whole name, and optionally a third time to position the cursor within the existing name.

When you edit an item under **icons** you not only name the icon but also actually define it there. By right-clicking on a name, you will see a file browsing window allowing you to locate the bitmap file that you want to use for this icon. Once the correct file has been located, clicking on “Open” will **make a copy** of the icon file to the project directory. Keep this in mind if you want to edit the icon image later. Complete instructions on creating and using icons are given in the icons application note in section V of this manual.

Once you are done, choose **Project → Save Project** and this will save your newly created System Map in the project file. If you now start to create a program or edit an already loaded one, you will see the assigned names right in the program text. If you already had a program loaded in the editor, you can immediately update the program listing with the new names by clicking on the **Update** button in the lower left-hand corner of the program editor screen. You will also see the assigned names in the appropriate list boxes when adding/editing lines. **Save Project As** allows you to save an existing project under a new name and/or directory. **Open Project** is used to load an existing project into C-Max for editing.

Saving your project always saves all the related files for that project (the program file, touch screen file, and system map. Each of these component files will be given the same name (but with the corresponding .pgm, .tch, .prj file extensions). It is a good idea to regularly use **Save Project** as you are working on the various tasks in creating your application, such as editing the program code or creating the touch screen layout.

The **Download Project** menu selection will show a pop-up window (fig. 28). This provides a handy single-operation way to download the various project components (icons, program, touch screen definition) to the controller. Simply select the item(s) you want to download by checking the appropriate boxes and start the download by clicking on **Begin Download**.

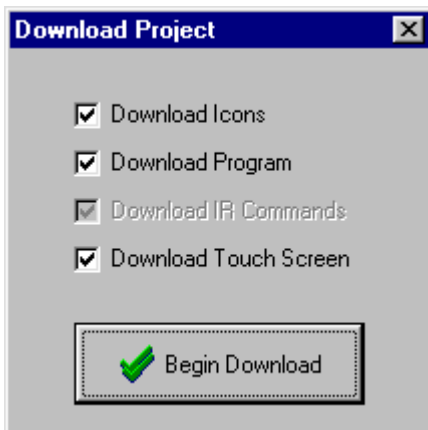


Fig. 28

4.3 The Program Editor

The program editor is the main screen that you see when you start up C-Max (fig. 25). The screen shows you 27 lines at a time but you can scroll through all 4096 possible lines of a program using the scroll bar on the right. Program lines are created and edited using the **control wizard** (fig 29), which is brought up by double clicking on any existing or blank line. In the control wizard screen, choose the instruction type first (IF, THEN, ...), then the instruction itself (Module/Point,...) and then the desired options that will appear in the bottom part of the window. Some instruction types will not display the list of options unless the expansion module type that it refers to is defined in the System Map. This is why it is best to start by defining your System Map first. Section III of this manual gives you the complete command reference guide to the options available for each instruction. You can also enter text comments for any program line by clicking the desired program line under the **Comments** column. A sample session showing the use of

the program editor and command wizard can be seen in the **Writing your first program** tutorial at the end of Section 1.

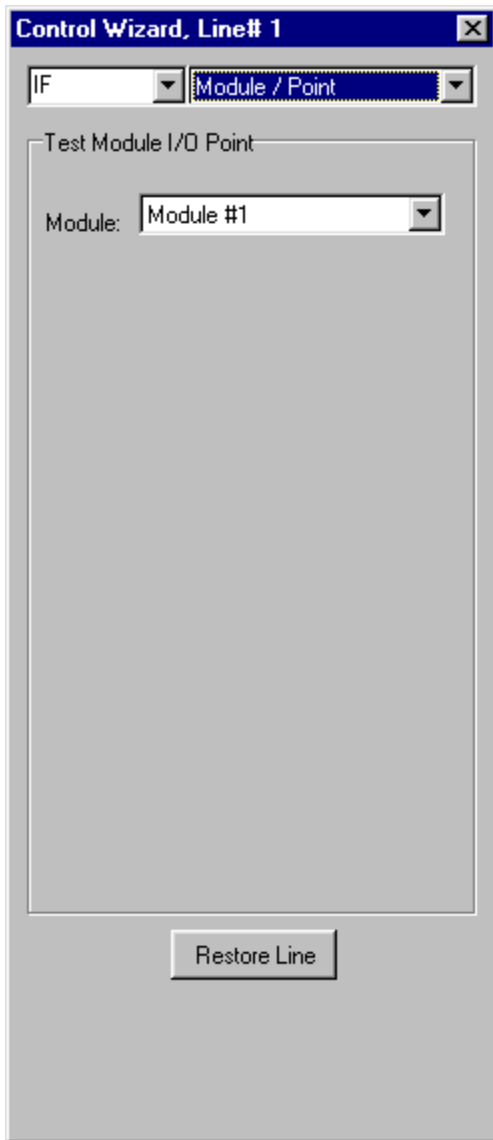


Fig. 29

4.3.1 Editing Commands

The **Insert** button on the left of the main screen allows you to insert a new program line at the currently highlighted line on the screen. The existing line and the other lines below it will all be pushed down by one line to make room for the new line. The **Delete** button does just the opposite, removing the currently highlighted line and pulling all lines below it up by one to fill the void. Doing each one of these operations causes the entire program listing to be updated and any "Skip To" offsets are recalculated to maintain correct program logic. Lastly, the **Clear Line** button removes any instructions on the highlighted line and leaves it blank. If you intend to insert or delete several lines at one time, see the discussion on cut/copy/paste below for time saving tips on doing this.

More complex editing tasks can be accomplished by right clicking your mouse over the program text area. The **Cut** control allows you to select a group of program lines to be removed from the program. To use it, begin by left clicking on the first line that you want to cut from the program and while holding down the mouse button, drag the mouse pointer to the last line that you want to cut, and release the mouse button. If the last line you want to cut is not displayed on the screen, drag your mouse above or below the program text area and the listing will automatically scroll through the lines in that direction. Once the mouse button is released, you will see the selected program lines highlighted. You can then right click your mouse over the program area and select the **Cut** command. The selected program lines will be immediately removed from the program and any program lines after the cut section will be pulled back up to fill the lines vacated by the cut. The lines that were cut are now held in memory for possible pasting. You can do one of two things at this point:

- 1- Continue editing your program and ignore the cut lines. This is equivalent to having deleted the lines one by one and is actually a good time saving shortcut to do this because every time the **Delete** button is used, C-Max needs to update the entire listing. By cutting these lines as a group, only one update will be done.
- 2- **Paste** the lines somewhere else. Cutting and pasting amounts to moving the lines from one location to another. To paste the lines, click the first line where you want to place the code segment and then do a right click and click on **Paste**. Any lines that were at that point or below will be pushed downwards to make room for the new lines.

The third program editing command in the right click menu is **Copy**. This works just like the **Cut** command except that the original program lines are not removed; they are only copied to memory for a subsequent **Paste** operation. This is used for making more than one copy of a group of program lines. The **Copy** command is also a handy time saver if you want to insert several new lines within your program. Suppose that you want to insert about 10 new lines in the middle of your existing program: Use the right hand scroll bar to quickly go past the end of your program and use **Copy** to quickly “grab” about 12 to 15 blank lines, then go to the line where you want to insert the new lines and **Paste** the blank lines there. C-Max will only need one screen update to reflect the insertion of all the new blank lines. Program the lines as you want them and then remove any superfluous blank lines by using **Cut** as described above.

Copying and pasting between different projects/programs is also possible by using the **Save Code Snippet to Disk** and **Load Code Snippet from Disk** commands, also in the right click menu. These are used just like the **Copy** and **Paste** commands except that instead of using memory to hold the selected program lines, they use disk files. Using these selections will open a file browser window allowing you to create or open the file where you want to store or load the file. Snippet files have a .snp file extension.

The **Find >** button (on the left of the editor screen) can be used to locate program lines by searching for text either in the program code itself or in the **comments** column. Use it as follows: Right click on the “Find” button and the shadow under the button will expand to a text field where you can enter the text you are looking. Enter the text string (it is not case sensitive) to be searched and also check the “include comments” box if you want to search the comments column as well. Then click on the “Find” button and the next line containing the search string will be highlighted every time the button is pressed. Note that the search will always begin from the currently selected program line (just click once on the desired start program line) downwards. You must re-select the start line if you want to do a new search. Finally, do another right click on the “Find” button to re-hide the search string.

4.4 Saving/Loading/Printing Programs

Under the **File** pull down menu of the program editor, you will find the following selections:

New Program – This will completely erase all program lines currently in the editor.

Open Program – This selection will open a file-browsing menu allowing you to load an existing program file saved on disk. This can be useful to create new projects from existing programs that were created with a previous version of C-Max, or that was created for another project. This will only load a program code file (.pgm) and will not affect other items like the System Map or the touch screen file.

Save Program As – Saves the current program code in the editor to a file of your choice, using a file browser to allow you to choose the location and name of the program file. A warning window will appear if you are about to overwrite an existing program file. Like the **Open Program** command above, Saving a program file this way keeps it independent from any project you might have opened at this time; only the .pgm component file will be saved and with the specified name.

Print – Prints a listing of your program code. A printer control window will be displayed, allowing you to specify the printer and printing options as offered by your printer's Windows driver.

Print to File – This option allows you to produce a program listing that will be saved in a text file. A file browsing window allows you to choose the location and name of the .txt file that will be produced. The listing will be an ordinary ASCII text file that can be opened with utilities like Notepad or any word processing program. All the program examples shown in this manual were created using this utility.

4.5 The Touch Screen Editor

The touch screen editor is used to define the controls, text, and general appearance of what you will see on the screen of your Leopard II. A bit of extra time spent in optimizing the “look” of your screens will certainly pay off in general user satisfaction, but remember that you can always go back to your project and optimize it later if you wish; and a certain amount of trial and error is often inevitable in getting the appearance you really want. The touch screen editor is accessed from the main C-Max screen by clicking on **Project -> Touch Screen Setup**. If you get an error message saying that your controller type must be defined as a Leopard, go to the System Map and make sure you define it as “Leopard” there first.

Once in the touch screen editor, you will see an editing screen like the one shown in fig 30:

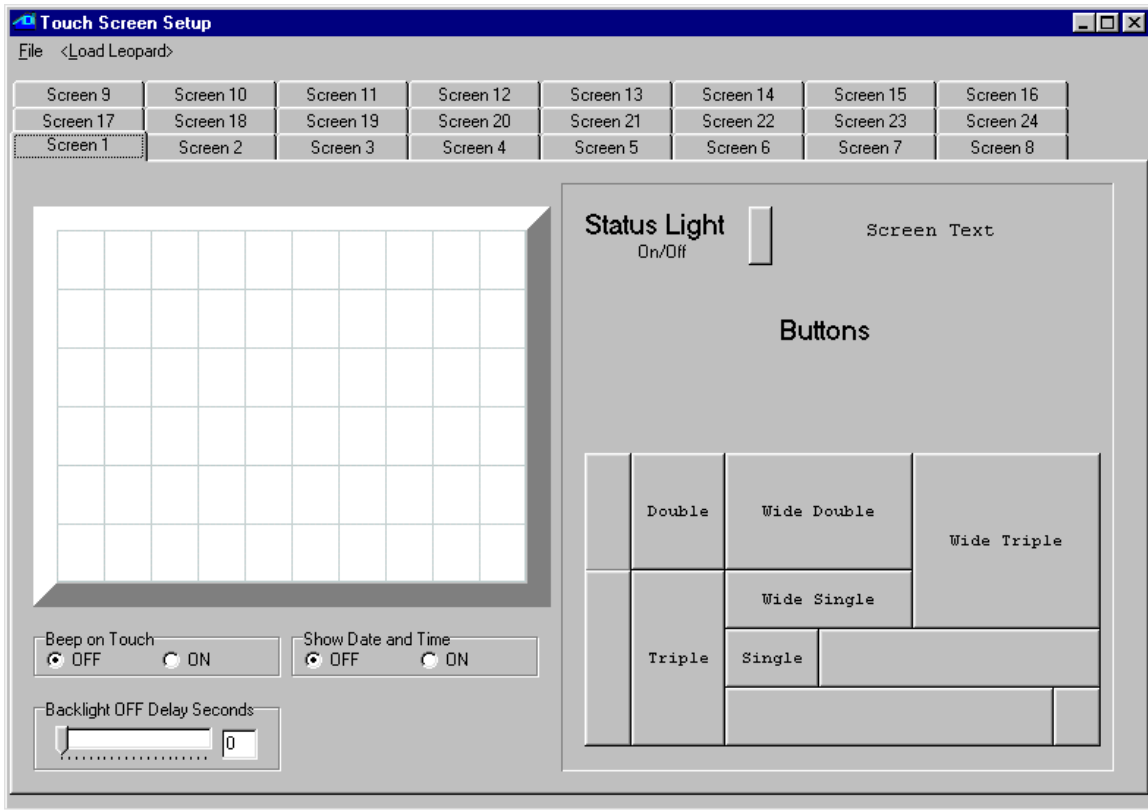


Fig. 30

Looking at the editor layout, we see:

On the left hand side, the screen image that corresponds to your Leopard II screen. It is divided up into individual “touch squares” (which are not truly “square”...). There are 10 squares horizontally and 6 vertically, for a total of 60 squares. Each touch square represents the smallest touch area that the screen controller can distinguish.

Along the top, 24 tabs representing the 24 individual screens that you can define. In the controller, switching screens is done under program control by setting variable #63 to the screen number that you want. Click on the tabs to edit the screen of your choice.

On the right hand side, you have the three different types of objects that you can define on the screen. These are:

- 1- **Status lights.** A status light object indicates the On/Off status of an associated device; either an X10 address or an I/O point.
- 2- **Screen Text.** The screen text object can be used to display simple text headings and labels, or to display text along with an embedded variable. This is handy for displaying variable analog information like temperatures, times, levels, etc. The variable can be formatted to display in a variety of formats using a formatting string. See the variable formatting application note in section 5 of this manual for instructions on doing this.
- 3- **Touch Buttons.** The “jigsaw puzzle” of rectangles in the lower right of the screen is a selection of the various button sizes and shapes that you can use on a Leopard screen.

Finally, under the screen layout area, there are three programmable options:

1- **Beep on Touch.** Setting this option to ON will cause the Leopard II to emit a short, low volume beep every time the screen is touched. This can be useful to obtain “tactile feedback” when touching an otherwise rigid screen surface.

2- **Show Date and Time.** Selecting this option displays the date and time in the upper right hand corner of the screen, regardless of the screen being displayed. If you use this, make sure you leave room for it in your screen layouts.

3- **Backlight OFF Delay Seconds.** Allows you to set the number of seconds after last touching the screen before the backlighting is turned off. You will usually want to set this to a value long enough to avoid having the backlight turn off while you’re busy pressing buttons or observing displayed information, but short enough to avoid having it stay lighted needlessly, which could cause the CCF tube to age prematurely. Commonly used settings range from 10 to 30 seconds. You can select the value either by using the slider or by manually entering the value in the edit box. Setting this value to 0 will cause the backlight to stay on indefinitely.

With the editor screen now explained, let’s see how it is used. You can see a sample of a screen editing session in the **Writing your first program** tutorial at the end of Section 1.

4.5.1 Adding and Editing Buttons.

To add a button to a screen, first make sure the tab at the top is displaying the right screen number. Click once on the desired button size in the right hand part of the editor screen. A new instance of that button size will be created in the screen layout area, as shown in figure 31:

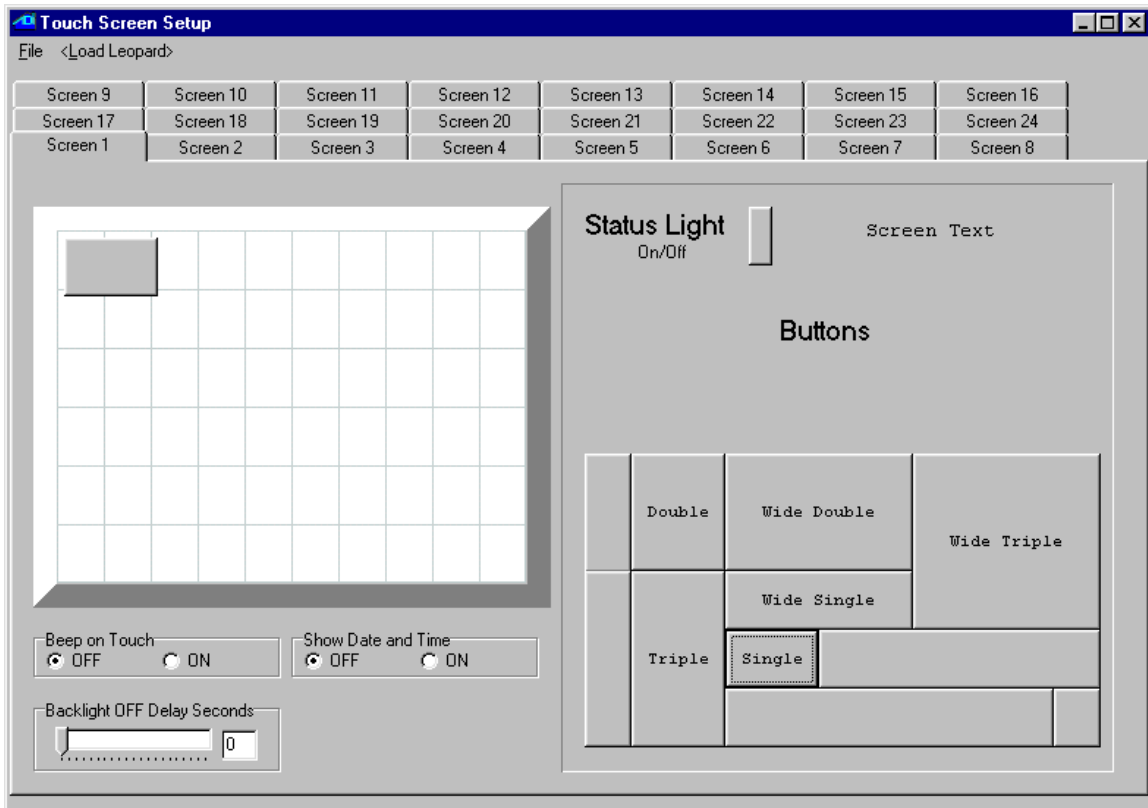


Fig. 31

Next, click on the new button and drag it to the location where you want it, being careful to align it perfectly with the touch square grid lines. Note that when you add a new button as shown in fig 31, it is purposely not aligned with the grid lines. This is to help you avoid confusing the new button with an existing button at or near the upper left corner, since you would not normally place a button this way. With the button dragged to its desired location, position the mouse cursor over it and then right click to reveal a small two-choice menu. You can either choose **Delete Button** to erase the button, or **Enter Caption** to edit the button's caption and other properties. Selecting **Enter Caption** will display the following window (fig 32):

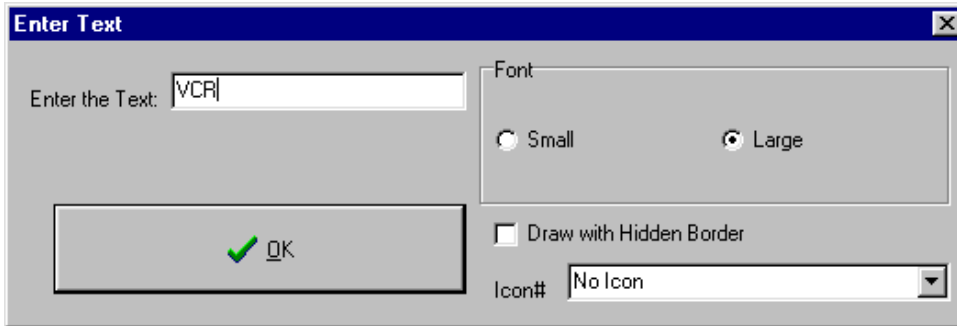


Fig. 32

You can now enter the caption you want on the button in the text box and select one of two available font sizes. The **Icon#** list box allows you to select an icon (as defined and named in the System Map) to be displayed on the button. Displaying an icon will overwrite any text caption. The **Draw with Hidden Border** check box, when checked, will cause the button's image (outline) to be invisible, allowing the icon itself to be the button's image. If you check the **Draw with Hidden Border** box and do not assign an icon to a button, you will have an invisible but fully functional button. Creating and using icons is covered in the icons application note, in section 5 of this manual.

Once your button properties have been entered, click on **OK** to close the window and see the new caption displayed in the screen layout area (fig. 33) Note: icon images are not displayed in the editor.

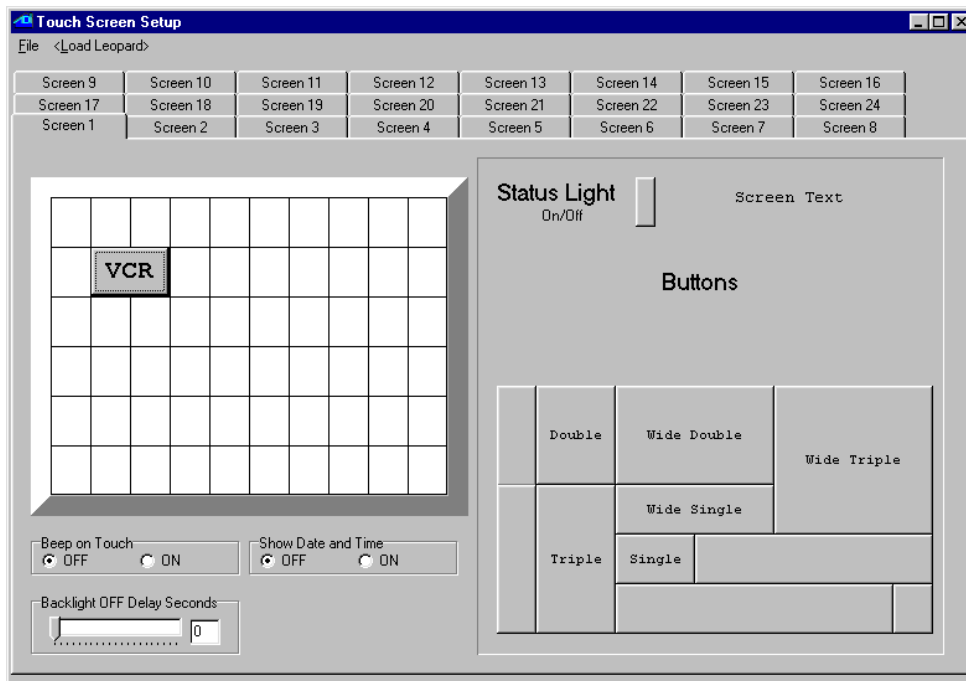


Fig. 33

In order to code instructions for that button in your C-Max program, you need to know its object number. You can see this number by moving your mouse cursor over the button (do not click any mouse buttons). A small point-help window will reveal the object#, the “hidden border” setting (1 = checked, 0 = unchecked) and the icon# associated with this button. Make a note of the button’s object number for coding your C-Max program.

4.5.2 Adding and Editing Status Lights.

To add a status light to a screen, first make sure the tab at the top is displaying the right screen number. Click once on the status light in the right hand part of the editor screen. A new instance of a status light will be created in the screen layout area, as shown in figure 34:

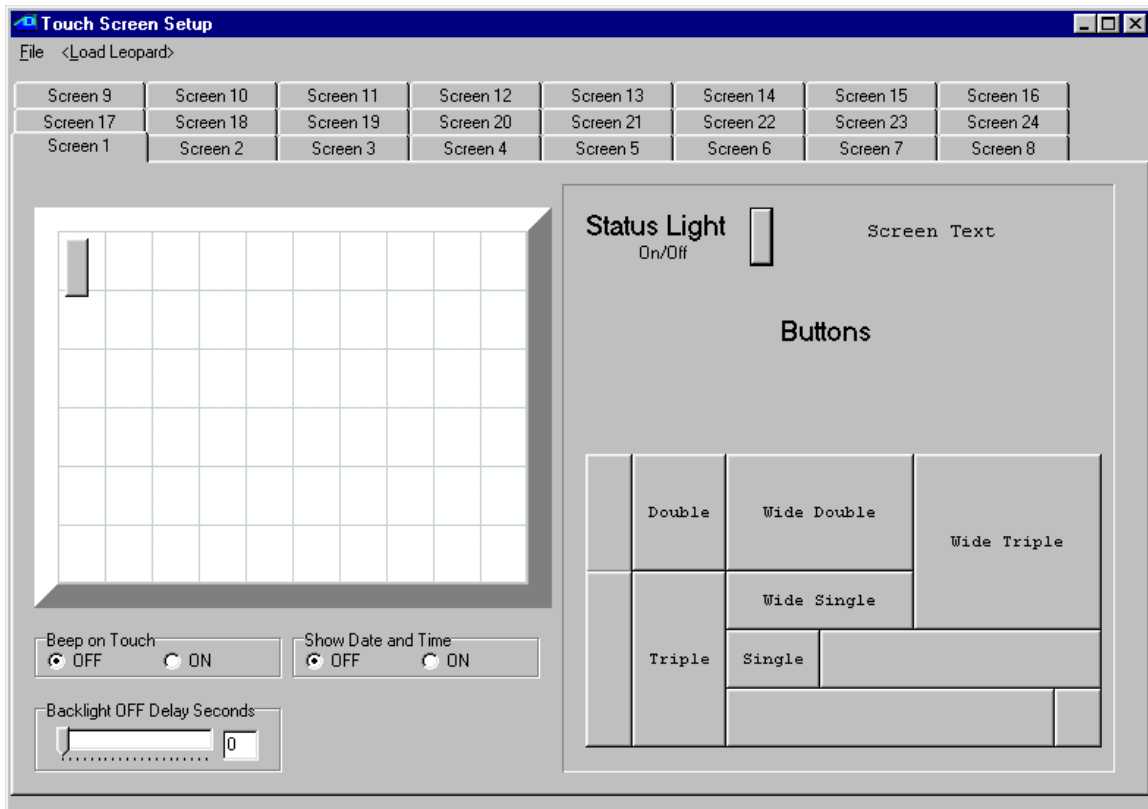


Fig. 34

Next, click on the new status light and drag it to the location where you want it. Contrary to touch buttons, status lights do not need to be aligned with the touch square grid lines. Note that when you add a new status light as shown in fig 34, it is purposely not aligned with the grid lines. This is to help you avoid confusing the new status light with an existing one at or near the upper left corner, since you would not usually place a status light this way. With the status light dragged to its desired location, position the mouse cursor over it and then right click to reveal a small three-choice menu. You can either choose **Delete Light** to erase the status light, or one of **Attach X-10** or **Attach I/O**. to associate the light with either an X10 address or an I/O module point.

If you select **Attach X-10**, you will see a small window (fig. 35) giving you access to the System Map names of all X10 devices.

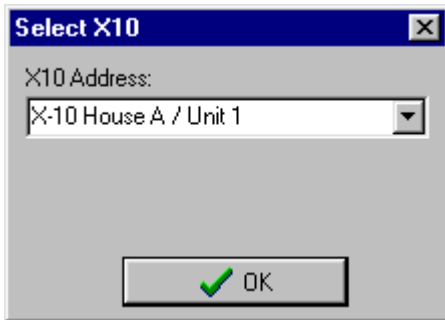


Fig. 35

Simply choose the desired X10 device and click on “OK”. Now, when that X10 device is ON (according to the controller’s internal status table) the light will also show “ON”, and show “OFF” when it’s OFF.

If you select **Attach I/O**, you will see a small window (fig. 36) giving you access to the System Map names of all I/O module devices.

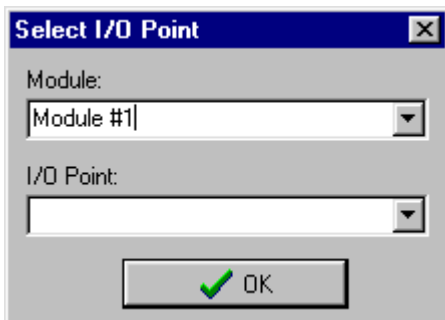


Fig. 36

Start by selecting the desired module (these have to be defined in the System Map first, see **Projects**), then the I/O point, and click on OK. Note that both input and output I/O points can be selected. Like for the X10 devices, the light will display “ON” or “OFF” according to that I/O point’s status in the controller’s internal status table.

4.5.3 Adding and Editing Screen Text

To add a screen text object to a screen, first make sure the tab at the top is displaying the right screen number. Click once on the “Screen Text” phrase in the right hand part of the editor screen. A new instance of a text object will be created in the screen layout area, as shown in figure 37

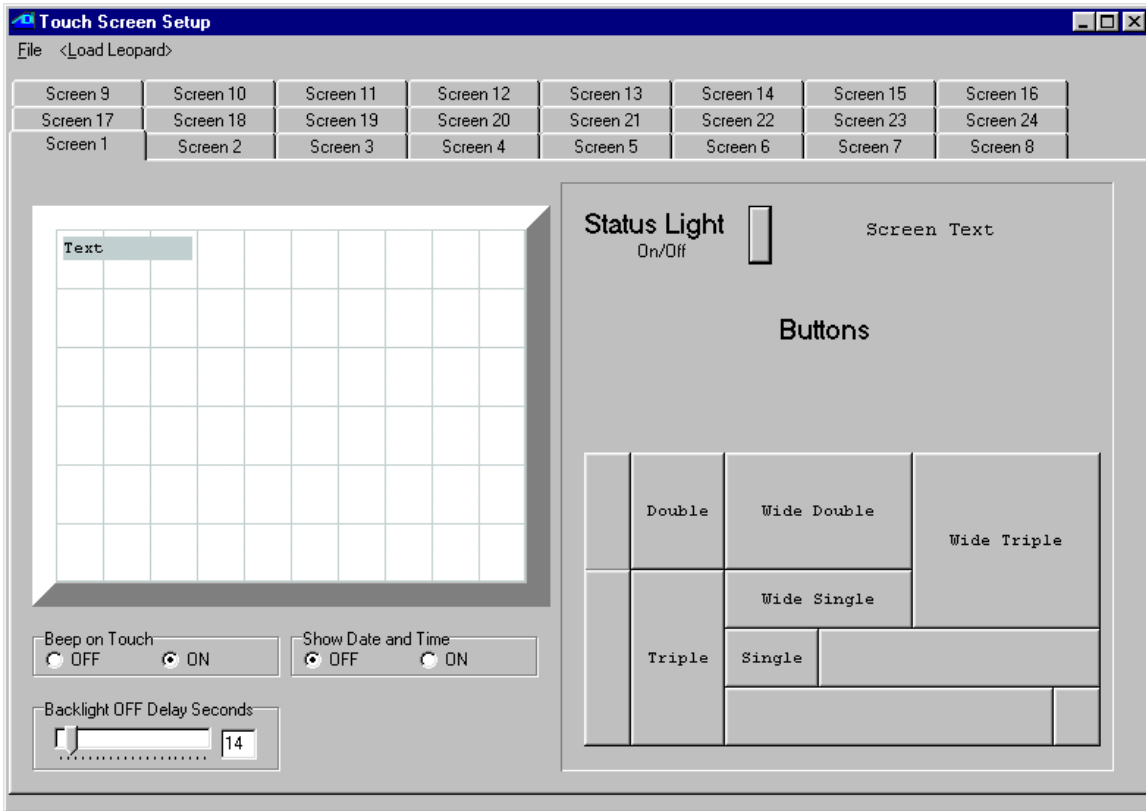


Fig. 37

Click on the new text object and drag it to the location where you want it. Contrary to touch buttons, text objects do not need to be aligned with the touch square grid lines. With the text object dragged to its desired location, position the mouse cursor over it and then right click to reveal a small two-choice menu. You can either choose **Delete Text** to erase the text object, or **Enter Text** to enter/edit the text and optional variable formatting string.

If you select **Enter Text**, you will see an editing window (fig. 38):

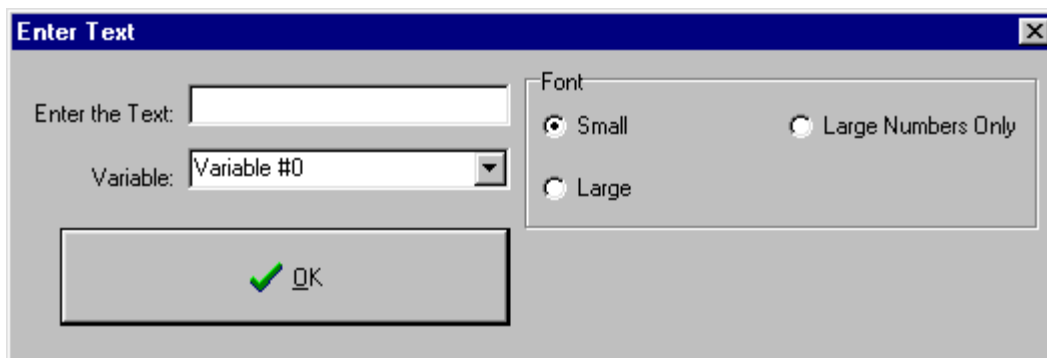


Fig. 38

Enter the text that you want to display in the edit box. You can also enter a variable formatting string to embedded a variable (selected with the list box underneath the text box). For example, you could enter the text as “Outside Temp: %3d F.” and select Variable #3. If your controller program captures the outside temperature in variable #3 and the temperature is 75 degrees, your screen will display “Outside Temp: 75 F.”. There are several formatting options available for displaying variables. You can learn how to use these various options by reading the variable formatting application note in section 5 of this manual.

The right hand side shows that you can display text in two font sizes, small and large. There is also a very large “Numbers Only” font that can only be used to display numbers or formatted variables. This font simulates the 7 segment “LED readout” style often seen on digital clocks and appliances. You can see examples of this type of display in the application notes.

Once you have finished entering your text, click on “OK” and the screen layout window will show the updated text object, resizing it as necessary. You may want to reposition it once the text has been entered.

4.5.4 Managing Touch Screen Files:

When you have finished creating your screen layout, you can save your work and/or download it into the controller. You have two approaches you can use: saving and downloading the screen file by itself, or doing these operations as part of a project.

4.5.4.1 Individual Screen Saving, Loading;

If you click on the **File** pull down menu at the top left of the touch screen editor, you will see three menu choices:

4.5.4.2 New Touch Screen File

Erases all objects on all screens in the touch screen editor. Used to start a new layout.

4.5.4.3 Open Touch Screen File

Used to load an existing touch screen file that can come from another project or a version of C-Max prior to version 2.00 (when projects did not exist). Selecting this will open a file browsing menu and will display all files in the current directory that have the .tch file extension.

4.5.4.4 Save Touch Screen Files As

Saves the touch screen file currently in the editor. Selecting this will open a file browsing menu and will display all files in the current directory that have the .tch file extension. You can either select an existing disk file to overwrite or give your file a new name. Saving a touch screen file this way keeps it independent from any project you might have opened at this time; only the .tch component file will be saved and with the specified name.

Clicking on <**Load Leopard**> will immediately open the controller access window and download the screen file to the Leopard II. This can be used as a quick shortcut to downloading the screen file when all you really did was modify the screen layout on an otherwise running controller. The other way to download the screen file is by downloading the project, as described in the **Projects** chapter.

4.5.4.5 Project Screen Saving, Loading

The other way to save and download your screen file is by doing it as part of your project management. To use this method, simply close the touch screen editor when you are finished editing your screen file (you will get a warning that your screen file has not been saved, but that's ok). Then Use the **Project** pull down menu from the main screen to do **Save Project** to save your touch screen file along with the other project files, and **Download Project** to download the desired components of your project to the Leopard II.

4.6 C-Max Utilities

The C-Max utilities are used for a variety of functions such as learning infrared codes, debugging (monitoring) variables and timers, editing ASCII strings, etc. Taking the time to learn their capabilities will allow you to save a lot of time when testing a new program or just monitoring activity when working on your home automation equipment. The main utilities menu is accessed by clicking (from the main editor screen) on **Comms** → **Attach to Controller**. You will see the **Controller Access** screen (fig. 39):

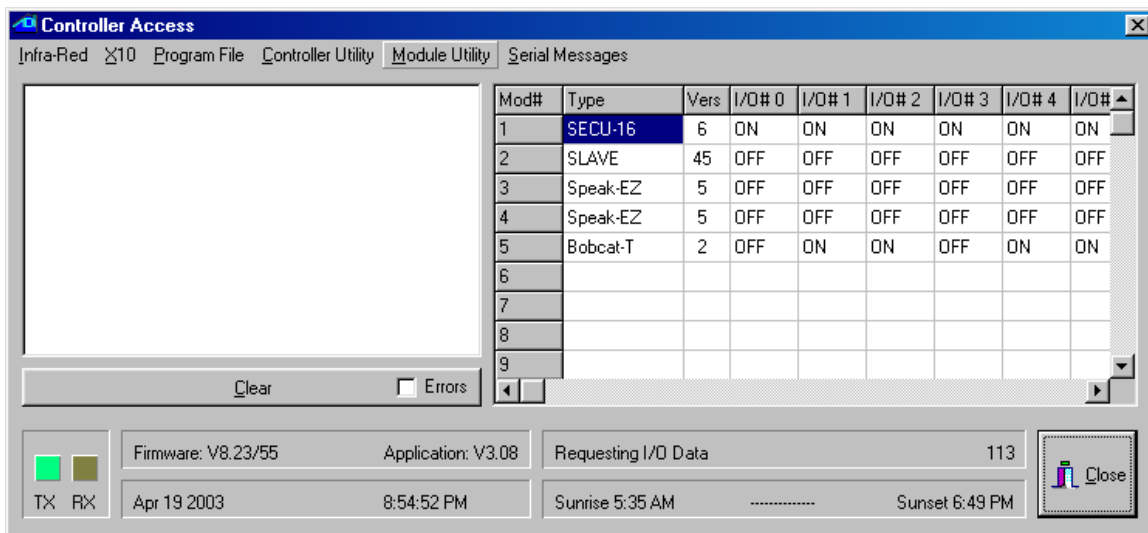


Fig. 39

The controller access screen is laid out as follows: The **activity status window** occupies the left part of the screen. Any activity you perform such as downloading a program, modifying a parameter, etc. will be listed here. New activity messages will always appear at the top, pushing down any previous messages. If the window gets full, a scroll bar will appear to its right, allowing you to view previous messages. The **Errors** check box will cause only error messages to appear instead of all activity, useful when debugging particular problems.

On the right, you have the **expansion module status** window. For each expansion module that you have, you will see its **module type** (eg. SECU16, SLAVE, etc.) and the **firmware version** of that module. Following that are the ON/OFF status indication of the 16 possible **I/O points** of that module, and finally the **Data** for that module. Note that not all columns apply to all module types. For example, the I/O points only apply to the SECU16, SECU16I and RLY8XA modules. Similarly, the Data value is only valid for Bobcat modules.

Along the bottom, you have the **Rx** and **Tx** indicator “lights” showing serial activity between C-Max and the controller. Next is the **Firmware** and **Application** version numbers. The firmware version is dependent upon the manufacturing date of the controller and other factors. This value will not change. The application version number tells you the version of the current **executive** version. The executive version refers to the part of the code in the controller that interprets the commands you use in your C-Max program, and new versions of C-Max often include a new matching executive as well. See the chapter on **Reloading the executive** to learn more about this important component of your controller’s software.

Below the firmware and application versions is your computer’s date and time. For certain operations like loading a program or touch screen file, this line turns into a progress bar. To the right of these are two more information lines: The top line shows what the serial link to the controller is being used for at any given moment. The bottom line shows the sunrise and sunset times for the PC’s current date, for the defined geographical location. The geographical location is entered as longitude and latitude in the **Comms setup** screen during the C-Max installation process (see section 1 of this manual).

Along the top, you have the pull down menus used to access each type of utility. We will now examine each one of these utilities in detail.

4.6.1 Infra-Red

The infra-red (IR) utilities are used to learn, transmit, and otherwise manage IR codes in your controller. The Leopard II can learn up to 1024 IR codes. Clicking on **Infra-Red** reveals a pull down menu with several choices:

4.6.1.1 Learn Infra-Red Command

Learning Infra-Red Command is used to “teach” a new IR command to your controller. Selecting this function opens a small window (fig. 40):

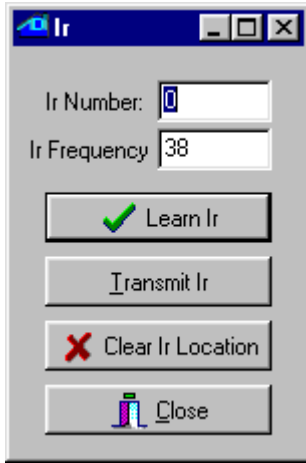


Fig. 40



Fig. 41

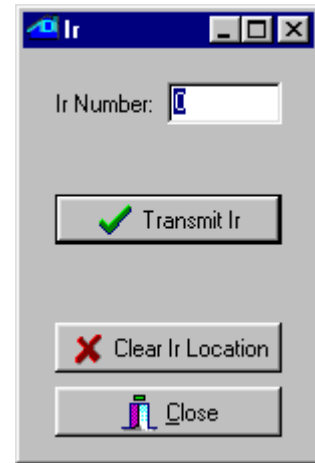


Fig. 42

To learn a new IR command, enter the **Ir Number** where you want to store the new code, and the **Ir Frequency** that you want to associate with this code. The IR frequency is used by the controller when it needs to “play back” the IR code to the device that this code normally controls. The frequency is the carrier frequency that the controlled device (TV, audio component, etc.) looks for in an attempt to reject IR noise or interference. Having the correct frequency is important for reliable recognition of IR codes being sent from your controller to the controlled device. If you don’t know the specific frequency for the brand of device that you have, use 38 or 40 kHz as a starting point, as these frequencies are quite commonly used.

With the code number and frequency entered, click on **Learn Ir**. You will see a small window (fig 41) prompting you to send the desired IR code to be learned. Point your IR device (TV remote, etc.) to the IR receiver of your Leopard II (the small square just below the bottom-center of the screen) and press the button you want to learn for a fraction of a second. The small red window should disappear and you will be back to the window in fig. 40. The Leopard II’s screen backlight will turn on when it receives an IR code.

Choosing the right IR code numbers is important: You can use IR codes in two ways with your Leopard II: They may be learned to be reproduced later by the Leopard II, for the purpose of controlling equipment under C-Max program control. IR codes can also be recognized by the Leopard II itself as input events, like button presses, etc. This gives you one more way to interact with your controller and have it respond to the IR codes to control devices in your home; be it X10, expansion modules, etc. If you plan on learning certain IR codes for the purpose of having the Leopard II recognize them as input commands later, store these in the lower numbered IR code locations. This is because whenever the controller receives an IR command, it must search through its list of learned IR codes to look for a match, which takes time and resources. To keep the search time reasonable, there is a controller parameter (parameter #20) which allows you to specify the maximum code location number to try before giving up looking for a match. By default this parameter is set to 80. If you need to recognize more than 80 codes, then adjust parameter 20 to the appropriate higher value. Conversely, if you do not need to match that many codes, then set parameter 20 to a lower number to speed up the processing of unrecognized codes. Note: Do not use code location #0 if the code will need to be recognized by the Leopard II.

Make sure you track which code is stored in which location. Using meaningful names for each IR code in the System Map should make this quite easy. If the same code is learned in two different locations, the one in the lower numbered location will be the one matched first when that code is received.

Once a code is learned, you can test it immediately if you wish by clicking on **Transmit Ir** (provided that an IR emitter has been set up previously). The learned code will be sent to the IR emitter immediately.

You can erase a learned code using the **Clear Ir Location** button.

4.6.1.2 Transmit Infra-Red Command

Transmit Infra-Red Command is used when you only want to transmit an already learned IR command. The window that will appear (fig. 42) allows you to enter the code number that you want to transmit, then click on **Transmit Ir**. You can also erase a code location by using the **Clear Ir Location** button.

4.6.1.3 Transmit Remote Infra-Red Command

Transmit Remote Infra-Red Command is like the previous command except that you can use it to transmit an IR code using either a SECU16IR module or a slave controller (fig 43). If you are transmitting an IR code with a slave controller, note that the code number specified will be the one stored in *that controller's* memory, not the code stored in the master.

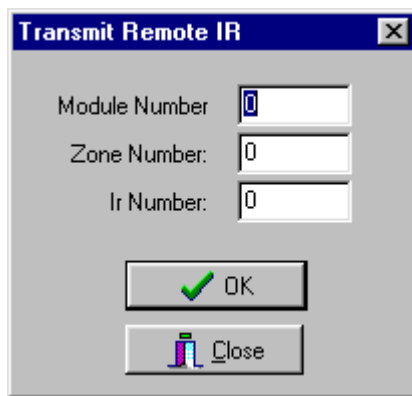


Fig. 43

To transmit a remote IR command, enter the module number (Adnet address) and the zone number, as well as the IR code number. The zone number is only used with the SECU16IR expansion module. This module has 16 addressable outputs that can be selected by the zone number.

4.6.1.4 Download Infra-Red File to Controller

Download Infra-Red File to Controller is used to send learned IR commands that are stored in a disk file to your controller. If you have a file containing IR codes (“.lir” file extension) that comes from another controller or a from a previous save, this is how you load them in. Clicking on this menu choice will open a file browsing menu, allowing you to locate the IR codes file that you want and download its contents into the controller.

4.6.1.5 Upload Infra-Red File from Controller

Upload Infra-Red File from Controller does the opposite operation, allowing you to store your learned IR codes to a disk file. This is useful for backing up all the IR codes that you learned manually, and can be used to load into another controller or even sent to another user as a disk file. Once again, a file browsing window will appear, allowing you to select a name and location for your IR codes file (the file will have a “.lir” extension).

4.6.2 X10

The X10 pull down menu has the following three choices:

4.6.2.1 Send X10.

This utility allows you to send raw X10 commands, similar to the **Transmit Single X10** command in C-Max. Selecting this command will open a window (fig 44):

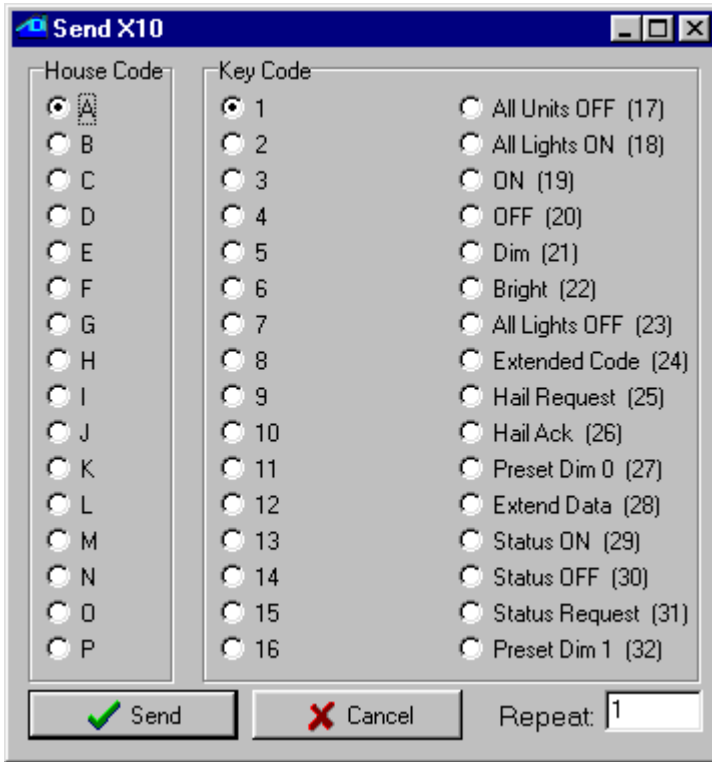


Fig. 44

To send X10 commands, select the desired house code in the **House Code** frame, then the **Key Code** which will be either a unit code or a command, then click on **Send** to transmit it. The **Repeat** edit box can optionally be used to send the command more than once. Since this is raw X10, you will need to send two separate commands if you want to send a standard X10 command pair to do something like turn on a light. For example, if you want to turn on the light at address B/5, you would begin by selecting **B** and **5**, click on **Send**, then leave **B** selected and choose **ON (19)** in the **Key Code** frame and click on **Send** again. This utility is not only useful as a general tool to send X10 commands during program testing, but also handy for setting up certain types of programmable light switches that require individual X10 commands to program them. If the instructions for setting up the switch say that you need an X10 brand Maxi-Controller for programming, then you can use this utility instead.

4.6.2.2 Monitor X10

This menu selection will open an X10 status and activity window (fig 45):

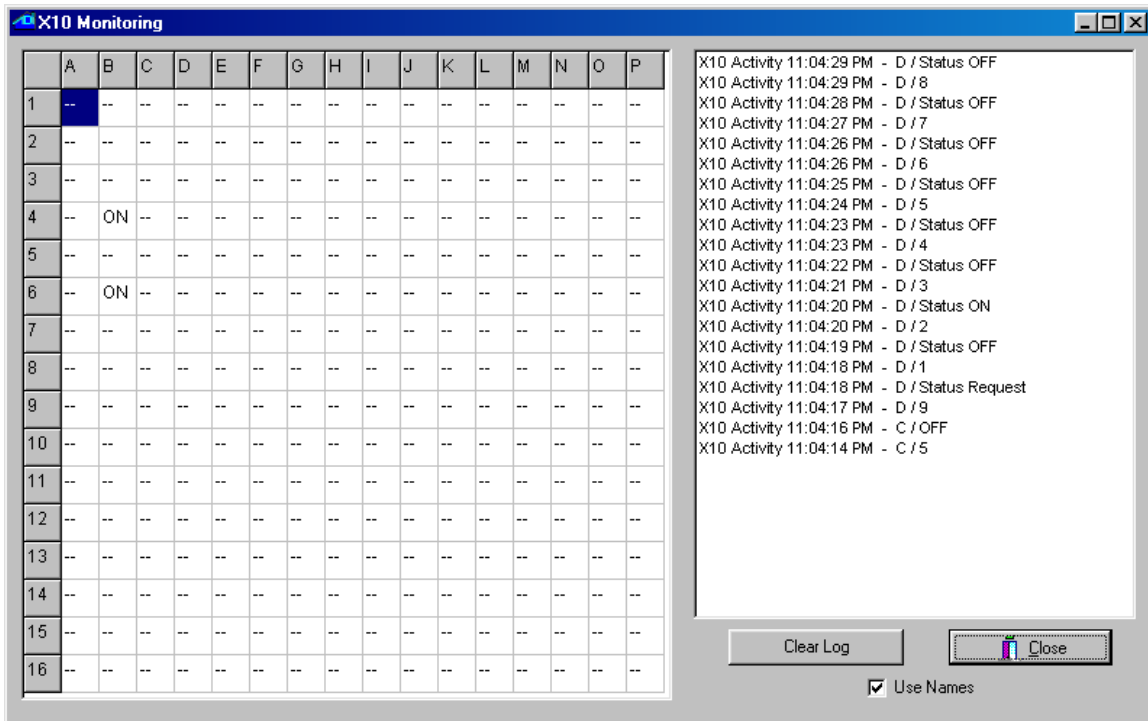


Fig 45

The left side of the window shows you the controller’s internal X10 status table. This is the table used by the **IF X10 Status/Cmnd Pair** C-Max instruction for the **IS ON/OFF** and **TURNS ON/OFF** options. The ON or OFF status indicated for each X10 address is based on the X10 commands that the Leopard II “sees” on the power line, including the ones it sends itself. Because of the nature of X10 devices, this table is not foolproof. Many X10 devices have “local control”, meaning that they can be turned on and off at the module, without sending any X10 status information over the power line to indicate the change. Also, a module (like a light switch or lamp module) may have been turned ON but then dimmed, maybe all the way down to zero, but will still be seen as being ON because no OFF command was ever sent.

The table will be updated whenever an **ON** or **OFF** command for an X10 address is received, and also by any **Status ON** or **Status OFF** transmission for an X10 address, in response to a **Status REQUEST**, whether that command is sent by the Leopard II or any other X10 device.

The right side of the window shows X10 events as they happen. Any new messages will be added to the top of the screen and push previous messages down. A scroll bar will appear on the right side of the window if the screen gets full. X10 commands will be shown in their raw form. You can choose to see commands either by their numeric value only (eg. An A/ON command will appear as “A/19”) or you can check the **Use Names** box to see the actual command name instead (e.g. “A/ON”). Use the **Clear Log** button to clear the window of all messages.

4.6.2.3 Send Leviton X10.

This third X10 utility allows you to both send Leviton X10 “group” commands and also set up (configure) your switches in groups. Selecting this utility will open a new window (fig 46):

Leviton modules, (16xxx and “Green Line” HC series) support group commands. These modules can be set to learn a group command. Every module in that group will brighten or dim to a preset level when a command is issued. Every module in the group must have the same house code. If you want each module in a group to be at different light levels, then each module must have a unique key code.

Sixty-four unique groups can be created. Each module can belong to 4 different groups. For example one light module can be in groups M1, M3, M7, and M9, while another module can be in groups M1, M25, M33, and M60. When a group M1 command is issued, both modules will respond. When an M3, M7 or M9 group command is issued, only the first module will respond. When a group command M25, M33 or M60 is issued, only the second module will respond.

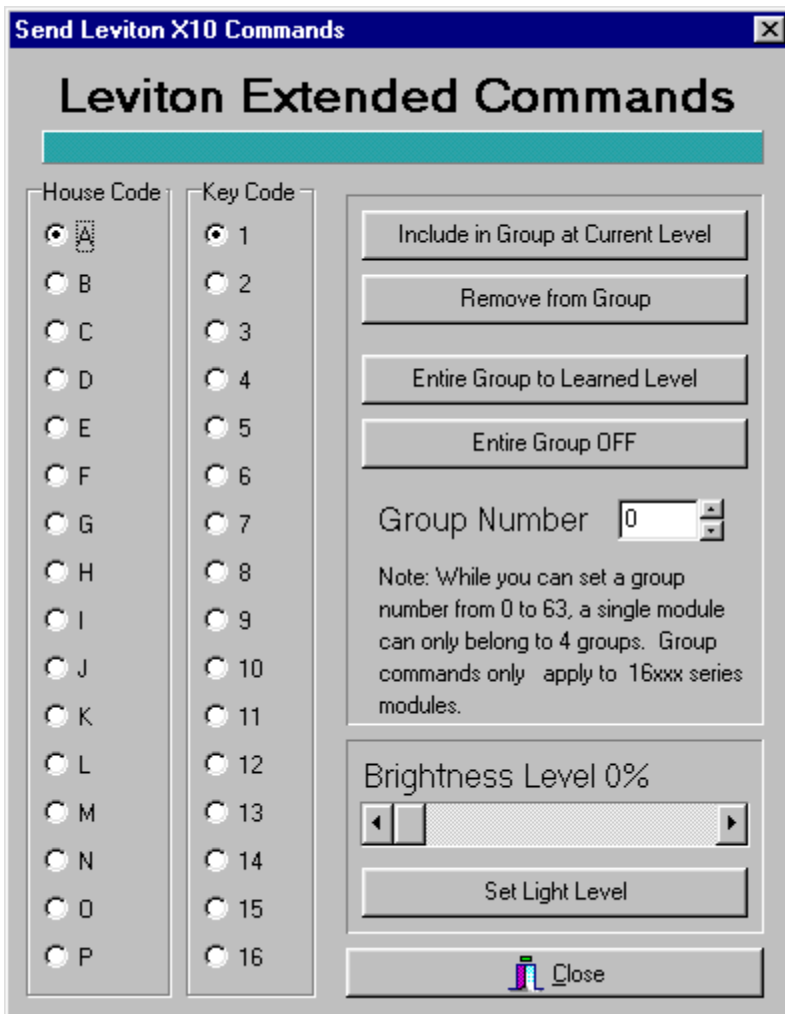


Fig. 46

4.6.2.3.1 To learn a group command:

- 1) Click the radio button to the left of the **House Code** you want to train with your left mouse button.
- 2) Click the radio button to the left of **Key Code** to select the Leviton module you want to add to the group.
- 3) Select the **Group Number** by either typing in the group number to the right of **Group Number** or click the Up and Down arrows to the right of the edit box until the desired group number is displayed.
- 4) Move the scroll bar on the lower right corner until the desired light level is displayed. You can either click the arrows to the Left (Dim) or Right (Bright) with your left mouse button, or directly drag the slider left (Dim) or right (Bright) until the desired brightness level is displayed.
- 5) Click **Set Light Level** with your left mouse button. The Leviton module will respond to that light level.
- 6) Click **Include in Group at Current Level** with your left mouse button. The Leviton module is now part of that learned group.

Repeat steps 1-6 until each Leviton module has learned its group(s).

4.6.2.3.2 Setting a Group to Their Learned Levels

- 1) Click the radio button to the left of the **House Code** you want to address with your left mouse button.
- 2) Select the **Group Number** by either typing in the group number to the right of **Group Number** or click the Up and Down arrows to the right of the edit box until the desired group number is displayed.
- 3) Click **Entire Group to Learned Level** with your left mouse button. Every light in that learned group will respond to their learned levels.

In a C-Max program, you can use the **THEN Transmit X10 Group (Leviton)** command to accomplish the same task under program control.

4.6.2.3.3 Turning Off an Entire Group

- 1) Click the radio button to the left of the House Code you want to address with your left mouse button.
- 2) Select the **Group Number** by either typing in the group number to the right of **Group Number** or click the Up and Down arrows to the right of the edit box until the desired group number is displayed.
- 3) Click **Entire Group OFF** with your left mouse button. Every Leviton module in the selected groups will turn off.

In a C-Max program, you can use the **THEN Transmit X10 Group (Leviton)** command to accomplish the same task under program control.

4.6.2.3.4 Removing a Leviton Module From a Group

- 1) Click the radio button to the left of the **House Code** you want to train with your left mouse button.
- 2) Click the radio button to the left of **Key Code** to select the Leviton module you want to remove from the group.
- 3) Select the **Group Number** by either typing in the group number to the right of **Group Number** or click the Up and Down arrows to the right of the edit box until the desired group number is displayed.

4) Click **Remove from Group** with your left mouse button. That module will no longer respond to that group command, but can be set to another group if desired.

4.6.3 Program File

This utility has a single menu choice: **Download Program in Editor to Controller**. Selecting this utility will immediately download the program currently in the program editor to you controller. The Date and Time field of the controller access screen will turn into a progress bar (horizontal bar graph) to show the progress of the operation. Note that you can also download a program into the controller by using the **Download Project** procedure described under **Projects** in this manual section.

4.6.4 Controller Utility

These utilities are used to access and update certain types of data in your controller, as well as provide tools that can be helpful in program testing or debugging.

4.6.4.1 Set Controller Clock to PC Clock

As the name implies, the controller's internal real time clock will be set to the same time and date as those of the PC. This is how you set the time in the Leopard II. Make sure the PC's date and time are set to the values you want in the controller before doing this!

4.6.4.2 Get Controller Clock.

This command will read the current value of the controller's internal real time clock (date and time) and display the values in the activity status window. The PC's clock will not be affected by this command. This is useful if you just want to see what the time and date in the controller are currently set at.

4.6.4.3 Reload Controller Executive.

Use this command to manually initiate a new download of the controller executive. The executive is the computer code that your Leopard II uses to interpret the C-Max programs that you create and download to it.

The C-Max program editor produces a compressed version of your program, which is then downloaded into the controller. The controller then runs it's own command interpreter to execute the commands in your program. Any new commands offered in a newer version of C-Max needs to have the equivalent new commands added to the internal interpreter so that it will know how to execute them. Reloading the executive thus equates to loading the new command interpreter into your controller.

Whenever you open the controller access window, C-Max compares the executive version currently in the controller against the one in the C-Max installation directory (the disk file is named "Flash512.bin"). If you install a new version of C-Max on your PC and it includes a new executive version, you will immediately see a window indicating that the current executive is not at the latest level as soon as you open the controller access window, and you will be asked if you want to update it now. You should normally answer **Yes** to this question and your executive will be automatically updated. This means that manually reloading the executive with the **Reload Controller Executive** command is rarely needed, but can be useful in certain situations such as:

- If you accidentally program a routine that continuously sends data to the serial port (ie: **Transmit ASCII Message** commands) and C-Max can no longer attach to the controller because of this. If this happens, you can regain control with the following procedure: Have C-Max running and ready to attach to the controller, then power the controller off for a few seconds and then back on again. As soon as the power is back on, go into the controller access screen and immediately run the **Reload Controller Executive** utility. This works because there is a time delay lasting a few seconds between the time the serial port can be accessed and the program actually starts running. Reloading the executive always “kills” a loaded program, so you can make the necessary corrections in your program code and then download it again.

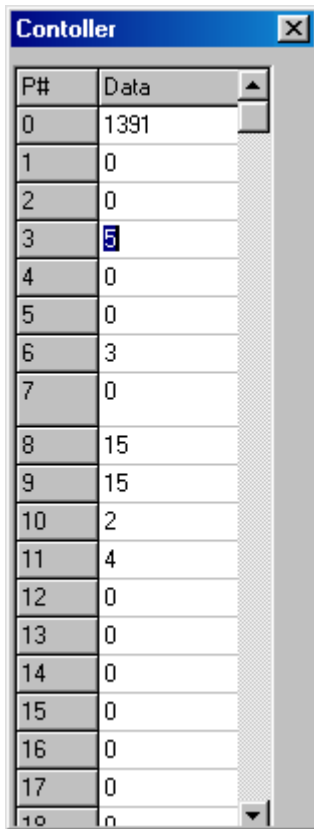
- Sometimes a new executive version is released to correct a very minor bug and the version number is left unchanged. This means that the automatic detection of the available new version will not happen. Use this command to manually do the executive reload.

4.6.4.4 Controller Memory Dump.

This is a diagnostic command used by ADI. There is no functionality in this command for the controller user.

4.6.4.5 Retrieve Controller Parameters.

This command will read the configuration parameters from the controller and present them in an editable list (fig. 47). See the list of parameters in section 5 of this manual to learn the purpose of each parameter. Do not modify the parameters unless you’re certain of what you’re doing, because entering inappropriate values can cause certain features or expansion modules to appear to give erratic results or cease working altogether.



P#	Data
0	1391
1	0
2	0
3	5
4	0
5	0
6	3
7	0
8	15
9	15
10	2
11	4
12	0
13	0
14	0
15	0
16	0
17	0
18	0

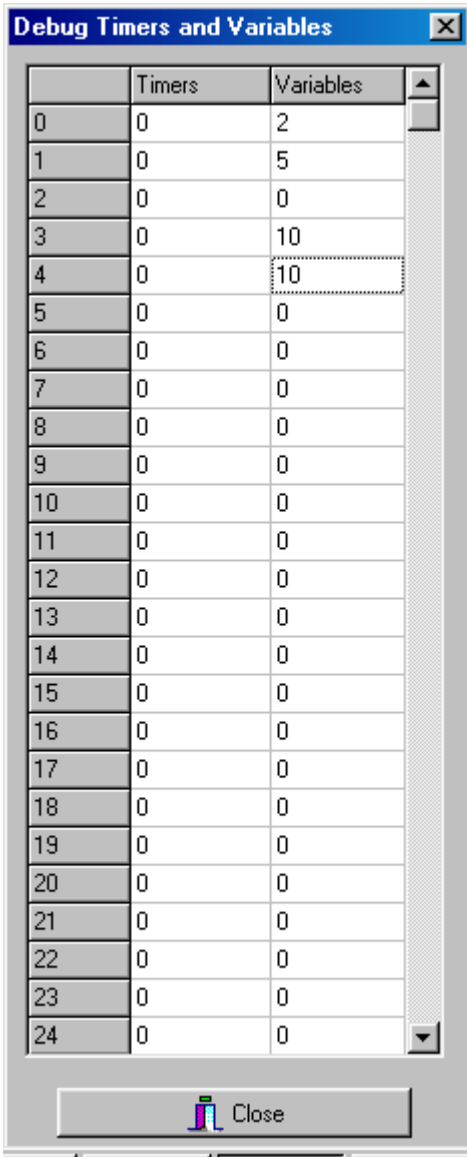
Fig. 47

4.6.4.6 Auto Address Modules.

This utility is used to configure the Adnet addresses of expansion modules. Read the “Auto Addressing” application note in section 5 of this manual to learn the complete procedure. *Warning:* Do not click on this menu choice unless you really intend to address new modules, because this utility will erase all the addresses of currently connected modules! If you do this, you will need to readdress all of them before they can be used again.

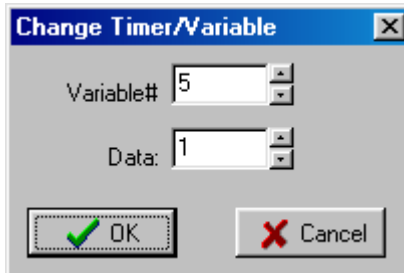
4.6.4.7 Debug Timers and Variables.

Selecting this menu choice will open a continuously updating window (fig. 48) showing the current value of all the timers and variables in the controller. This is one of the most often used utilities because it is handy to be able to see the immediate progression or changes of timers and variables in a running program. You can literally visualize changes as they happen, with no more than about one second of delay.



	Timers	Variables
0	0	2
1	0	5
2	0	0
3	0	10
4	0	10
5	0	0
6	0	0
7	0	0
8	0	0
9	0	0
10	0	0
11	0	0
12	0	0
13	0	0
14	0	0
15	0	0
16	0	0
17	0	0
18	0	0
19	0	0
20	0	0
21	0	0
22	0	0
23	0	0
24	0	0

Fig. 48



Change Timer/Variable

Variable# 5

Data: 1

OK Cancel

Fig. 49

You can also modify the value of a timer or variable on the fly. To do that, click on the timer or variable that you want to change and a small editing window will appear (fig. 49). Enter the new value you want, then click OK.

4.6.5 Module Utility

The module utilities give you direct access to certain module functions. There are 4 utilities:

4.6.5.1 Retrieve Module Parameters.

Clicking on this selection will open a new window (fig. 50) and will initiate a read of all the parameters for all expansion modules. If your system has many modules, this can take a minute or more to complete.

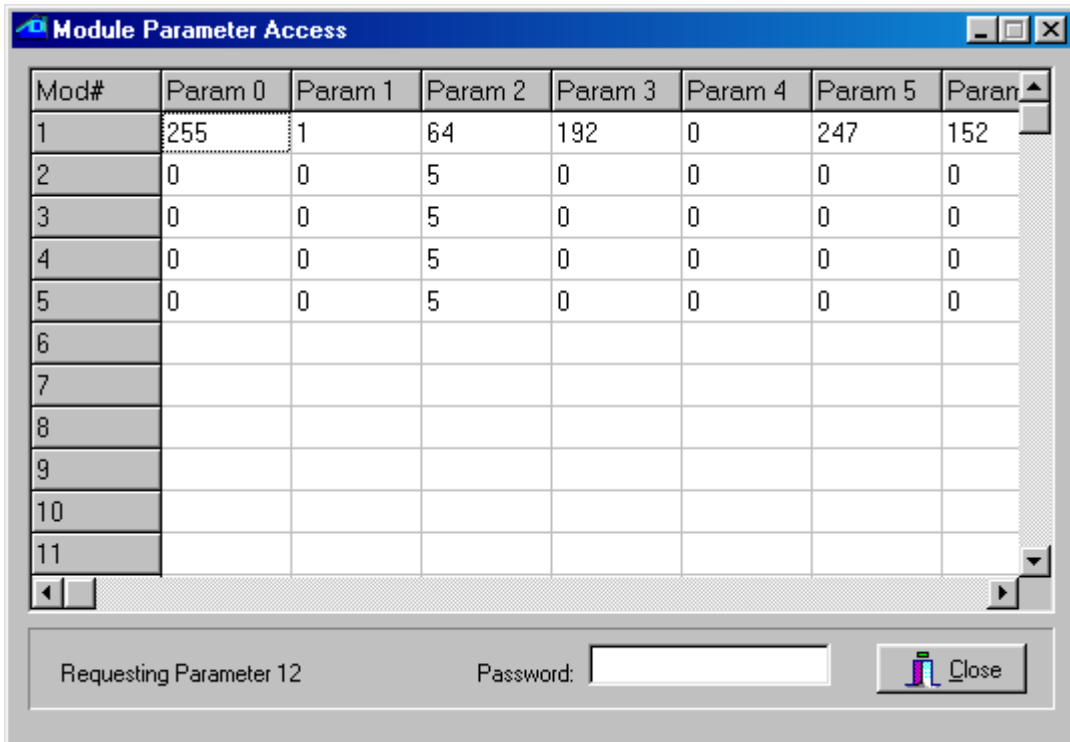


Fig. 50

Module parameters range from 0 to 63. You can use the horizontal scroll bar to view the higher numbered parameters. To modify a parameter, click on the one you want to change. A small window will open (fig 51) allowing you to edit the parameter. If you are asked for a password, enter the value of parameter #6 of module #1 as the password. Modifying module parameters is used to configure certain operating characteristics for that module. Not all parameters can be modified. To find out what a given parameter represents or controls, read the documentation for that specific module type. For example, on a SECU16, parameters 2 and 3 set the low and high thresholds for supervised input mode, while parameters 10 through 17 are read-only and show the analog values of input points 0 to 7 respectively.

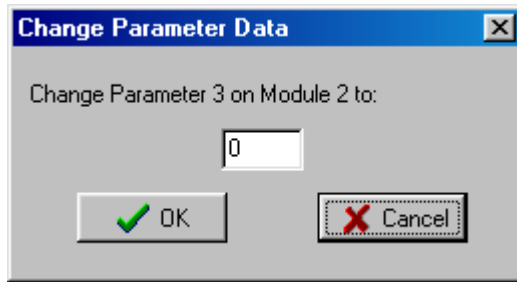


Fig. 51

Retrieving module parameters can also be useful as a programming or debugging tool. For example, you have an analog sensor connected to a SECU16 input and you want to see the analog values for various inputs states: put the sensor in one of the input conditions and retrieve the parameters to see the resulting analog value for the sensor. Then modify the sensor's input conditions and retrieve the parameters again to see the change in analog value. This can be used to establish the range of analog values or simply to test the sensor's operation.

4.6.5.2 Set/Clear Relay

Use this utility to manually set or clear an output relay (SECU16 and RLY8XA modules only). Selecting this utility will open the following window (fig. 52):

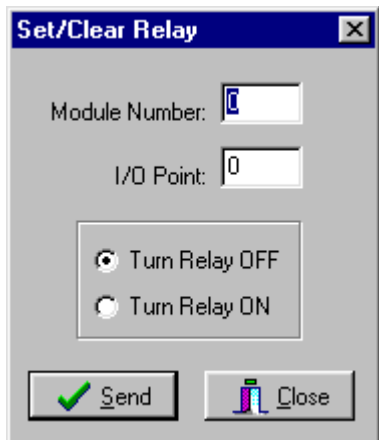


Fig. 52

Simply enter the **Module** and **I/O Point** numbers in the edit boxes, select the **ON** or **OFF** radio button, and click on **Send** to control the relay. This is a generic utility and makes no attempt at preventing you from entering invalid Module and/or I/O Point numbers, so make sure you are addressing the correct relay before clicking on **Send**. For a SECU16, the relays are points 8 through 15, while for a RLY8XA, these are points 0 through 7.

4.6.5.3 Speak Easy.

The Speak Easy utility allows you to record the messages and play them back. The menu selection will open a small utility window (fig 53):



Fig. 53

The detailed procedure for recording messages and playing them back are given in the Speak Easy manual, but here is a summary of the steps involved:

4.6.5.3.1 Recording a Message:

- Select the module number of the Speak Easy and the message number to record then click the RECORD button.
- The Power LED on the Speak Easy will begin to blink green.
- Press and hold the Auto Address/Record button on the Speak Easy.
- The Power LED on the Speak Easy will turn to solid green and start recording.
- Release the Auto Address/Record button to stop recording. The Power LED will start blinking red.
- The Speak Easy will automatically stop recording if the maximum message length is reached.

4.6.5.3.2 Playing a Message:

- Access the Speak Easy Module Utility as described in the Recording a message section
- Select the module number of the Speak Easy and the message number to play then click the PLAY button.
- The message will begin to play and the Power LED will turn solid orange.
- When the message is finished the Power LED will return to blinking red.

4.6.5.4 ASCII Bobcat

This utility allows you to trigger the transmission of an ASCII message from a serial Bobcat. This menu choice opens up a window (fig 54) in which you can specify the module # of the serial Bobcat and the message number that you want to send.

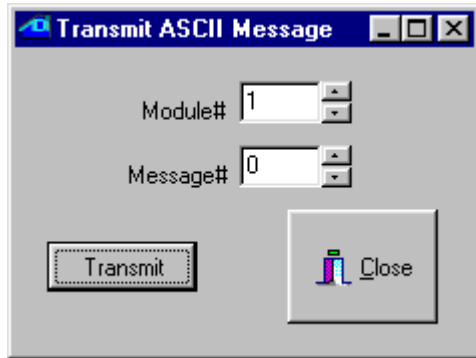


Fig 54

The master controller is also capable of transmitting ASCII messages, but that utility is not provided here. This is because the controller access utilities need the controller's serial port to operate. Transmitting an ASCII message through this same port would conflict with the controller access commands and cause communications error messages.

4.6.5.5 Serial Messages

This pull down menu has two choices, allowing the entering and editing of pager messages and ASCII messages.

4.6.5.5.1 Pager Messages.

This menu choice is used to enter and edit the alphanumeric pager messages that the controller can be programmed to send (using the **THEN Send Page** C-Max command). Note that the ADICON modem is needed to use the paging feature. Selecting this menu opens the following window (fig 55):

User PIN	342234576
Message #1	Temperature Below Lower Limit
Message #2	Water Detector Triggered
Message #3	Alarm Tripped
Message #4	
Message #5	
Message #6	
Message #7	
Message #8	
Message #9	
Message #10	
Message #11	
Message #12	
Message #13	
Message #14	
Message #15	

✓ Load Messages Close

Fig. 55

The first editable line is used to enter the **User PIN** (personal identification number) that your paging service provider has assigned to identify your pager. The subsequent lines allow you to define up to 15 different alphanumeric messages that can be sent to display on your pager under program control.

4.6.5.5.2 ASCII Messages.

Use this menu choice to enter and edit ASCII messages that you want to use with either your controller or a serial Bobcat. These messages can be transmitted under program control using the **THEN Transmit ASCII Message C-Max** command. This selection opens the following window (fig 56):

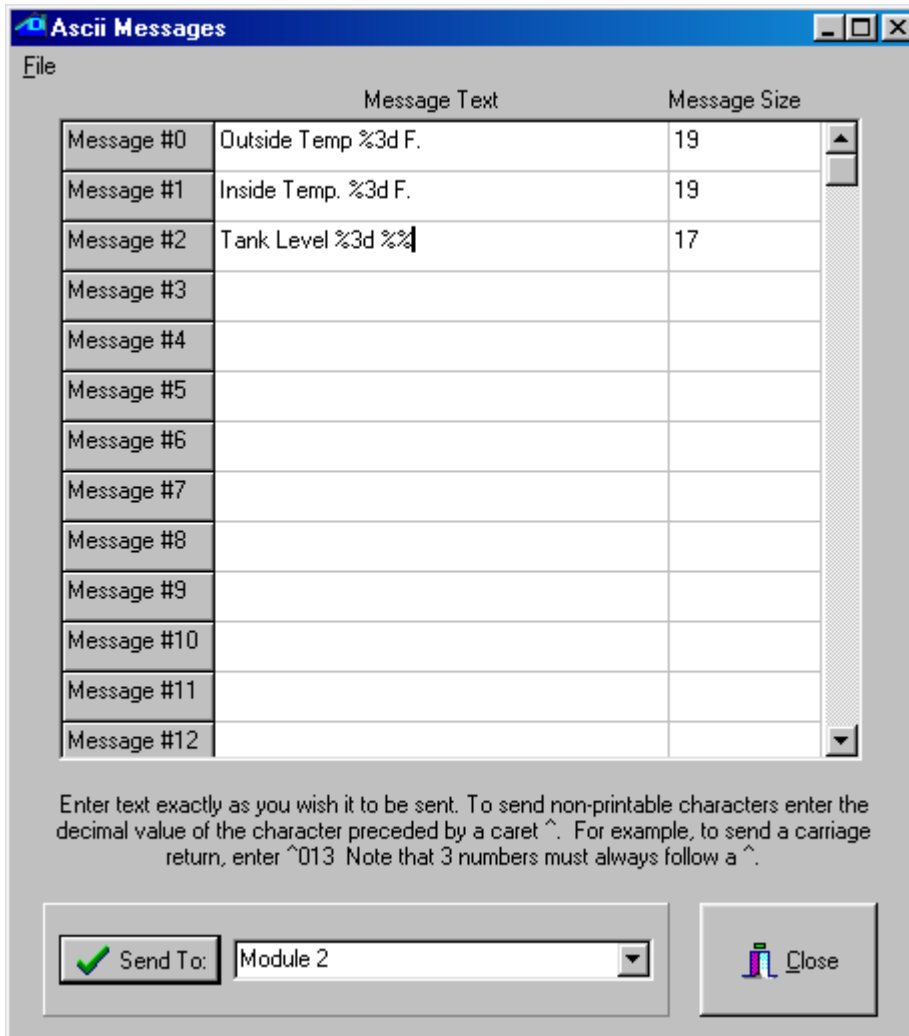


Fig. 56

You can enter up to 128 messages in any given controller or serial Bobcat. Use the scroll bar on the right to edit the higher numbered messages. Note that each controller or serial Bobcat can have its own, different set of ASCII messages. Each message can have a maximum of 32 characters (the **Message Size** column displays the number of characters in a message as it is being edited). In your program code, you can transmit longer messages by simply using two or more **THEN Transmit ASCII Message** commands, as needed.

Click on the line that you want to edit and enter your text. To enter non-printable ASCII characters, enter a caret (^) followed by the 3 digit decimal value for the character. For example, a carriage return would be entered as “^013” (without the quotes). Always enter the three digits for an ASCII value, using leading zeros if needed. You can enter leading and trailing spaces if you wish, they will be saved as such, although they will not be easily visible in the editor. Such spaces can be useful when you plan on transmitting “long” ASCII messages by sending several strings in succession.

The ASCII messages also support embedded formatted variables. The formatting string always begins with a “%” sign and can have several formatting options. See the variable formatting application note in section 5 of this manual to learn how to use the different formatting options. Embedded variables are supported by all ADI controllers and by serial Bobcats with firmware version 6 and above.

Once you have finished editing the ASCII messages, they can be downloaded to the controller or serial Bobcat by selecting the appropriate module # in the list box at the bottom of the screen and then clicking on **Send To**. You can also load and save ASCII messages as disk files on your PC by clicking on the **File** pull down menu:

- **Open** will open a browsing window allowing you to locate the directory and file containing the ASCII message file that you want to load. ASCII message files have a “.asc” file extension.
- **Save** allows you to quickly save a file that you previously opened to make changes.
- **Save As** allows you to create a new ASCII message file. A file-browsing window will appear, allowing you to locate or create a directory, as well as a file name, to save your ASCII messages in. ASCII message files will be given a “.asc” file extension.

5 Application Notes

This manual section contains instructions and tutorials designed to help the reader learn some of the more complex features or procedures that can be done with the Leopard II. Some of these are referred to in the previous sections when a detailed explanation would have been needed to fully understand the specified feature. The following application notes are included:

- Formatting variables that are embedded in screen text objects and ASCII messages.
- Creating and using icons on Leopard buttons.
- A sample C-Max application, creating a two stage heat and cool thermostat
- The procedure for auto-addressing expansion modules.
- An example of using expansion modules
- List of controller configuration parameters

5.1 Formatting Variables in C-Max 2.0

One of the many new features in C-Max 2.0 is the enhanced formatting now available for variables. This new capability is available in two distinct areas of “variable” usage: in embedded screen text objects on a Leopard screen and in ASCII strings, where embedded variables were not even possible before. This second possibility also means that Ocelot users can take advantage of this new feature. In the case of ASCII strings, embedded variables and formatting can be used with the master controller’s serial port, a slave’s serial port, and the serial bobcat. Note: A serial bobcat needs to have firmware version 6 or higher to support embedded variables.

This new feature is also the subject of a minor backward compatibility issue if you have an existing (pre C-Max 2.0) Leopard application that has embedded variables in screen text objects. You will need to add a “u” after the field width number to duplicate the former display behavior (i.e.: a “%4” must be modified to “%4u”). Failure to do that will result in a blank field where the variable should be, but before doing just that, you will want to look at the new formatting capabilities presented here and possibly optimize the way your variables are displayed.

This application note will show two sample programs where variables are formatted to obtain the desired representation or display. The first example will show formatting of Leopard screen variables while the second example will embed and format variables in ASCII strings, which can be used with a Leopard, Ocelot, or serial bobcat. But first, let’s take a closer look at variables themselves.

5.1.1 Controller Variables

What exactly do we mean by formatting variables? To help you better understand we will start by a short description of your controller’s variables in their natural form. You will then understand the formatting capabilities more easily.

If you are already familiar with your controller’s variables, you will have noticed that they are always an integer with values ranging from 0 to 65535. This apparently odd range is because they are internally represented by 16 bit binary memory locations. In a binary system, each bit (from right to left) represents an increasing value of a power of two. Thus the first bit is the 1’s, then the 2’s, the 4’s, the 8’s and so on until we get to the 16th bit which is the 32768’s. This is just like the more familiar decimal system where we have the 1’s, the 10’s, the 100’s, etc. The maximum value that can be represented is when we have each bit set. Therefore if we add $1 + 2 + 4 + 8 + \dots + 32768 = 65535$.

One thing you may have noticed is that in C-Max, there are no negative numbers. The above description of your controller’s variables is called an “unsigned 16 bit integer”. In the past, this was sometimes annoying when we wanted to display things like temperatures in an embedded variable on a Leopard screen...if the temperature went below zero. If you did try to make a variable go below zero by subtracting or decrementing it past zero, you might have noticed that it “rolled over” to 65535 and then continued to decrement downwards. This is very similar to a mechanical car odometer that counts up to 99999 and then rolls over to 00000. If you then turn it backwards it will show 99999 again and keep decreasing. In binary number usage, there is a convention that allows us to consider a 16 bit number as a “signed 16 bit integer”. In this case, the highest bit indicates whether the number is positive or negative (1 = negative, 0 = positive) and the remaining 15 bits give us the number’s value. We still have the same total range but it is now “shifted” to represent numbers from -32768 to 32768. In that system, called “two’s complement”, the bit pattern for 65535 is equal to “-1”, $65534 = -2$ and so on. This means that The rolling over of your controller’s variables from 0 to 65535 when subtracting or decrementing is already the correct behavior for signed 16 bit integers, but there was no means to visually represent them on your Leopard screen until now. This signed representation is obtained when you use the “d” formatting option (as described in the next section).

Although you can now interpret and display a negative number on your Leopard screen or in an ASCII string, they are still positive-only in the C-Max instruction logic. This can cause a few surprises if you are not careful with “greater than/less than” type instructions. For example, if you are looking for a temperature

as being between -10 and +10 deg F for a true condition, you will need to consider it logically as “IF < 11 OR > 65525”. You cannot use an AND because you are looking for a number that is possibly at either end of the positive number scale.

5.1.2 Formatting Options

Here is a complete list of the formatting options available for screen objects and ASCII strings (Fig 57). If you are familiar with the “C” programming language and the *printf* statement, then these options will already be familiar to you.

```
%d    signed decimal  
%u    unsigned decimal  
%o    unsigned octal  
%x    unsigned hexadecimal with lowercase for letters  
%X    unsigned hexadecimal with uppercase for letters  
%c    single ASCII character  
%%    to display a “%” sign literally
```

Fig. 57

The above formats can be modified with one or more of the following options (these are inserted between the % and the letter) (Fig. 58):

```
-      left justify  
+      always display a sign (+ or -)  
0 (zero) pad with leading zeros
```

Fig. 58

Finally, you specify the width of the field reserved for the variable (including any + or – sign) using a number between the % sign (or after one of the above modifiers) and the format letter. You can also specify a minimum number of displayed digits (padded if necessary with leading zeros) by adding a decimal point and a second digit.

One thing to remember about field width specifications: these specify a minimum width, padded with blanks or zeros as specified. If the variable should happen to be or become wider than the specified width number, the field will automatically expand and push any text after it to the right. However, if the variable then becomes shorter again, you might get unwanted characters displayed past the end of the text object. You can easily avoid this by simply specifying a field width wide enough to accommodate the largest variable value that can be expected.

Here are a few examples of format specifications, a variable value, and what it would look like once formatted (Fig. 59).

<u>Format</u>	<u>Variable</u>	<u>Appearance</u>
%6u	1234	1234
%-6u	1234	1234
%3d	65535	-1
%+3d	12	+12
%03d	7	007
%5.3u	17	017
%c	65	A
%3u%%	68	68%
%4X	65535	FFFF
%4x	65535	ffff

Fig. 59

5.1.3 Screen Display Example

This first example will be of a Leopard screen display showing various types of information using formatting to optimize each type of data presented. Let’s create a general information screen that will be displayed when the Leopard’s home control menus are not being utilized by the user. We will show the current time, date, inside and outside temperatures, and relative humidity (these last three values obtained from temperature and humidity bobcats). Let’s examine each type of data and the formatting that will be applied:

Time – We want to use the largest font available to display the time in a digital clock format. Since the largest font can show numbers only, we will use two such fields side by side so that a “colon” can be created in between by using two single character text objects each with a lowercase “o”. The left side number field will show the hours using a simple “%2u” format to allow for two characters. The right number will be the minutes and will use a “%02u” format string to allow for two characters and also pad the leftmost column with a zero if the minutes value is a single digit. This is to get a display of “12:02” instead of “12: 2”.

Date – We want the date to show in the form “Nov 9,2002”. The Leopard does not have variable text string functions but we can simulate it by creating single character fields side by side and then loading the ASCII value of the character for each letter we want to make up the string. We will use three “%c” format strings and overlap them in the screen definition file to get the desired effect. Note that even though creating the format string needs two characters (“%c”) in the touch screen editor, we know this will display as only one character, so it is ok to overlap the text objects in the editor. The day of the month uses a “%2u,” format to reserve two character positions and display the comma. The year uses a “%4u” format for 4 characters.

Temperature – This is a more classic format string using text with the variable format specification inserted at the appropriate location. For the inside temperature, we use “Ins. Temp:%3d F.” for up to three digits (if it gets really hot...). For the outside temperature we use “Out. Temp:%+4d F.” because we want up to three digits as well as a “+” or “-“ displayed, which uses up one digit space.

Relative Humidity - This is expressed as a percentage so we will use a string in the form “R.H. %3u%%” which is really a 3 digit display (for up to 100%) followed by the percent symbol itself.

Here is a list summarizing all the properties of our text objects (Fig. 60):

<u>Obj#</u>	<u>Format string</u>	<u>Font Size</u>	<u>Embedded Variable</u>
1	%2u	Lrg # only	10
2	%02u	Lrg # only	11
3	o	Large	(none) used to make "colon" top
4	o	Large	(none) used to make "colon" bottom
5	%c	Large	12
6	%c	Large	13
7	%c	Large	14
8	%2u,	Large	15
9	%4u	Large	16
10	Ins. Temp:%3d F.	Large	17
11	Out. Temp:%+4d F.	Large	18
12	R.H. %3u%%	Large	19

Fig. 60

Here is a screen capture of the touch screen editor showing the relative positioning of each text object (Fig. 61). Getting the best-looking display sometimes requires experimentation and testing with the actual Leopard. Notice that overlapping causes some of the objects to be only partially visible:

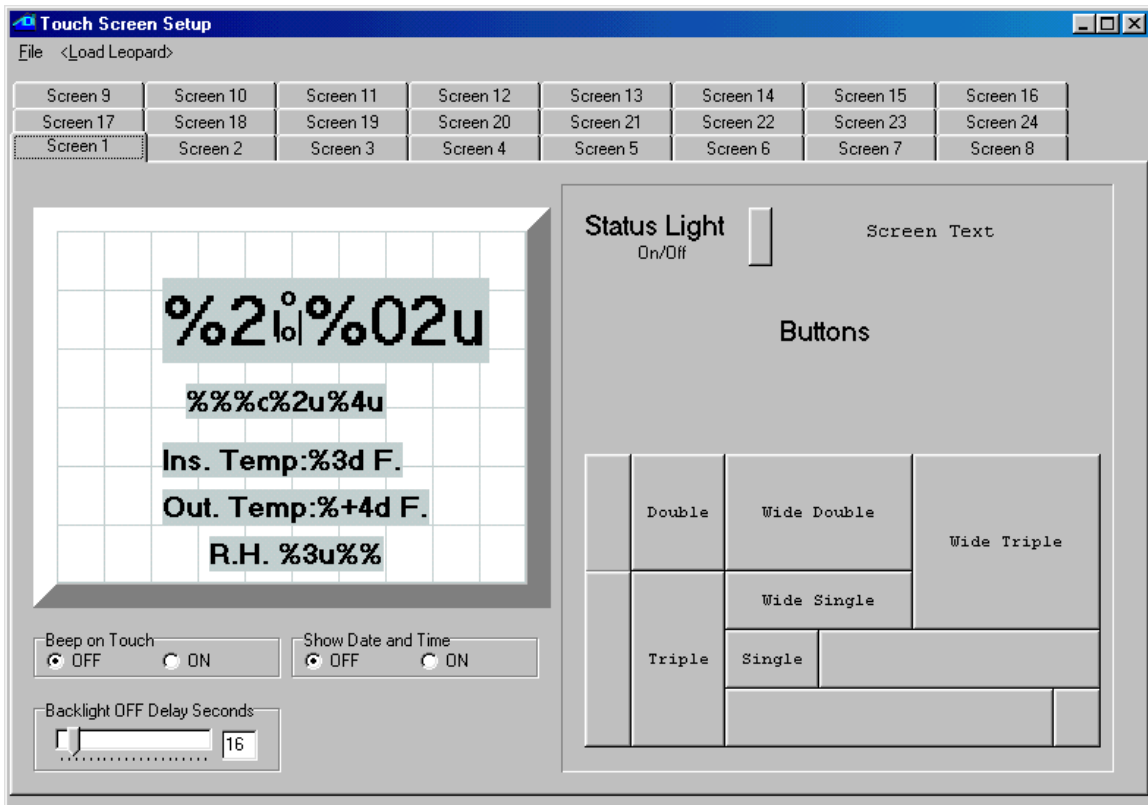


Fig. 61

The C-Max program used to produce the various data values displayed is shown below (Fig. 62). Only the relevant program lines are shown, omitting the other screens and buttons used to switch to the information screen:

```

0001 - // ** Time of Day **
0002 - IF Time of Day is > 00:00 // whether time = 0:00
0003 -     THEN Load Data to: Variable #0 // or not, store in 0
0004 -     ELSE Load Data to: Variable #0 // (minutes since 0:00)
0005 - IF Variable #0 is = Variable #0 // on every pass
0006 -     THEN Variable #1 = Variable #0 // copy time in var 1
0007 -     THEN Variable #1 / 60 // calculate hours
0008 -     THEN Variable #10 = Variable #1 // put result in var 10
0009 -     THEN Variable #0 % 60 // calculate minutes
0010 -     THEN Variable #11 = Variable #0 // put result in var 11
0011 - // ** Month **
0012 - IF Month is = January (1) // If January
0013 -     THEN Variable #12 = 74 // ASCII "J"
0014 -     THEN Variable #13 = 97 // ASCII "a"
0015 -     THEN Variable #14 = 110 // ASCII "n"
0016 - IF Month is = February (2) // If February
0017 -     THEN Variable #12 = 70 // ASCII "F"
0018 -     THEN Variable #13 = 101 // ASCII "e"
0019 -     THEN Variable #14 = 98 // ASCII "b"
0020 - IF Month is = March (3) // If March
0021 -     THEN Variable #12 = 77 // ASCII "M"
0022 -     THEN Variable #13 = 97 // ASCII "a"
0023 -     THEN Variable #14 = 114 // ASCII "r"
0024 - IF Month is = April (4) // If April
0025 -     THEN Variable #12 = 65 // ASCII "A"
0026 -     THEN Variable #13 = 112 // ASCII "p"
0027 -     THEN Variable #14 = 114 // ASCII "r"
0028 - IF Month is = May (5) // If May
0029 -     THEN Variable #12 = 77 // ASCII "M"
0030 -     THEN Variable #13 = 97 // ASCII "a"
0031 -     THEN Variable #14 = 121 // ASCII "y"
0032 - IF Month is = June (6) // If June
0033 -     THEN Variable #12 = 74 // ASCII "J"
0034 -     THEN Variable #13 = 117 // ASCII "u"
0035 -     THEN Variable #14 = 110 // ASCII "n"
0036 - IF Month is = July (7) // If July
0037 -     THEN Variable #12 = 74 // ASCII "J"
0038 -     THEN Variable #13 = 117 // ASCII "u"
0039 -     THEN Variable #14 = 108 // ASCII "l"
0040 - IF Month is = August (8) // If August
0041 -     THEN Variable #12 = 65 // ASCII "A"
0042 -     THEN Variable #13 = 117 // ASCII "u"
0043 -     THEN Variable #14 = 103 // ASCII "g"
0044 - IF Month is = September (9) // If September
0045 -     THEN Variable #12 = 83 // ASCII "S"
0046 -     THEN Variable #13 = 101 // ASCII "e"
0047 -     THEN Variable #14 = 112 // ASCII "p"
0048 - IF Month is = October (10) // If October
0049 -     THEN Variable #12 = 79 // ASCII "O"
0050 -     THEN Variable #13 = 99 // ASCII "c"
0051 -     THEN Variable #14 = 116 // ASCII "t"
0052 - IF Month is = November (11) // If November
0053 -     THEN Variable #12 = 78 // ASCII "N"

```



```

0054 -     THEN Variable #13 = 111           // ASCII "o"
0055 -     THEN Variable #14 = 118           // ASCII "v"
0056 - IF   Month is = December (12)         // If December
0057 -     THEN Variable #12 = 68           // ASCII "D"
0058 -     THEN Variable #13 = 101          // ASCII "e"
0059 -     THEN Variable #14 = 99           // ASCII "c"
0060 -                                           // ** Day **
0061 - IF   Day of Month is > 0              // If day > 0 (always)
0062 -     THEN Load Data to: Variable #15   // put in var. 15
0063 -                                           // ** Year **
0064 - IF   Year is > 0                      // if year >0 (always)
0065 -     THEN Load Data to: Variable #0    // store in var 0
0066 -     THEN Variable #0 + 2000          // convert to 4 digits
0067 -     THEN Variable #16 = Variable #0   // copy to var 16
0068 -                                           // ** Inside temp **
0069 - IF   Module #1 is < 256              // If temp < 156 deg
0070 -     THEN Load Data to: Variable #0    // store in var 0
0071 -     THEN Variable #0 - 100           // subtract 100
0072 -     THEN Variable #17 = Variable #0   // store in var 17
0073 -                                           // ** Outside temp **
0074 - IF   Module #2 is < 256              // If temp < 156 deg
0075 -     THEN Load Data to: Variable #0    // store in var 0
0076 -     THEN Variable #0 - 100           // subtract 100
0077 -     THEN Variable #18 = Variable #0   // store in var 18
0078 -                                           // ** Relative Hum. **
0079 - IF   Module #3 is < 256              // If hum < 156 deg
0080 -     THEN Load Data to: Variable #0    // store in var 0
0081 -     THEN Variable #0 - 100           // subtract 100
0082 -     THEN Variable #19 = Variable #0   // store in var 19

```

Fig. 62

The program is quite straightforward and needs little explanation, but the following comments might be useful to help you understand some areas:

- Lines 1,2, and 3 look to see if the time is equal to midnight or not, and captures it's value in Variable #0 in either case. This is just a way to capture the current time into a variable. The time value obtained is in minutes since midnight, not the base 60 time we are used to. The conversion to hours and minutes is done in line 5 to 10.

- Line 7 takes a copy of the time and divides it by 60 to get the number of hours.

- Line 9 gets the remainder (modulo) of the time divided by 60 to get the minutes. The modulo function is also new in C-Max 2.0 and is quite useful for this type of base conversion.

- Lines 12 through 15 (as well as the eleven other program segments following them) copy the ASCII values of the letters we want to display for the month names.

- Line 65 captures the year into a variable in a manner similar to the way the time was captured. The year value will be a two digit year (i.e.: 12 for the year 2012). Line 66 adds 2000 to the year to convert it to a 4 digit value (...please remember to adjust this line in the year 3000...).

- Lines 69 through 72 (as well as the subsequent two similar segments) reads the bobcat's value as an "extended variable" which is easier to work with then using an "IF bobcat..." (more suited for a direct comparison instead of capturing in a variable for screen display). Note that such raw bobcat values need to have 100 subtracted to get the actual reading. For the outside temperature bobcat, a temperature below zero will cause the final value (once 100 is subtracted) to roll over and be in the 65000 range, but the variable

formatting string ending with a “d” will ensure that it is interpreted as a signed integer and a “-“ sign will be displayed if the temperature is indeed below zero.

Fig. 63 shows an actual Leopard screen photo of the resulting display produced by this program:



Fig. 63

5.1.4 Formatting Variables in ASCII Strings

Our second example will show how we can use variable formatting in ASCII strings. As previously stated, embedding variables (in any form) in ASCII strings was not available prior to C-Max 2.0. This example will show an application where we want to log a temperature reading from a temperature bobcat every half-hour (along with the date and time of the sample) to the serial port of the controller. It is assumed that the controller’s serial port will be connected to a PC or other device capable of recording serial data in a file for later examination.

As mentioned at the beginning, you could use your master controller’s serial port, a slave’s serial port, or a serial (ASCII) bobcat to output the strings (the module’s number or map name is selected when programming the “Transmit ASCII Message” command). If you use a serial bobcat, follow the instructions received with the bobcat for loading the ASCII string definitions to the bobcat. If you want to use a slave controller to output the strings (this is also a new capability in C-Max 2.0) you must load the string definitions into that slave using it’s own serial port; you cannot load them over the bus from the master like for a serial bobcat. You must also make sure that both the master and the slave(s) are running the new executive version that comes with C-Max 2.0

We will create the format definitions and C-Max code necessary to produce a text log of the temperature readings with the following appearance (Fig. 64):

```
MM/DD/YY hh:mm Outside Temperature: +/- xx F.
```

Where:

```
MM = month  
DD = day  
YY = year  
hh = hour  
mm = minute  
xx = temperature
```

Fig. 64

Just like format definitions are entered in screen text objects for Leopard screens, we use the “ASCII Messages” editing utility to first define each string along with the formatting information for the embedded variable. This utility is found under the “Serial Messages” heading after attaching to the controller. The ASCII messages are defined as text strings with a maximum length of 32 characters, and up to 128 such strings can be defined. Note that characters such as carriage return are entered as “^013” and this counts as 4 characters in the ASCII string. Since we cannot have more than one embedded variable per ASCII string, we will need to define several such strings which will then be output in succession so that they concatenate to form one long string in the log. Only the last string will thus have the carriage return and linefeed characters to end the line. We will define the following five ASCII strings (Fig. 65). The quotes shown are to help you see the spaces in some strings and not actually entered:

```
"%02d/"      used for the month and day (with leading zero if 1 digit)  
"%02d "  
"%02d:"      used for the year and minute (with leading zero if 1 digit)  
"%02d:"      used for the hour (with leading zero if 1 digit)  
"Outside Temperature: "  
"%+4d F.^013^010"  temperature always with sign and line end chars
```

Fig. 65

Here are the strings as they are entered into the ASCII strings definition utility (Fig. 66)

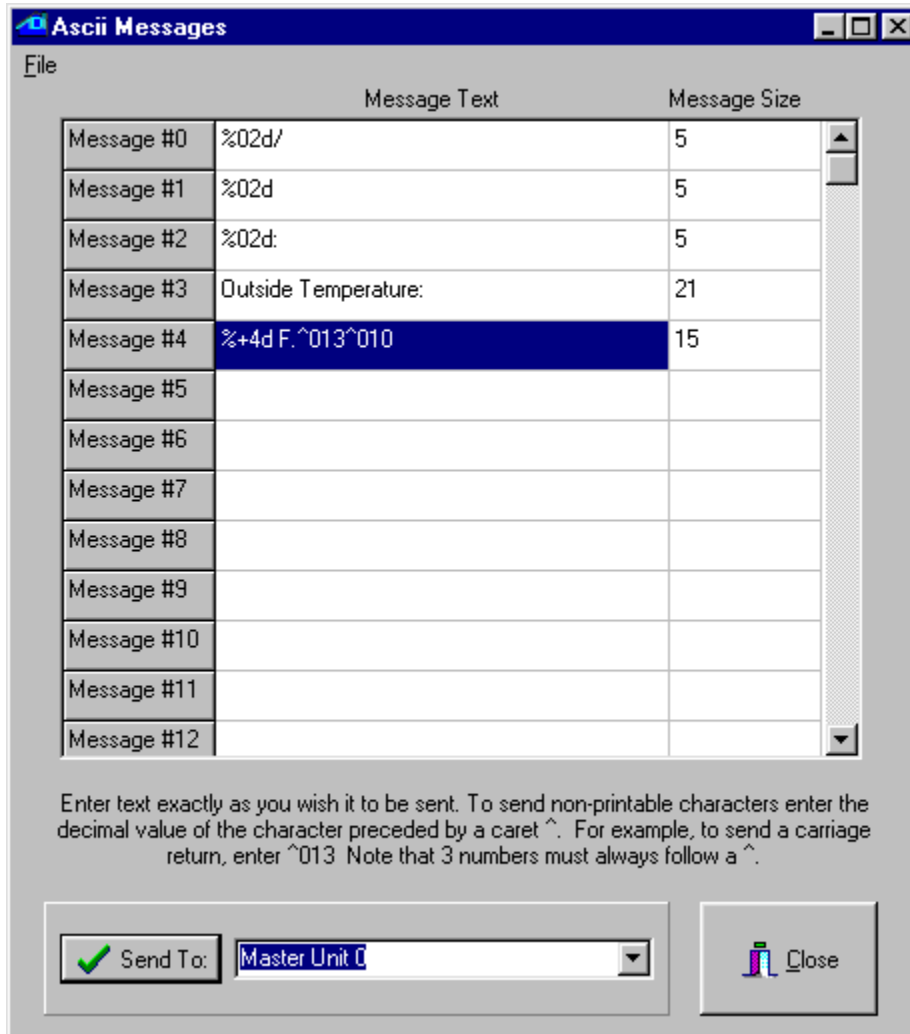


Fig. 66

Once you are done entering the strings, you download them to your controller by selecting the appropriate module and then clicking on the “Send To:” button. The Controller Access utility window will show the strings being downloaded in the status window. Once this is done, we are ready to create the C-Max program that will use these strings. Here is the program text used for our example (Fig. 67). If you looked at the preceding example with the Leopard screen, you will find that the first program lines are quite similar, since that example also involved obtaining the current date and time into variables:

```

0001 -                                     // ** Time of Day **
0002 - IF Time of Day is > 00:00         // whether time is 0:00
0003 -     THEN Load Data to: Variable #0 // or not put in var 0
0004 -     ELSE Load Data to: Variable #0 // (mins since 0:00)
0005 - IF Variable #0 is = Variable #0    // on every pass
0006 -     THEN Variable #1 = Variable #0 // copy time in var 1
0007 -     THEN Variable #1 / 60          // calculate hours
0008 -     THEN Variable #10 = Variable #1 // put result in var 10
0009 -     THEN Variable #0 % 60         // calculate minutes
0010 -     THEN Variable #11 = Variable #0 // put result in var 11
0011 -                                     // ** Day **
0012 - IF Day of Month is > 0            // If day > 0 (always)
0013 -     THEN Load Data to: Variable #12 // put in var. 12
0014 -                                     // ** Month **
0015 - IF Month is > January (1)         // If month > 1
0016 -     THEN Load Data to: Variable #13 // put in var. 13
0017 -     ELSE Load Data to: Variable #13 // whether jan or not
0018 -                                     // ** Year **
0019 - IF Year is > 0                    // if year >0 (always)
0020 -     THEN Load Data to: Variable #14 // copy to var 14
0021 -                                     // ** Outside temp **
0022 - IF Module #1 is < 256            // If temp < 156 deg
0023 -     THEN Load Data to: Variable #0 // store in var 0
0024 -     THEN Variable #0 - 100        // subtract 100
0025 -     THEN Variable #15 = Variable #0 // store in var 15
0026 -                                     // ** trigger event **
0027 - IF Variable #11 becomes = 0       // if on the hour
0028 -     OR Variable #11 becomes = 30   // or on the half hour
0029 -     THEN Send Ocelot Message 0 w/ Variable #13 // transmit month
0030 -     THEN Send Ocelot Message 0 w/ Variable #12 // transmit day
0031 -     THEN Send Ocelot Message 1 w/ Variable #14 // transmit year
0032 -     THEN Send Ocelot Message 2 w/ Variable #10 // transmit hour
0033 -     THEN Send Ocelot Message 1 w/ Variable #11 // transmit minute
0034 -     THEN Send Ocelot Message 3 w/ Variable #0 // xmit "Out. temp.."
0035 -     THEN Send Ocelot Message 4 w/ Variable #15 // xmit temper. +eol
0036 - End Program                       //

```

Fig. 67

If you need help in understanding how lines 1 to 25 work, you can view the notes for the Leopard screen text example shown earlier in this document (see Fig. 62 and the text following it) since the code usage is almost identical. In our current example, the events that trigger a series of logging strings to be transmitted (to make up one log entry) are shown on lines 27 and 28; when the minutes component of the time is 0 and 30. Here is an actual sample of a log file produced with our logging program (Fig. 67):

```
11/12/02 21:00 Outside Temperature: +60 F.
11/12/02 21:30 Outside Temperature: +60 F.
11/12/02 22:00 Outside Temperature: +60 F.
11/12/02 22:30 Outside Temperature: +58 F.
11/12/02 23:00 Outside Temperature: +57 F.
11/12/02 23:30 Outside Temperature: +55 F.
11/13/02 00:00 Outside Temperature: +54 F.
11/13/02 00:30 Outside Temperature: +52 F.
11/13/02 01:00 Outside Temperature: +49 F.
11/13/02 01:30 Outside Temperature: +47 F.
11/13/02 02:00 Outside Temperature: +47 F.
11/13/02 02:30 Outside Temperature: +46 F.
11/13/02 03:00 Outside Temperature: +47 F.
11/13/02 03:30 Outside Temperature: +47 F.
11/13/02 04:00 Outside Temperature: +46 F.
11/13/02 04:30 Outside Temperature: +46 F.
11/13/02 05:00 Outside Temperature: +49 F.
11/13/02 05:30 Outside Temperature: +50 F.
11/13/02 06:00 Outside Temperature: +50 F.
11/13/02 06:30 Outside Temperature: +51 F.
11/13/02 07:00 Outside Temperature: +51 F.
```

Fig. 68

Note that any single triggering event could be used as a trigger, such as a received X-10 command, Leopard button press, SECU16 input turning on/off, etc. You could also have several types of events in the same program, each one re-using a copy of lines 29 to 33 to transmit the date and time followed by their own customized ASCII string to identify the event or value. You could keep track of events like when your alarm system is armed/disarmed, HVAC run times, etc... anything that translates to a single event. Here is an example where you would want to log when your alarm system is armed and disarmed, supposing that arming produces an X-10 M/1 On command and disarming an X-10 M/1 Off. We added messages 5 and 6 to the ASCII messages with the following text (without the quotes) (Fig. 69):

```
Message #5: "Alarm ARMED^013^010"
Message #6: "Alarm DISARMED^013^010"
```

Fig. 69

Then line 36 (the END statement) of the program in Fig. 67 was removed and the following code was added (Fig. 70):

```
0036 - IF X-10 House M / Unit 1, ON Command Pair // if alarm armed
0037 -     THEN Send Ocelot Message 0 w/ Variable #13 // transmit month
0038 -     THEN Send Ocelot Message 0 w/ Variable #12 // transmit day
0039 -     THEN Send Ocelot Message 1 w/ Variable #14 // transmit year
0040 -     THEN Send Ocelot Message 2 w/ Variable #10 // transmit hour
0041 -     THEN Send Ocelot Message 1 w/ Variable #11 // transmit minute
0042 -     THEN Send Ocelot Message 5 w/ Variable #0 // xmit "Armed" msg.
0043 - IF X-10 House M / Unit 1, OFF Command Pair // if alarm disarmed
0044 -     THEN Send Ocelot Message 0 w/ Variable #13 // transmit month
0045 -     THEN Send Ocelot Message 0 w/ Variable #12 // transmit day
0046 -     THEN Send Ocelot Message 1 w/ Variable #14 // transmit year
0047 -     THEN Send Ocelot Message 2 w/ Variable #10 // transmit hour
0048 -     THEN Send Ocelot Message 1 w/ Variable #11 // transmit minute
0049 -     THEN Send Ocelot Message 6 w/ Variable #0 // xmit "Disarmed" msg.
0050 - End Program
```

Fig. 70

Here is what a log from this new program could look like (Fig. 71):

```
11/12/02 21:00 Outside Temperature: +60 F.
11/12/02 21:14 Alarm ARMED
11/12/02 21:30 Outside Temperature: +60 F.
11/12/02 22:00 Outside Temperature: +60 F.
11/12/02 22:30 Outside Temperature: +58 F.
11/12/02 23:00 Outside Temperature: +57 F.
11/12/02 23:30 Outside Temperature: +55 F.
11/13/02 00:00 Outside Temperature: +54 F.
11/13/02 00:30 Outside Temperature: +52 F.
11/13/02 00:41 Alarm DISARMED
11/13/02 01:00 Outside Temperature: +49 F.
11/13/02 01:30 Outside Temperature: +47 F.
11/13/02 02:00 Outside Temperature: +47 F.
```

Fig. 71

As we have seen by these examples, variable embedding and formatting allows the creation of customized display and serial data that can be optimized for many user specific applications where visual appeal and/or proper organization can make the end result look really professional.

5.2 Creating Icons for Leopard Buttons

5.2.1 Introduction

Among the new features that C-Max 2.0 brings to the Ocelot and Leopard controllers, one of the more sophisticated ones allows the user to create icons that can then be superimposed on Leopard touch button objects. This feature can be used in various ways to create highly customized screens that are more visually appealing and intuitive for the sometimes less technical end user. Professional installers can also provide a much more customer-oriented, value added service by allowing their clients to make specific choices and special requests as to what they would like to see on their screens.

The method used to implement the icons allows a given icon to be used on more than one button. This is done by linking an icon number to the button or text object. Doing it this way results in a considerable memory savings and eases maintenance and updating because only one icon drawing might need to be modified and then the new version automatically applies to all objects using it. To properly use this new feature, it is important that the screen object sizing be well understood beforehand. Let's start by reviewing some screen basics.

The Leopard screen has a graphic resolution of 320 pixels (individual dots) wide by 240 high. If you look at the Touch screen Setup screen in C-Max you will notice that the grid divides the screen area in 10 horizontal touch squares by 6 vertical squares. Each square is thus 32 pixels wide by 40 pixels high (so they are not really "square", but we'll use that term for simplicity). A square is the minimum area that the Leopard can use to determine a button press (ie: pressing anywhere particular within a square makes no difference). The various screen button objects that can be created will always be composed of one or more of these touch squares. Contrasting with that, icons can be defined in almost any size, as long as their width is a multiple of 8 pixels. That means that icons can be created to cover only part of a button or its entire surface area. Furthermore, you can choose to make the original button frame invisible and make the icon itself become the button's image. This offers interesting possibilities that we will see later. Whatever the icon size is, it will always be centered in relation to the button it is linked to. The only minor constraint to the automatic centering is that the icon will always begin on an 8 pixel multiple in relation to the button's left edge. This means that if you create an icon that is 24 pixels or any other odd multiple of 8 wide, then that icon will display slightly to the left (4 pixels in fact) of the actual horizontal center position. If available memory constraints make you still want to do this, you can always compensate at icon design time by shifting your icon's image by four pixels to the right...

The other question you may have is: how many icons can you create? The answer is: up to 200, or until the 48k of memory assigned to icon storage is used up, whichever occurs first. The 48k memory (49,152 bytes) area sets aside 1200 bytes for the icon "directory"; the rest is available for your icon designs. Figuring out the memory usage for an icon is simple: multiply the width of the icon by the height (in pixels), and then divide by 8. For example, the 48 x 32 icon designed in the example below equals 192 bytes. To give you an idea of how much memory is available, the 48k total memory size corresponds to about 5 full Leopard screens.

The icon display feature also incorporates the formerly separate "Screen bitmap" function to display on screen 0 when the Leopard is first powered up. As before, you first create a bitmap image that you want to use for this purpose, but you now assign it as icon# 0 in the System Map. It will then get loaded along with your other icons when you perform the download procedure explained in this application note. One advantage to this is that if you elect not to use a bitmap on screen 0, the memory it would have used becomes available for your other icons instead. The memory in question is part of the 48k total as explained above.

The best way to learn how to design and use icons is with real examples. The steps involved are the following: 1- Decide on the size (in pixels) of the icon you want to create based on what you want to use it for as discussed above. 2- Use a graphics-editing program to create a black and white, bitmap format file of your icon. 3- Assign the bitmap file to an icon object number by editing the C-Max System Map and then download into the Leopard. 4- Design or update your touch screen file to assign the icon object number to the touch button(s) you want to use it with and load the screen file (.tch) into the Leopard.

In this application note, we will present three examples of icon design and usage. The first example shows the detailed steps required to create a simple icon that will be displayed on top of a button object. The second example uses a very large icon to create a whole screen image on which “hot spots” are shown that correspond to invisible buttons defined around the button displaying the large image. The third and last example shows a sample application using the new icon switching instruction for changing icons on buttons under program control.

5.2.2 Designing an icon

For this example, we will design a pair of icons, one showing a light bulb as being lit and the other as dark. We will make these 48 pixels wide and 32 pixels high. These will then be used as substitutes for the “On” and “Off” captions on buttons of various sizes. The 48 x 32 size was chosen here because in this example we will use them on “two square wide by one square wide “ (2 x 1) buttons, one of the most commonly used sizes in Leopard screen design. A button this size is 64 pixels wide and 40 pixels high. If you look carefully, you will notice that the “raised border” look of the Leopard buttons uses 4 pixels along each edge. This means that the usable display area of a 2 x 1 button is reduced to 56 x 32. A width of 56 pixels corresponds to 7 bytes wide, not an even multiple of 8, so we’ll reduce it by another 8 pixels, hence the 48 x 32 size.

The examples shown use the “Paint” program (Mspaint.exe) that comes standard with Microsoft Windows. Feel free to use any another graphics editing program of your choice.

Open the paint program (this is under [Start](#) → [Programs](#) → [Accessories](#)). This will give you a blank screen for a large image. Click on [Image](#) → [Attributes](#) as shown below (Fig. 72):

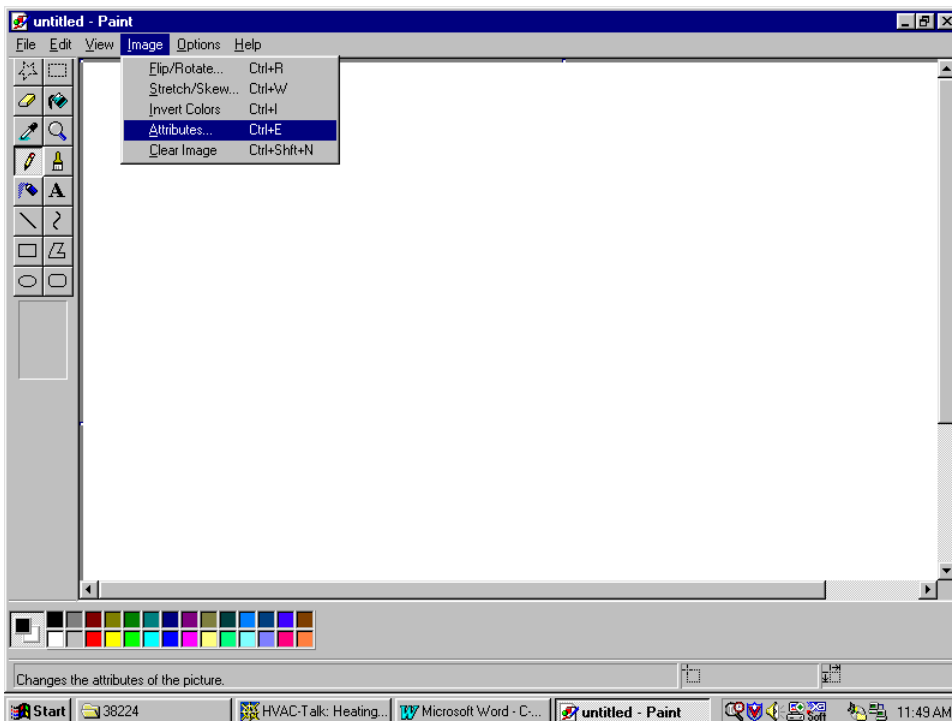


Fig. 72

This will open a smaller window allowing you to choose the image attributes of the bitmap file you will produce. Note that the width and height in pixels has been entered and that it will be a black and white image (Fig. 73)

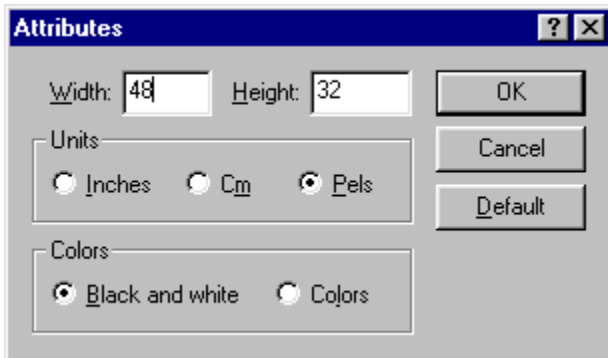


Fig. 73

When you click on “OK” you will probably get a warning message stating that converting to black and white cannot be undone. Just click on “Yes” to continue. You will now notice a rather small white rectangle where you are expected to create your icon... Unless you have lots of patience or terrific eyesight, you’ll want to zoom in on your image to make it easier to work with. Just click View → Zoom → Custom... to open up a custom zoom box and select 600% or 800% to bring it up to a much more workable size (Fig. 73 and Fig. 75)

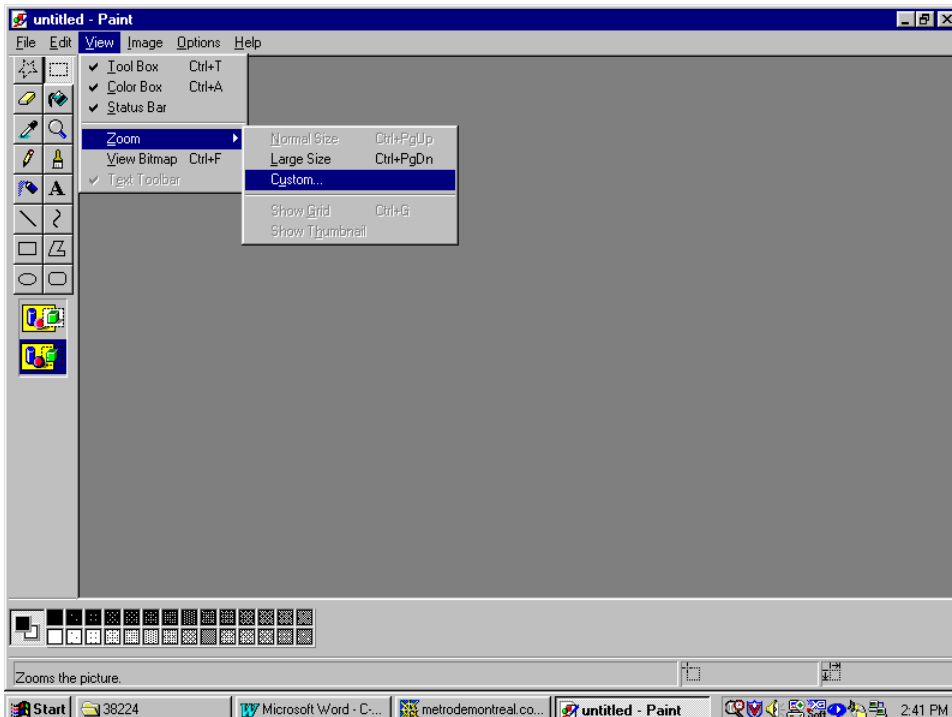


Fig. 74

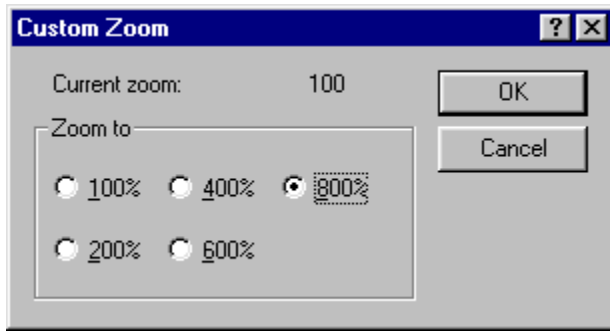


Fig. 75

You may now use the various tools available in the program to create you caption design (Fig. 76)

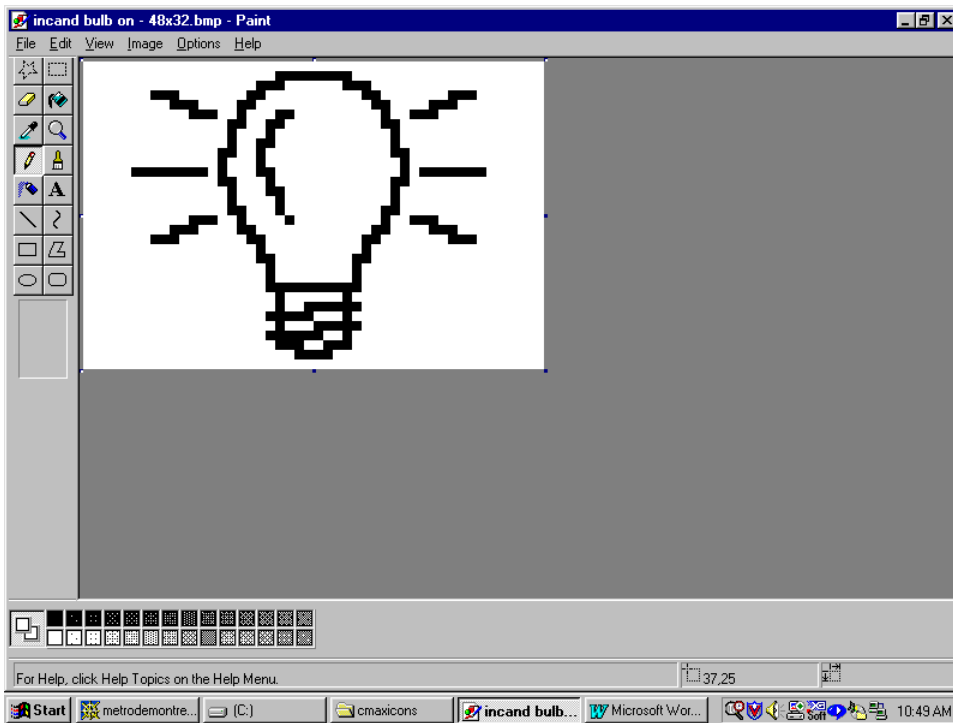


Fig. 76

The design will certainly have a “rough” look when viewed at such a large size but will look much better when viewed normally. You may do this at any time during your design work by simply turning off the zoom feature under View to take a look. Once you have finished, save the image using File → Save As... and give it a meaningful name. You may want to create a directory specifically for all your icons, or sort them by size or any other criteria. After your initial icon is designed, it becomes easy to design similar or related images by using the first one you did as a master. Let’s create the “dark” bulb now by removing the surrounding rays and inverting the color inside the bulb (Fig. 77).

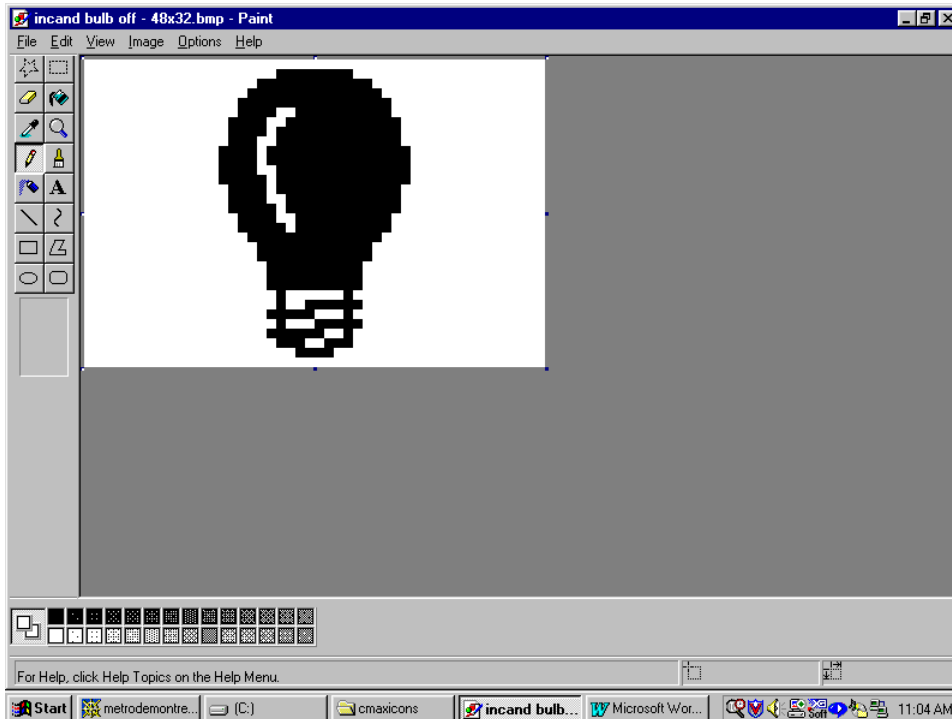


Fig. 77

One thing to consider when designing icons is the final desired dark/light “look” of the icon once displayed on a Leopard screen. In that respect, there is a difference between the original Leopard I and the newer Leopard II that you might want to take into consideration. The Leopard I displays buttons in “reverse video” (in relation to the icon design as seen above) while the Leopard II will show them as displayed above. The Leopard I reverses the colors so that the icon will blend in correctly against the already dark background of a button. In the case of a light bulb icon as we are doing here, we want the lit bulb to have the “light” color and the dark bulb to really look “dark”. For a Leopard I we would thus make the lit bulb; the one with the rays, filled-in with black and the dark bulb with just the bulb outline instead so that the color inversion on the screen will give us the desired effect. Note that in either case, we’ll leave the surrounding background white because that will allow a seamless display with the button’s background over which the icon will display. If you need to invert the colors of your icon after designing it, you can very easily do this in Paint by clicking on Image → Invert Colors.

Once your icons are created, you must now load them into the Leopard’s memory. This is done in C-Max 2.0 by opening the System Map configuration screen (Fig. 78):

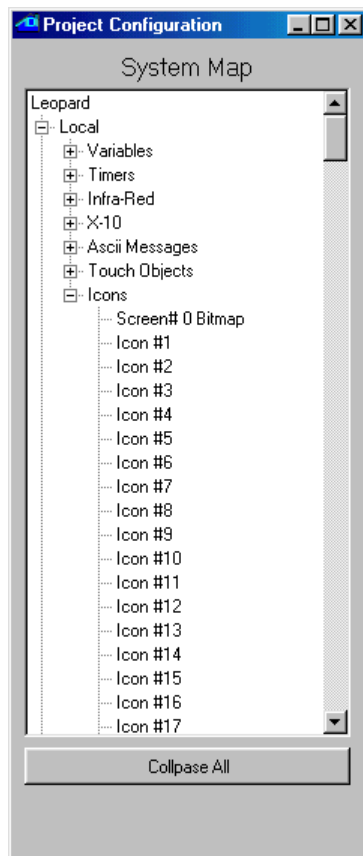


Fig. 78

Select “Icon #1” by clicking on it once and then right click on it to open the file browsing window (Fig. 79).

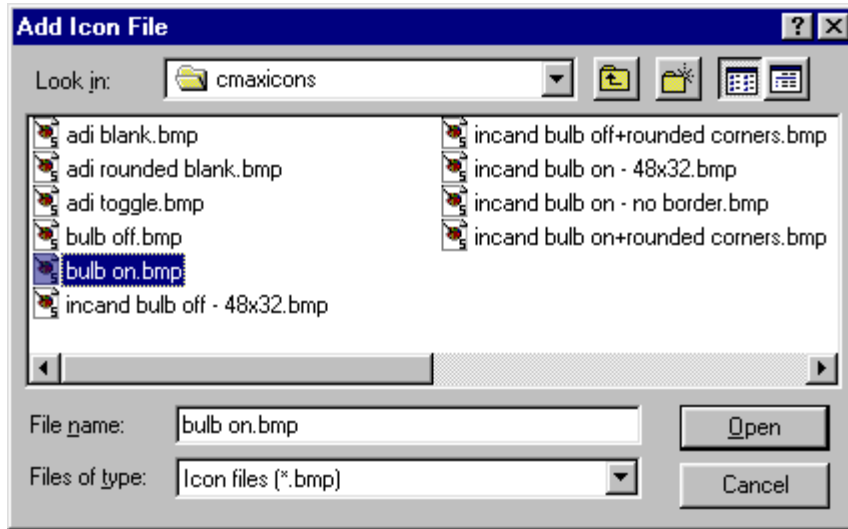


Fig. 79

Select the icon file you want to assign to that icon number and click on “Open”. This will copy the icon file to your project directory (remember this fact; because if you create and keep your icons in a directory other than the project directory, like if you want to use them for several projects, you will need to re-copy any updated versions of the icon to the project directory). Once all your icon files have been assigned to icon numbers, you will need to load them into the Leopard’s memory. This is done using the “Download Project” selection under the “Project” pull down menu on the main C-Max screen (you can also perform any of the other downloads at this time too) and checking the “Download Icons” box (Fig. 80):

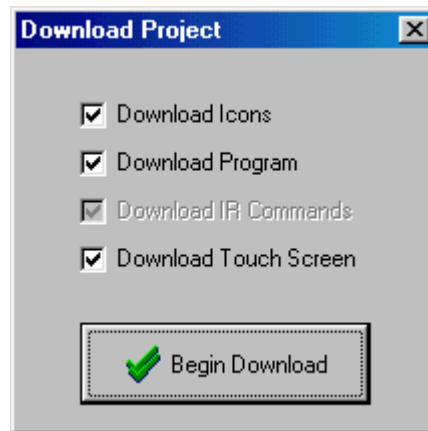


Fig. 80

Click on “Begin Download” and you will see the controller access screen showing the progress of the download operation. Once this is finished, you will now have the icons loaded into your Leopard. The only thing left to do now is to assign the icons to the screen buttons you want to display them on. Open the touch screen editor and load your screen or create a new one and choose the button you want to put an icon on. Right click the button and click on “Enter Caption”. You will see that the caption-editing box now has two new options: the icon number and a check box labeled “hidden border”. Leave that box unchecked for this example, and enter the icon number in the appropriate box. Note that we will not enter a caption since it is being replaced by the icon instead (Fig. 81):

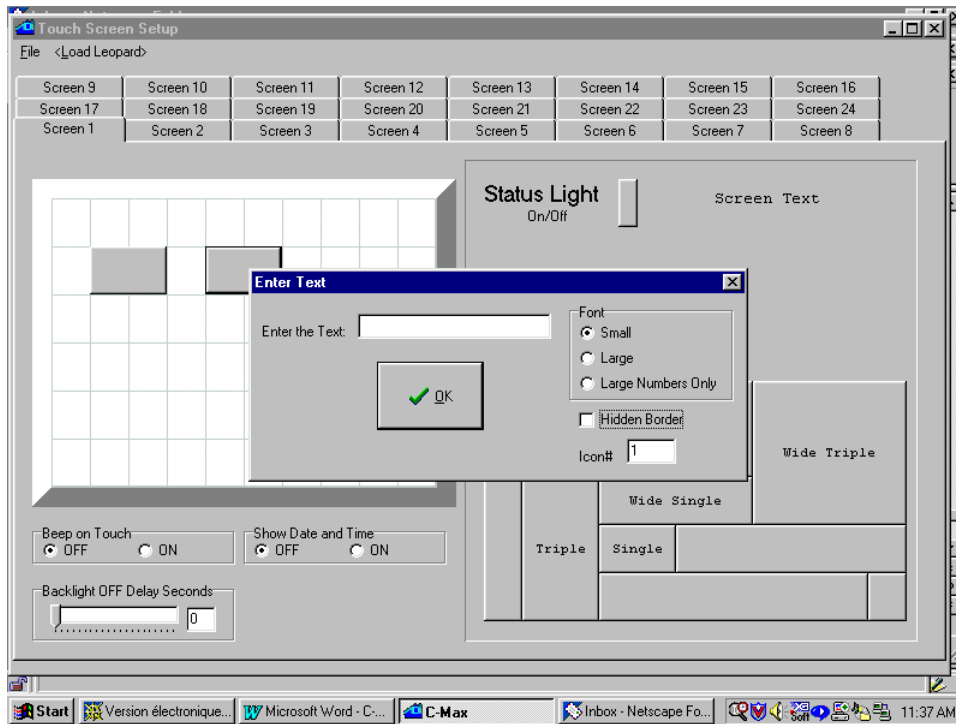


Fig 81

Do this for all the buttons you wish to use to display icons and download the touch screen file by clicking on <Load Leopard> or with the new “Download Project” option on the main screen and see the results of your efforts! Here is an actual picture of various icons that were created, being displayed on a Leopard I screen (Fig 82):

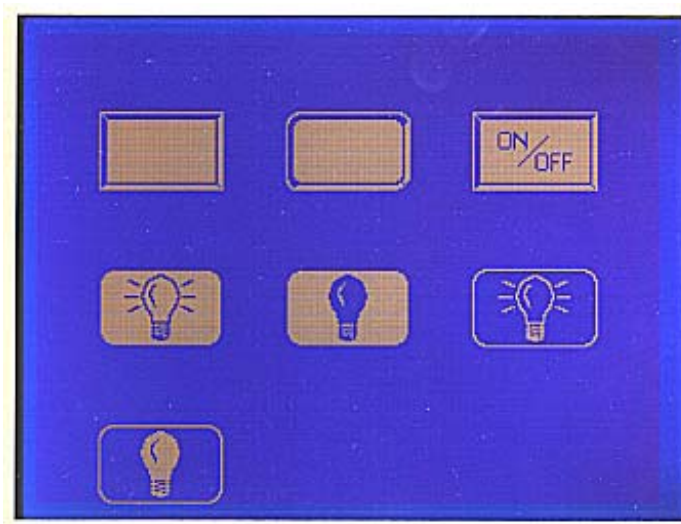


Fig. 82

The blank button in the upper left is not really a blank button but an icon made to look just like one. The button next to it is a variation made to look like a standard button but with rounded corners. You can put graphic designs or even hand created text like the “on/off” on the third button to create custom captions that the regular text captions cannot display. The four remaining icons were displayed using the “hidden border” checkbox option to make the original button outline invisible; the icon becomes the actual button outline. The ones displayed here are the full 64 x 40 dimensions of the button they cover, but they could be smaller or larger than the button they’re on. Note however that no matter how small or large the displayed icon may be, the actual active area where the button press will be detected still corresponds to the original button object’s size. Finally, creating a button with the hidden border box checked and assigning no icon or captions creates a totally invisible button, like the 8th button in the picture (..!). This can be useful to access hidden menus and other such applications, or to create “hot spots” under the very large icon of another button as our next example will show:

5.2.3 A full screen image and hidden buttons

This second example shows how you can create one large icon to create the impression of a bitmap screen image with invisible buttons hidden underneath the “hot spots” in the picture. An icon can be created that is larger than the button it is displayed on; its image will “spill over” the edges of the button but still be centered on it (although the actual “touch sensitive” area for that button remains the original button size, which is what makes this example possible). This time, we will create an image representing a home theater setup so that the user can easily find the buttons associated with each device. The first step is once again to start your favorite graphics editing program and layout your picture in such a way as to make the labeling of the hot spots coincide with the center of the hidden buttons that will respond to the screen presses. Here, we used Paint Shop Pro because it allows us to set up a grid with adjustable gridline positions, which we set to 32 horizontally and 40 vertically. This gives us a drawing grid very similar to the one we see in the touch screen editor. We can now start our screen design. For reasons you will see later, we chose to leave one horizontal row of touch squares out of this picture to use for a row of buttons along the bottom of the screen, so the bitmap here is actually 320 x 200 instead of 320 x 240 and is thus 5 touch squares high instead of 6 (Fig. 83). Notice how the various hand made captions aim to fall close to the center of the touch squares (except for the 4 buttons that make up the TV screen, we’ll use 2 x 1 buttons for these).

Next, we make some icons to put on buttons along the bottom of the screen to perform standard functions associated with playback devices, like “play”, “rewind”, etc. We’ll use the industry standard graphic symbols for these. Here is the “fast forward” button designed with a thin outline and with one less pixel along the bottom and right edges so that these buttons will appear separate when displayed side by side. Also note that the colors have been inverted because we want a light colored button with a dark symbol to display on our Leopard I (Fig 84). The image in Fig 83 was also color-inverted before downloading to the Leopard to get dark lines on a light background:

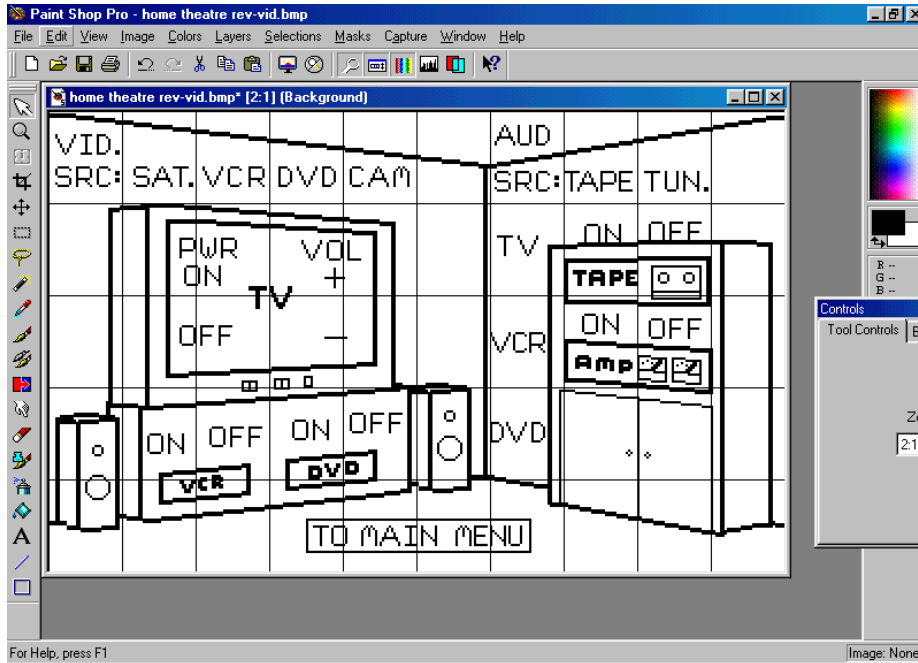


Fig. 83

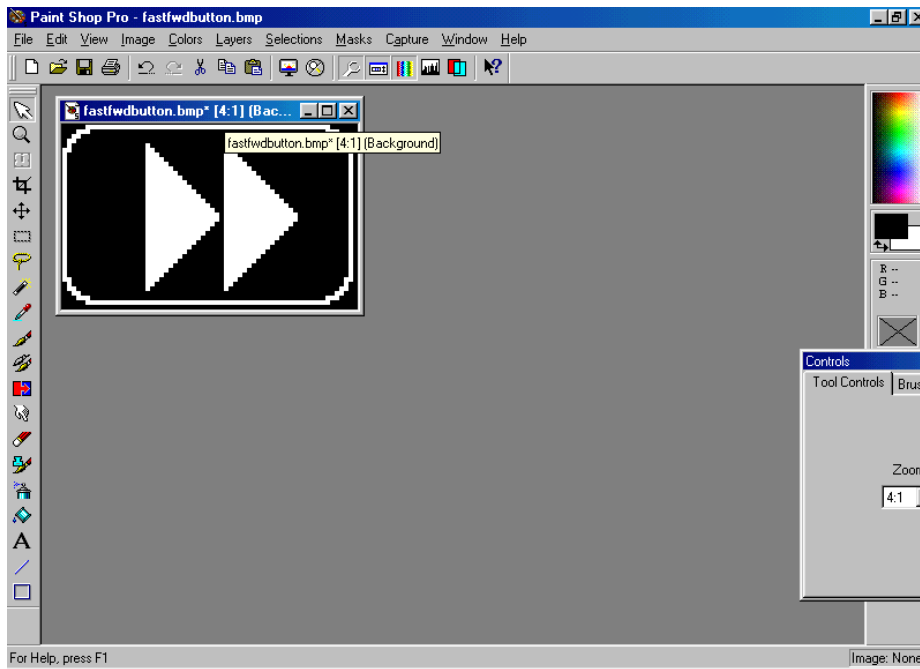


Fig. 84

Next, we create the touch screen layout over which the large image and the control buttons will be laid out (Fig. 85). All the buttons on this screen have the “hidden border” box checked and use no captions. All specify icon #0, which means display no icon, except for the 2 x 1 button to which the large icon image will be attached. This button is the one under the upper left corner of the “Enter Text” box in the picture.

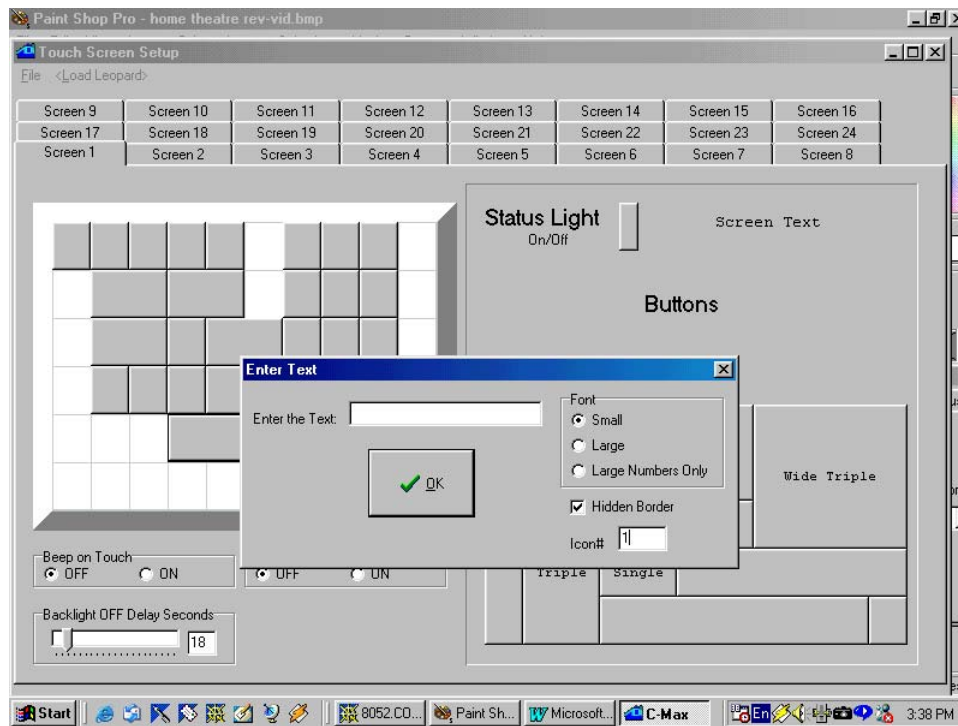


Fig. 85

This leaves the bottom row of touch squares on which we now add the playback control buttons. Each of these will also specify hidden borders and each one will be associated with the icon created for it (Fig. 86). It is also possible to actually display buttons or icons that overlap a large “spill over” icon (but not the button itself) on the screen. We could have done that here, but it is a waste of icon memory to do this if the image under the button(s) will not be visible anyway. If you do want to do this, make sure that the icon that is to appear “above” the other one has a higher object number within that screen in the touch screen editor. If the button is totally hidden (as in this example) and only serves as a hot spot for the large icon, then the object number is not important and can be either higher or lower, but if your intent is to have a visible button with an icon or caption appear over the large icon then you must observe this requirement.

You can see the object numbers by moving the mouse cursor over the button. A small information window will appear showing you the object number, hidden border flag, and icon number for that object. This is shown over the bottom right button in Fig. 86. The best way to be sure to have the lowest object number for the large background image is to define that button first when creating that screen. Even then, you must be careful because if you then go to another screen containing lower-numbered objects, delete one or more items, and then go back to your “large icon” screen and continue adding new objects; these will now have lower object numbers than even the first button you put on the screen because any freed object numbers are reused before new higher ones are assigned. If you have any doubts as to the object numbers of your various objects, just pass the mouse pointer over every one and note the numbers assigned to each one.

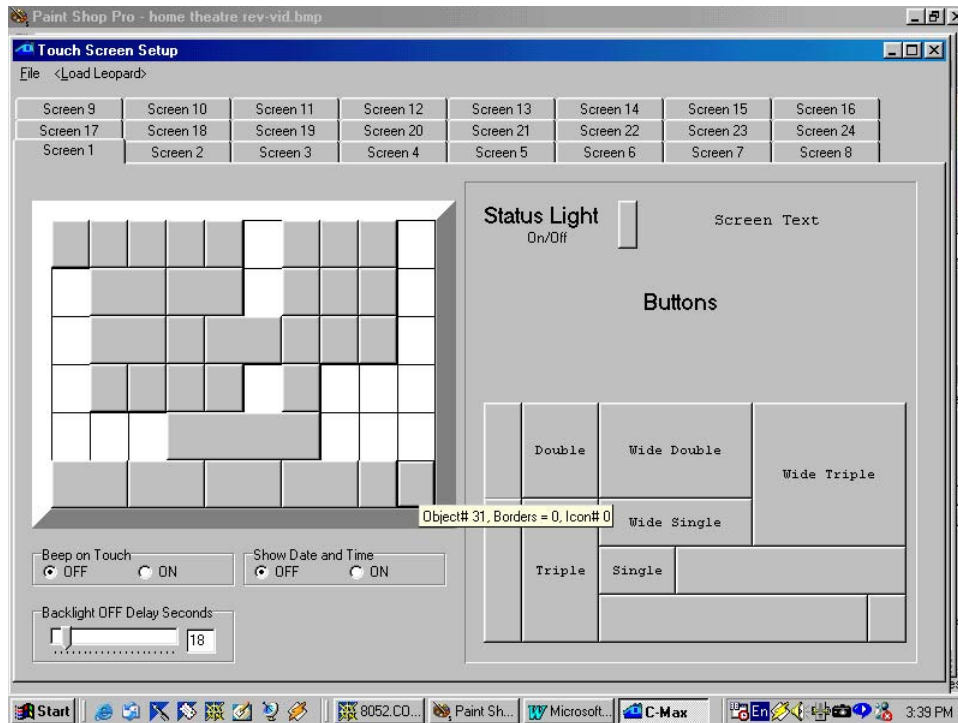


Fig. 86

Fig. 87 shows an actual picture of the final result on a Leopard II screen:

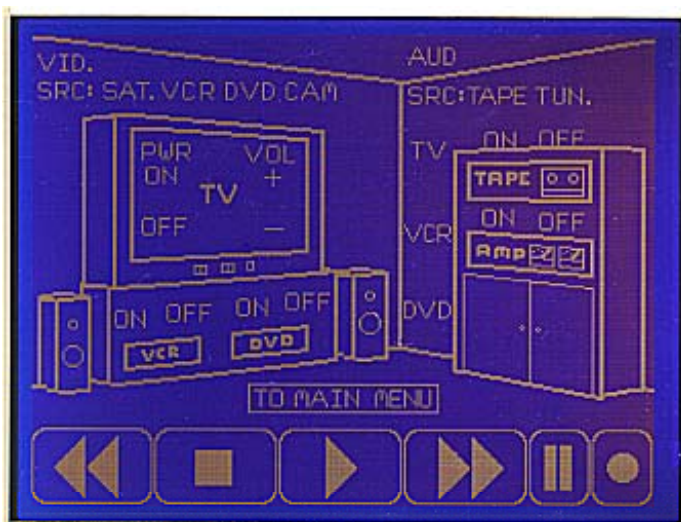


Fig. 87

5.2.4 Changing icons under program control

In this third and final example, we will show how you can have a running program actually change the displayed icon on buttons. You can use this to make dynamic screens where the illustration or hand created “captions” adapt to the choices the user makes as they happen. You can also use this feature to create graphical representations of data that you might prefer to see pictorially instead of just with numbers. The example we present here does just that: Imagine that you have an oil or water tank with a sensor that sends an analog signal between 0 and 5 volts to indicate the content’s level from empty to full respectively. You could always display a number indicating the level as a percentage of full, but you are so used to seeing liquid levels displayed on mechanical gauges that you would prefer a gauge-like display. Here, we will read the 0 to 5 volt output of the tank sensor using a SECU16 module input configured for analog mode and display an automobile gas-gauge like picture to indicate the level in the tank. We will do this by making six versions of the gas gauge icon, each one with the pointer in a different position. One of these icons will always be displayed, and will be selected depending on the analog value read from the SECU16 input.

First we choose our icon size. The gauge being vertical, we will make it 1 square wide by 3 squares high (32 x 120). We begin by designing the gauge scale itself (Fig. 88) and then the pointer (also Fig. 88) as a separate image. This way, the pointer can be pasted to the scale in various positions to simplify the procedure.

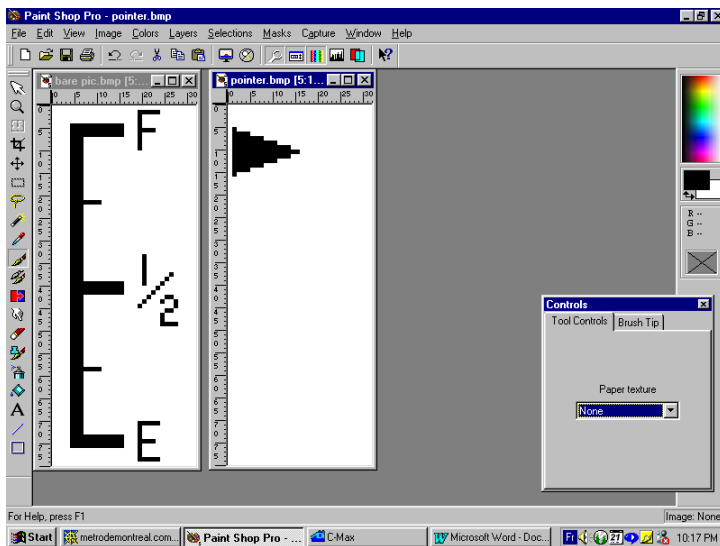


Fig. 88

With these two pictures we create the six pointer versions needed for our application (Fig, 89)

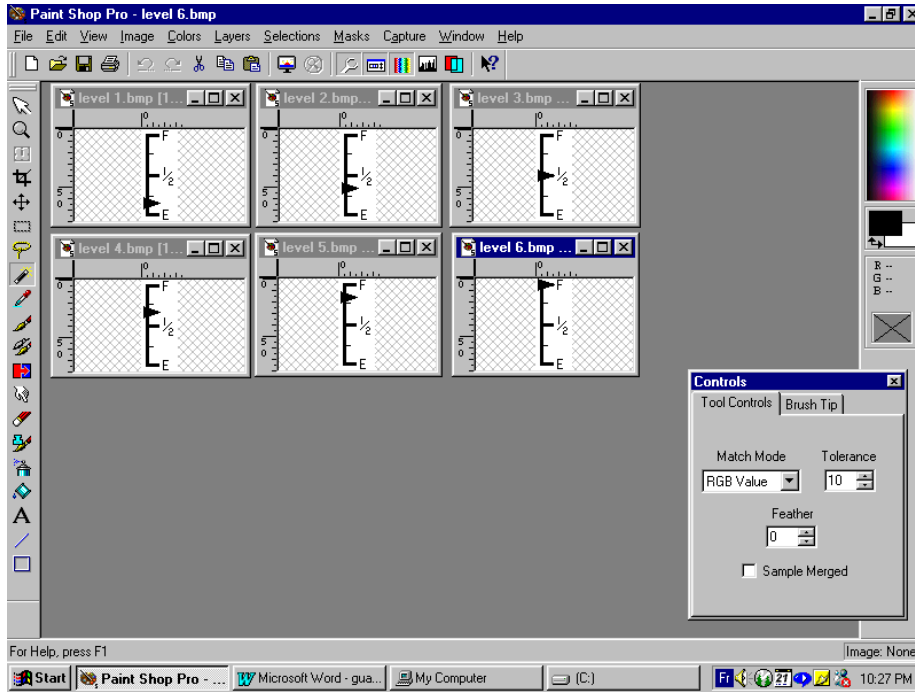


Fig 89

The six icons are then assigned in the System Map to six consecutive icon numbers (Fig. 90). Notice here that the first icon used happens to be icon# 5 and goes up to icon# 10 (any range can be used as long as the icons are in consecutive order and the illustrated levels rise with the higher icon numbers).

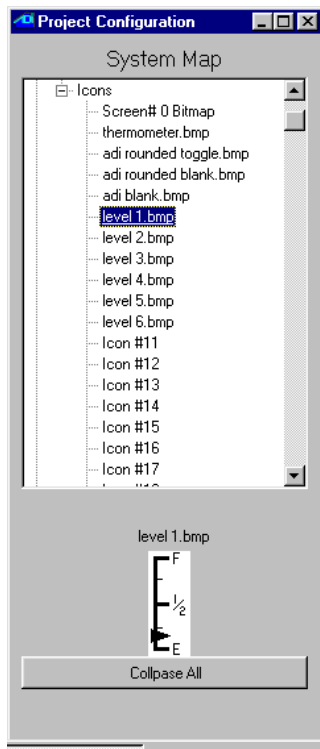


Fig. 90

The icons are loaded into the Leopard with the same procedure as in the other examples. A touch screen is then defined to display a 1 wide x 3 high button with no caption, hidden border is checked, and assigned icon #5 by default. In our example, this button is object #1. Finally, a screen text object is also added to identify the purpose of our gauge (Fig. 91). This screen definition is then also loaded into the Leopard.

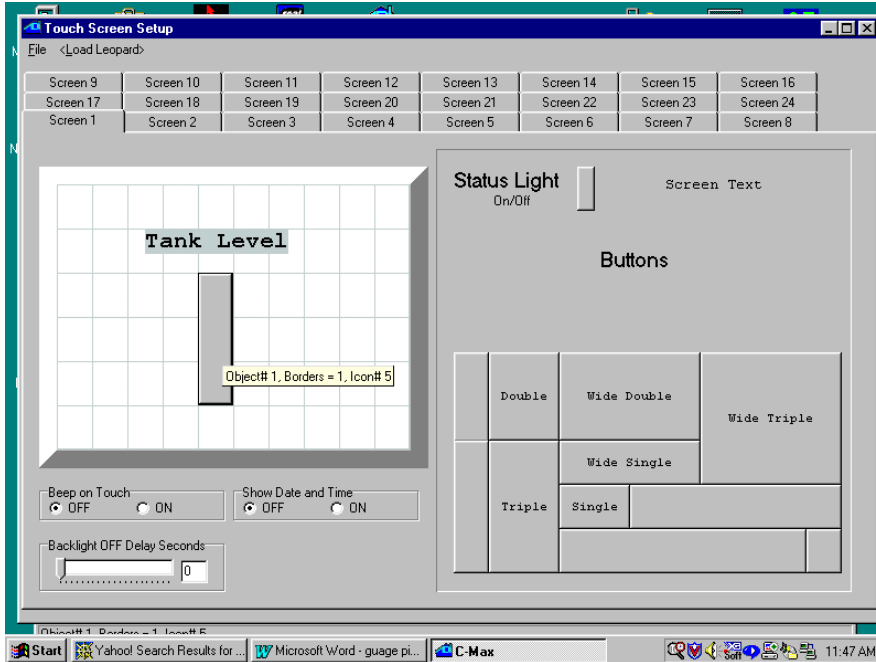


Fig. 91

Here is the program code used to read the SECU16 analog input and display the appropriate icon (Fig. 92)

```

0001 - IF Module #1 -SECU16 Tank-level-sensor is > 0 // read tank level voltage
0002 - THEN Load Data to: Variable #0 // store in var# 0
0003 - ELSE Load Data to: Variable #0 // whether > 0 or not
0004 - IF Variable #0 is > 0 // if > 0
0005 - THEN Variable #0 / 43 // then scale from 0 to 5
0006 - THEN Variable #0 + 5 // and add 5 for offset
0007 - THEN Touch Object #1 displays Icon# in Variable #0 // update displayed icon
0008 - ELSE Touch Object #1 displays Icon# 5 // if level 0 then icon #5
0009 - End Program //

```

Fig. 92

Lines #1 to #3 capture the analog value, which will range from 1 to 255 for signals of 0 to 5 volts respectively, into variable #0. The variables were not mapped to names here for clarity. Line #5 takes the value from 0 to 255 and divides it by 43 to obtain a value from 0 to 5, since our gauge can display six different levels. Thus input values from 215 to 255 will show the “full” icon, input values from 172 to 214 will show the next level down and so on. Line #6 adds 5 to our scaled value because our icon numbers for this example run from 5 to 10. Line #7 then displays the calculated icon number on the touch button. Line #8 exists because if line# 4 tests false, then the analog input is indeed zero and we want to show the lowest level icon in that case. Fig. 93 shows an actual picture of a Leopard running this program.

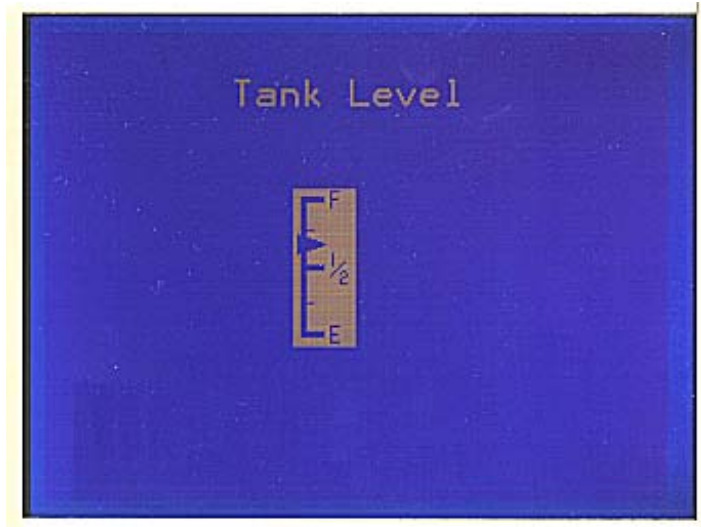


Fig. 93

As these three examples have demonstrated, you can make highly customized screens using the icon display capabilities and switching instruction in C-Max 2.0

5.3 A Sample C-Max Application, Heat/Cool Thermostat

Introduction

The purpose of this application note is to demonstrate how C-Max logic tests can be used to create home automation control programs. The intent is not to present a “ready made” solution for a particular problem, but to show how the ideas for how you want to do things can be translated into actual program code.

A simple thermostat for heating

For most people, the first example that comes to mind when thinking about a user settable setpoint application is a thermostat; so let’s use it. This sample application supposes that the programmer wants to use a Leopard II controller with a temperature Bobcat to turn on a heating appliance when the temperature reading from the Bobcat goes below a setpoint that can be selected by the touch screen user. The Leopard II screen will have two buttons defined; one for raising and one for lowering the setpoint. Additionally, the touch screen will display the setpoint and the current temperature. Finally, the buttons and displayed temperatures will have appropriate text labels. An Ocelot user will obviously not have the same user interface to choose the setpoint, but alternate methods (such as X-10 signals) can be used, and the actual control logic will work just as well with either controller.

These examples make the following assumptions: The controller is module 0 and the Temperature Bobcat is module 1. The heat will be turned on by sending an X-10 “N/1 N/On” command pair (with the “Quick ON” controller commands) and conversely be turned off with a N/1 N/Off command pair. Variables and timers are assigned from 0 on up as needed. Of course, feel free to substitute other module numbers and Variable/Timer numbers as you see fit, if you are trying or using these examples in an existing system.

User Interface

This section assumes that the programmer is familiar with setting up Leopard screens. If not, review the procedures earlier in the manual. Create the first button with a “-“ caption and the second one with a “+”. Also note the “object #” displayed when the pointer is positioned over the buttons you just created, these become the button numbers used when programming. For this example, the “-“ button is object #1 and the “+” button is object #2. Create the two text fields that will display the setpoint and the actual temperature by using the “Large Numbers Only” font setting when adding these text fields. The “%3d” caption allows for displaying up to 3 digits. Also be sure to link the setpoint to the right Variables by clicking “Variable Data in Text” in the field properties. In this example, the setpoint will be in Variable #0 and the actual temperature in Variable #1. Finally, create the text labels to properly identify the temperatures. Feel free to position the various buttons and text fields to your liking; this is what a programmable controller is all about! Below is a picture of the screen created for this example (fig. 94):

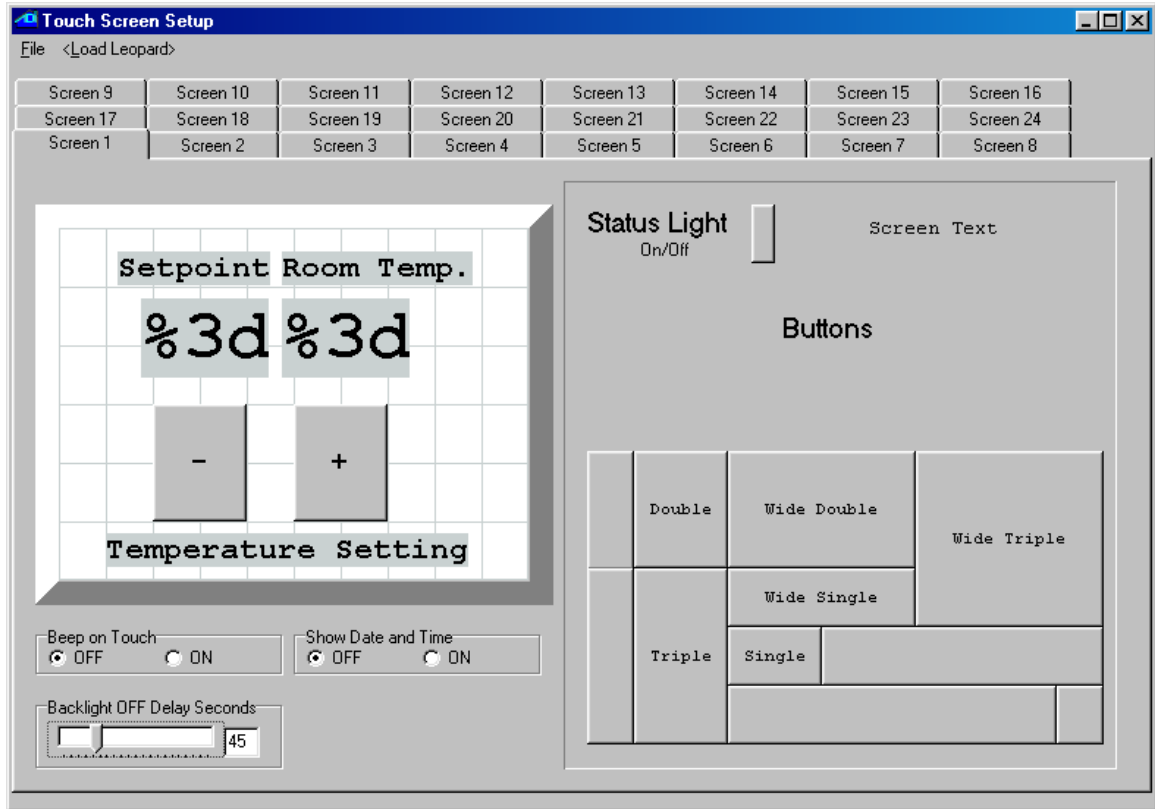


Fig. 94

Program Logic

The first thing we want to do is convert our button presses on the touch screen to a setpoint stored in Variable #0. We need two program segments to do this, one for the “-” button and one for the “+” button (fig. 95).

User interface logic

```

0001 - IF Touch Object #1, Button Leopard is pressed
0002 - THEN Variable #0 - 1
0003 - IF Touch Object #2, Button Leopard is pressed
0004 - THEN Variable #0 + 1

```

Fig. 95

We then need to create two more program segments to actually read the temperature and compare it against the setpoint. One segment looks for the temperature being below the setpoint and will turn on the heat if that is the case, the other segment tests for the setpoint having been exceeded so that the heat can be turned off (fig 96).

Control logic (version 1)

```

0005 - IF Module #1 -BOBCAT-T becomes < Variable #0
0006 - THEN X-10 House N / Unit 1, Turn ON
0007 - IF Module #1 -BOBCAT-T becomes > Variable #0
0008 - THEN X-10 House N / Unit 1, Turn OFF

```

Fig. 96

That's it. We now have a simple heating thermostat. A few things to note here. First: notice how we look for the temperature to have *exceeded* the setpoint instead for just reaching it? This was done to avoid short, inefficient heating cycles by having the program look for a full 1 degree increase in temperature before stopping the heater, and for a 1 degree drop before turning it back on (in a control system, this is called *hysteresis*). A razor-sharp detection precision would not be in our best interest here. Second: we used the "becomes" versions for the compare instructions because a simple "<" test would have been sending X-10 commands continuously as long as the test was true. By using single triggering instructions like these, we are free to send single shot events like X-10 commands or using continuous state commands like turning on SECU16 or RLY8/XA relays. Now, as good and functional as this may look, there is one slight potential shortcoming that could be improved upon. If the heat is turned on and the temperature rises *to* the setpoint, but then dips below it again because of air movement or someone opening a door, *another* X-10 "on" command will be sent. This means that unnecessary repeated On or Off commands will still occur if there are slight variations on subsequent temperature readings (in digital systems, this is sometimes called *digit bobble*). To avoid these extraneous X-10 signals, the first solution that comes to mind is to use a variable as a flag to know that if the heat has already been turned on, there is no need to do it again until the setpoint is exceeded. Here thus is a modified version of the control logic, using Variable #2 as a flag (fig. 97):

Control logic (version 2)

```

0005 - IF Module #1    -BOBCAT-T becomes < Variable #0
0006 -   AND Variable #2 is = 0
0007 -   THEN X-10 House N / Unit 1, Turn ON
0008 -   THEN Variable #2 = 1
0009 - IF Module #1    -BOBCAT-T becomes > Variable #0
0010 -   AND Variable #2 is = 1
0011 -   THEN X-10 House N / Unit 1, Turn OFF
0012 -   THEN Variable #2 = 0

```

Fig. 97

That will work and could be left this way without any further changes, but there *is* a better way to do this with C-Max. The controller maintains an internal status table of all 256 X-10 addresses and of all module (ie:SECU16) outputs. This means that if it turns on an X-10 module or "sees" and external command coming from any X-10 transmitter to turn a module on or off, it will update its table with the appropriate status. Since our own program is turning the heater on or off, we can consult the status table to see if the unit is already on or off instead of using a Variable to "remember" if this was done. Rewritten to use this, our control logic becomes (fig. 98):

Control logic (version 3)

```

0005 - IF Module #1    -BOBCAT-T becomes < Variable #0
0006 -   AND X-10 House N / Unit 1, Is OFF
0007 -   THEN X-10 House N / Unit 1, Turn ON
0008 - IF Module #1    -BOBCAT-T becomes > Variable #0
0009 -   AND X-10 House N / Unit 1, Is ON
0010 -   THEN X-10 House N / Unit 1, Turn OFF

```

Fig. 98

Now, not only is the program easier to understand, but we have "saved" a Variable and also eliminated the two "Then" statements used to update the Variable.

Initialization

We bring up the subject of initialization at this point because now is a good time to look at some potential problems with our control logic and with the controller in general. The question is: what do we do

when the power is first applied to the controller, or if the power fails and comes back on? Suppose for example that the power is interrupted briefly and the heater was left ON. Many X-10 appliance modules have a latching relay that will cause *it* to be ON when the power comes back. The module also has no method to tell the controller that it is ON. The controller will reboot and start with a blank status table (everything assumed OFF). Also suppose that the ambient temperature was warm enough that the setpoint is exceeded (or at least reached) at the time the power comes back. Looking at our program logic, the second segment can *never* test true because the status table assumes the module is OFF. So the heater keeps heating and the controller cannot do anything to stop it! For this reason, and others (like wanting an initial setpoint to be other than zero!) an initialization sequence may be desirable and even necessary. Assuming that we are using variables below that specified in Parameter 22 of the controller (variables at and above this value are retained after a power interruption), you can create a sequence of commands that will be executed only once, right after a power on. Use a spare Variable and look for it being equal to zero, and then at the end of your initialization sequence set it to a non-zero value so that it is not executed again. For our example, lets set an initial setpoint of 72 degrees, and lets turn OFF the heater if the ambient temperature is above the setpoint. Using spare Variable #9, our code is (fig 99):

Initialization

```

0011 - IF Variable #9 is = 0
0012 -     THEN Variable #0 = 72
0013 - IF Variable #9 is = 0
0014 -     AND Module #1 -BOBCAT-T is > Variable #0
0015 -     THEN X-10 House N / Unit 1, Turn OFF
0016 - IF Variable #9 is = 0
0017 -     THEN Variable #9 = 1

```

Fig. 99

If you wanted a non-volatile setpoint, then you could use a Variable for it whose number is greater than or equal to the number in Parameter 22 of your controller.

A full featured thermostat for heating and cooling

The first application above gave us a feel for how a user interface and control logic can create a workable thermostat. However functional, it is still rather primitive compared to commercially built units. After all, how many thermostats have you seen that let you raise the setpoint all the way up to 65535! The following application takes the same principles as the first one and brings it up to standards similar to those of commercial units. Specifically, it adds the following features:

- Two stage heating and single stage cooling (if the heating setpoint is not reached after 15 minutes of having turned on the heat, a second heating source is turned on).
- A high setpoint limit of 90 degrees and a low one of 50, with an error “beep” if the user tries to exceed these values.
- A minimum spread of 2 degrees between the heat and cool setpoints (eg: you can’t raise the heat setpoint above 72 if the cooling setpoint is set to 74).
- A 3 second delay before turning on heating or cooling whenever the setpoint(s) are being modified by the user. This is to avoid damaging the heating/cooling equipment caused by powering it on and then off quickly.
- Indicator lights on the Leopard screen to show which heating or cooling stages are turned on.

User Interface

Here is the new, modified Leopard screen to allow the setting of the heating and cooling setpoints, as well as showing the indicator lights showing current status (fig. 100).

This example screen has the following characteristics:

- Heat “-“ button is object #1
- Heat “+“ button is object #2
- Cool “-“ button is object #8
- Cool “+“ button is object #9

- Heat setpoint display is linked to Variable #0
- Cool setpoint display is linked to Variable #1
- Room temperature display is linked to Variable #4

- Heat status light is linked to X-10 status for N/1
- Aux Heat status light is linked to X-10 status for N/3
- Cool status light is linked to X-10 status for N/2

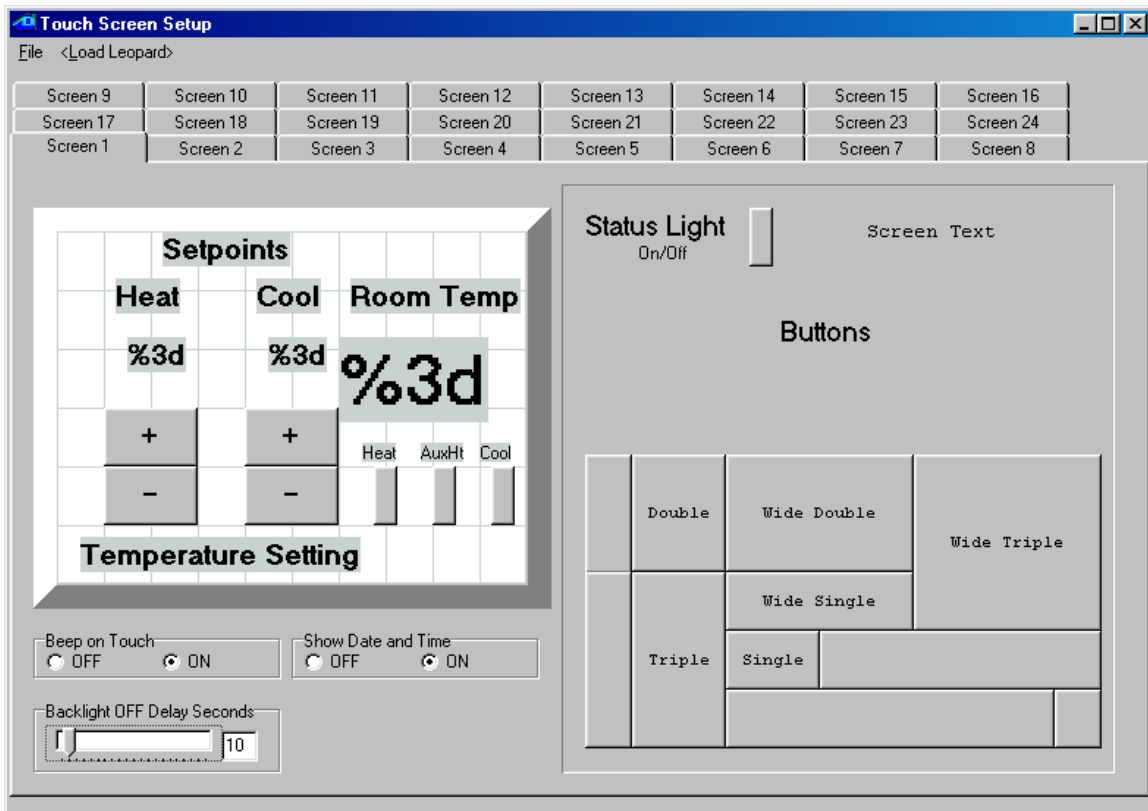


Fig. 100

Program Logic

For this application, the entire program will be shown first, and we will then analyze the various program segments, explaining each one by referring to line numbers. This will allow you to see each function in its logical context and make the whole program easier to understand. Here is the program listing (fig. 101):

initialization

```
0001 - IF Variable #0 is = 0 // if heat setpt. is 0
0002 - THEN Variable #0 = 72 // then set heat to 72
0003 - THEN Variable #1 = 75 // then set cool to 75
0004 - THEN X-10 House 0 / Unit 1, Turn OFF // turn off heat
0005 - THEN X-10 House 0 / Unit 2, Turn OFF // turn off cooling
0006 - THEN X-10 House 0 / Unit 3, Turn OFF // turn off aux heat
0007 - IF Touch Anywhere Button #0, Button Leopard is pressed // if on blank screen
0008 - THEN Variable# 63 / Screen# = 1 // go to screen 1
```

continuously calculate spread...

```
0009 - IF Variable #2 is = Variable #2 // dummy "if" (always)
0010 - THEN Variable #2 = Variable #1 // take the cool setpt
0011 - THEN Variable #2 - Variable #0 // and subtr heat setpt
```

...and display current temperature

```
0012- THEN Variable #3 = Data for Module# 1 // get temp. from bobcat
0013 - THEN Variable #3 - 100 // subtract 100
0014 - THEN Variable #4 = Variable #3 // put in display var.
```

user-error beep section

```
0015 - IF Touch Object #2, Button Leopard is pressed // if heat + button
0016 - OR Touch Object #8, Button Leopard is pressed // or cool - button
0017 - AND Variable #2 is < 3 // and setpt at min spread
0018 - THEN Variable# 61 / Beep-Variable = 1 // then beep user!
0019 - IF Touch Object #1, Button Leopard is pressed // if heat - button
0020 - AND Variable #0 is = 50 // and at low setp lim
0021 - THEN Variable# 61 / Beep-Variable = 1 // then beep user!
0022 - IF Touch Object #9, Button Leopard is pressed // if cool + button
0023 - AND Variable #1 is = 90 // and at hi setp lim
0024 - THEN Variable# 61 / Beep-Variable = 1 // then beep user!
```

heat setpoint setting

```
0025 - IF Touch Object #2, Button Leopard is pressed // if heat + button pressed
0026 - AND Variable #0 is < 90 // and setpoint is below 90
0027 - AND Variable #2 is > 2 // and spread is > than 2
0028 - THEN Variable #0 + 1 // increment heat setpoint
0029 - THEN Timer #0 = 1 // and start the setp timer
0030 - IF Touch Object #1, Button Leopard is pressed // if heat - button pressed
0031 - AND Variable #0 is > 50 // and setpoint is above 50
0032 - THEN Variable #0 - 1 // decrement heat setpoint
0033 - THEN Timer #0 = 1 // and start the setp timer
```

cool setpoint setting

```
0034 - IF Touch Object #9, Button Leopard is pressed // if cool + button pressed
0035 - AND Variable #1 is < 90 // and setpoint is below 90
0036 - THEN Variable #1 + 1 // increment cool setpoint
0037 - THEN Timer #0 = 1 // and start the setp timer
0038 - IF Touch Object #8, Button Leopard is pressed // if cool - button pressed
0039 - AND Variable #1 is > 50 // and setpoint is above 50
0040 - AND Variable #2 is > 2 // and spread is > than 2
0041 - THEN Variable #1 - 1 // decrement cool setpoint
0042 - THEN Timer #0 = 1 // and start the setp timer
```

setpoint timer expiry test

```
0043 - IF Timer #0 becomes > 4 // if setp timer expires
0044 - THEN Timer #0 = 0 // then stop it.
```

turn ON heat

```
0045 - IF Module #1 -BOBCAT-T becomes < Variable #0 // if temp is below setp
0046 - AND Timer #0 is = 0 // and not in setp timeout
0047 - AND X-10 House 0 / Unit 1, Is OFF // and heat is off
0048 - THEN X-10 House 0 / Unit 1, Turn ON // then turn if on
0049 - THEN Timer #1 = 1 // start aux heat timer
0050 - IF Module #1 -BOBCAT-T is < Variable #0 // if temp is below setp
0051 - AND Timer #0 becomes = 0 // and setp timer expires
0052 - AND X-10 House 0 / Unit 1, Is OFF // and heat is off
0053 - THEN X-10 House 0 / Unit 1, Turn ON // then turn if on
0054 - THEN Timer #1 = 1 // start aux heat timer
```

turn OFF heat

```
0055 - IF Module #1 -BOBCAT-T becomes > Variable #0 // if setpoint is exceeded
0056 - AND Timer #0 is = 0 // and not in setp timeout
0057 - AND X-10 House 0 / Unit 1, Is ON // and heat is already on
0058 - THEN X-10 House 0 / Unit 1, Turn OFF // then turn it off
0059 - THEN X-10 House 0 / Unit 3, Turn OFF // and turn aux heat off
0060 - THEN Timer #1 = 0 // stop the aux heat timer
0061 - IF Module #1 -BOBCAT-T is > Variable #0 // if setp is exceeded
0062 - AND Timer #0 becomes = 0 // and setp timer expires
0063 - AND X-10 House 0 / Unit 1, Is ON // and heat is already on
0064 - THEN X-10 House 0 / Unit 1, Turn OFF // then turn it off
0065 - THEN X-10 House 0 / Unit 3, Turn OFF // and turn aux heat off
0066 - THEN Timer #1 = 0 // and stop aux heat timer
```

Aux Heat timer expiry test

```
0067 - IF Timer #1 becomes > 900 // if aux heat timer expires (15 mins.)
0068 - AND X-10 House 0 / Unit 1, Is ON // and 1st stage heat is on
0069 - THEN X-10 House 0 / Unit 3, Turn ON // then turn on aux heat
0070 - THEN Timer #1 = 0 // stop the aux heat timer
```

turn ON cooling

```
0071 - IF Module #1 -BOBCAT-T becomes > Variable #1 // if temp is above setp
0072 - AND Timer #0 is = 0 // and not in setp timeout
0073 - AND X-10 House 0 / Unit 2, Is OFF // and cooling is off
0074 - THEN X-10 House 0 / Unit 2, Turn ON // then turn if on
0075 - IF Module #1 -BOBCAT-T is > Variable #1 // if temp is above setp
0076 - AND Timer #0 becomes = 0 // and setp timer expires
0077 - AND X-10 House 0 / Unit 2, Is OFF // and cooling is off
0078 - THEN X-10 House 0 / Unit 2, Turn ON // then turn if on
```

turn OFF cooling

```
0079 - IF Module #1 -BOBCAT-T becomes < Variable #1 // if setp is exceeded
0080 - AND Timer #0 is = 0 // and not in setp timeout
0081 - AND X-10 House 0 / Unit 2, Is ON // and cooling is on
0082 - THEN X-10 House 0 / Unit 2, Turn OFF // then turn it off
0083 - IF Module #1 -BOBCAT-T is < Variable #1 // if setp is exceeded
0084 - AND Timer #0 becomes = 0 // and setp timer expires
0085 - AND X-10 House 0 / Unit 2, Is ON // and cooling is on
0086 - THEN X-10 House 0 / Unit 2, Turn OFF // then turn it off
0087 - End Program //
```

Fig. 101

Explanatory Notes:

Beginning with the initialization section, note how we can use Variable# 0 as an initialization flag this time. This is because our low and high setpoint limits prevent us from ever setting this variable (the heat setpoint) to zero again, so the only time it can be zero is upon power up. The segment that looks for button 255 being pressed (line 7) is looking for the first time the touch screen is pressed, and then displays our thermostat screen.

Line 9 might seem a bit odd at first. This is just a way of having the following five “Thens” executed on *every* pass. Line 12 is a good example of the usage of extended *variables*. These variables refer to the module number of a bobcat + 128. In our case, the Leopard is module #0 and the temperature bobcat is module #1 (as you can deduce from looking at line 45). This allows you to *copy* the bobcat’s temperature reading into a variable, instead of just comparing against it like in line 45. We need to do this because we want to do some math manipulation on it before displaying it (and you cannot link a Leopard screen object to a variable greater than 127 anyway). Also note that this copies the bobcat’s *raw* data value, which is 100 higher than the temperature in degrees F. (this is to allow reading temperatures below zero without running into negative numbers, which are not implemented as such in C-Max). Be careful with this distinction if you mix bobcat compares and extended variables in your program, like in this application. The final calculated temperature is placed in Variable #4 for screen display.

The “user-error beep” is watching for the user trying to either exceed a high or low setpoint limit, or trying to set either limit less than 2 degrees from the other. Note that there is no need to look for the user trying to raise the heat setpoint to the high limit or lower the cool setpoint to the low limit, because trying this will be caught by the minimum spread instead (ie: if cooling *is* set to the high limit of 90, the minimum 2 degree spread rule will prevent the heat setpoint from going any higher than 88 automatically). Notice that these routines only look for the user *trying* to press buttons that will violate the limits; the actual enforcing of these limits is done in the setpoint setting routines, which come next. Lines 18,21 and 24 make the Leopard “beep” for 1 second, by copying a “1” into Variable #61, which is specially designated by design to beep for the number of seconds copied into it. Another very important thing to understand is that *any* single triggered events, like X-10 commands, screen button presses, etc., are valid for one *entire* pass through the program, so lines 15 and 25 are actually detecting the *same* button press! Single events are thus NOT “consumed” by the first program line that detect them, which would otherwise oblige us to flag the single event into a variable and then compare against the flag in other segments where we need to detect the event, and finally we would need to reset the flag near the end of our pass. This would burden our program code considerably! Lastly, single triggered events *always* get detected *between* passes through the program, so lower line-numbered statements will always see them *first*. This is important to know if you have order-sensitive code. We have a good example of this right here: lines 22 and 23 produce the error beep if the high limit of 90 degrees is already reached and the user tries to exceed it. Lines 34 and 35 look to allow setting the setpoint all the way up to 90 if it not reached *yet*. If you were to invert the two routines, you would get an error beep when the cool setpoint is set exactly to 90 degrees, because the line 22,23 segment would then test true later in the pass, *after* the setpoint was set to 90.

The heat setpoint setting section is quite straightforward and needs little explanation; verify if a limit or spread is going to be violated by the change and if not, do it. The cool setpoint setting section is virtually identical except that it modifies Variable #1 instead of Variable #0. If you carefully followed the description of the error-beep section in the last paragraph, you will notice that lines 26 and 39 are actually superfluous, since the checking of the setpoint spread will never allow these limits to be reached (these lines were deliberately left in the code to illustrate this...). It also shows how you can quickly code similar segments by copying and pasting code and then editing the differences. The other item of interest in the setpoint setting routines is Timer #0 started by lines 29, 33, 37 and 42. This means that any setpoint button presses start (or restart) the timer at 1. This is used in later program sections to prevent any heating or cooling action to be started until we are sure the user has finished modifying the setpoints (to prevent damaging equipment, especially compressors, with very short on or off cycles caused by the user modifying the temperature settings). The setpoint timer expiry section simply looks for the setpoint timer to have exceeded 4 seconds of continuous incrementing, which means the user has not pressed a button for at least 4 seconds, and then stops the timer. The way this is used will become clear in the next sections.

The four routines for turning the heat on or off, or the cooling on or off are all quite similar and also illustrate a very important point about steady state and single triggering events. First of all, you'll notice that each of those program sections have *two* sets of test statements for doing the same thing (if we had parenthesis, these could have been coded as IF (line 45 AND line 46 AND line 47) OR (line 50 AND line 51 AND line 52) THEN lines 48 and 49... this is equivalent here). The reason for the two sets of tests is that to turn on the heat (to use that one as an example), either: 1-The temperature drops below the setpoint due to the room getting colder, with the setpoint timer not running because the setpoint has been set a while ago (lines 45 and 46) or : 2- the temperature was fine, but the user has decided to raise the setpoint, which means the room temperature might have been below the *new* setpoint for some time, but now we have to wait for the setpoint timer to expire before turning on the heat (lines 50 and 51). The lesson here is that you will want *one and only one* single triggering event to cause a single shot action like sending an X-10 command. If you were to look for Timer# 0 *becoming* 0 AND the setpoint *becoming* less than Variable #0, these two single triggering events would have to occur during the *same pass* through the program for the segment to test true, which is almost impossible. Conversely, to look for both conditions simply being true, you would be sending X-10 commands continuously as long as the setpoint is not yet exceeded by the heating in the room. You want to look for a single triggering event plus any other required conditions being already true. Two possible single triggering events thus means two sets of test conditions. This is the reason for all those “becomes” versions of the IF statements. Without them, programming would be a nightmare, with the need once again to use variables as flags to accomplish what we can do here with a lot less code. Make sure you understand this concept thoroughly and you will find your programs much easier to design. You must “think through” all the possible conditions and code them appropriately.

Like in the first simple heat-only thermostat example, we look at the existing state of the X-10 module to determine if an on or off command is needed. Note that all four program segments are very similar, with the only differences being the variables being compared and the X-10 module addresses being turned on or off. The only real difference is that the heating mode also starts Timer #1 whenever the heat is turned on. This timer is used to determine if the heat has been turned on for 15 continuous minutes (lines 67 and 68). If that is the case, an auxiliary heating source is turned on, as a second stage to provide faster heating. Note that lines 59 and 65 in the turn off heat section always send an off signal to turn off auxiliary when the setpoint is finally exceeded, whether auxiliary heat had been turned on or not (this occurs rarely enough to not warrant a separate test section, but it could be done if needed). It is easy to see how auxiliary cooling could be implemented if required.

5.4 Auto Addressing your ADICON™ 2500 Series

Auto addressing sets the addresses on all the modules connected in the daisy chain.

IMPORTANT!

If you plan to use more than 1 Ocelot™ or Leopard™, disconnect (or remove power from) each slave Ocelot™ or Leopard™ before auto addressing. Slave Ocelots™ and Leopards™ cannot be auto addressed.

1) Remove the covers from the new ADICON™ module if needed to access the Auto Address button.

It is not necessary to remove the cover from the Ocelot™, Bobcat™ or SPEAKEZ™ modules.

2) Connect the ADICON™ 2500 Series daisy chain and apply power to each module as described in each module's manual.

3) Start C-Max. Click **Comms** with your left mouse button. A window will appear. Click **Attach to Controller** with your left mouse button. The Controller Access window will appear.

4) Click **Controller Utility** with your left mouse button. A window will appear.

5) Click Auto Address Modules with your left mouse button. The following window will appear (fig 102):

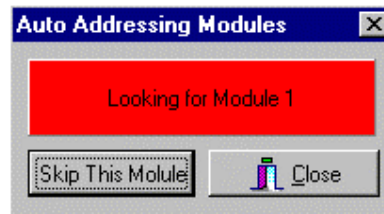


Fig. 102

The Active light on each module will flash rapidly.

6) Wait 30 seconds. Go to each module and press and release the address button.

Make certain that only one button is pushed at a time. Each time an address button is pushed, the Active light on the module will stop flashing rapidly and flash slowly. The number after "Looking for Module" will increment each time a button is pushed.

Note: Be sure to write down which modules are programmed as Module 1, Module 2 etc. This will help you when you start writing your program.

7) Once each module has been addressed, click Close with the left mouse button. The following windows will appear (fig 103): Click OK with your left mouse button.

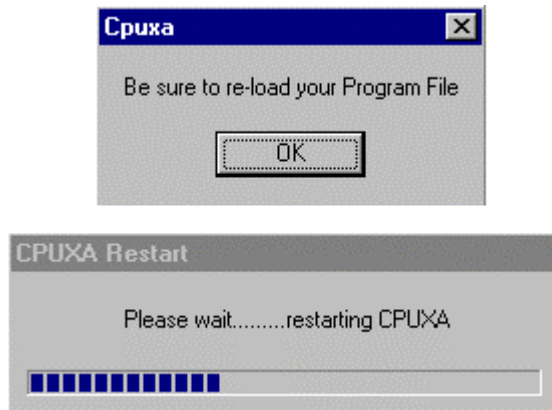


Fig. 103

If you had a program loaded in the Ocelot™ or Leopard™ you will need to reload it.

8) Exit then re-enter Controller Access screen.

When you enter the Controller Access screen, the grid to the right will show each module that is connected to the Ocelot or Leopard. Each module will report a type and version as shown below (fig 104):

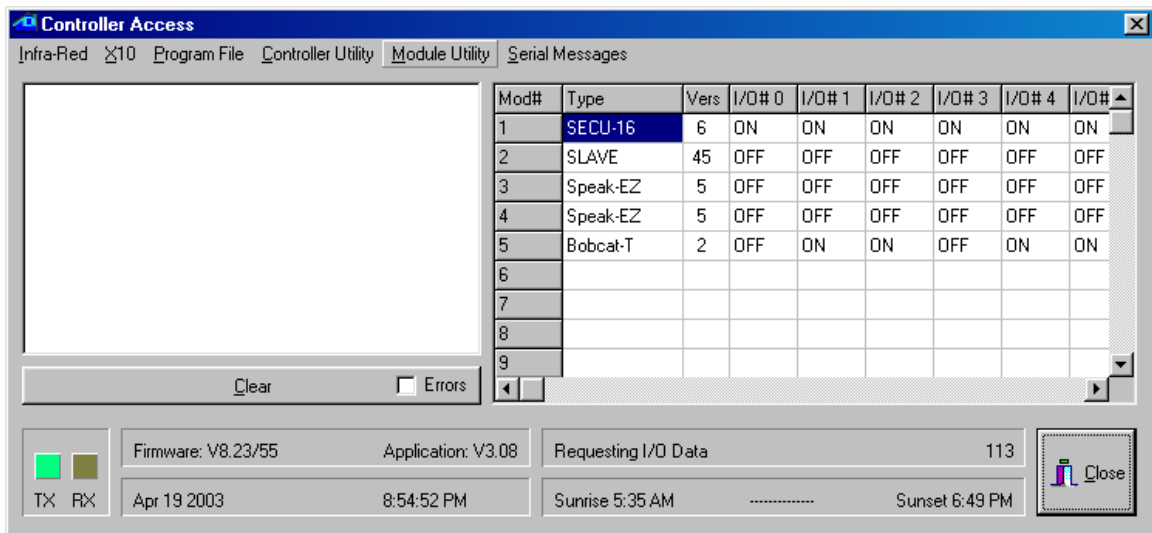


Fig. 104

The Type indicates what module is at the specified address indicated under MOD#. Version (Vers on the controller access screen) is the software version stored in each module. As features are added to modules, this software version is updated by replacing a chip inside the module.

5.4.1 Adding a new module to an existing installation

You can add a new module one of two ways:

- You can connect the new module to the current installation as described in Section 1 and re-auto address the entire system
- You can independently address the new module. To independently address the new module, do the following:

1) Launch C-Max, Click **Comms** and **Attach to Controller**. Write down what addresses are currently being used and determine what address you want the new module (or modules) to be. Each module must have a unique address.

2) Disconnect the existing wires from COMS A and B on the Ocelot™ or Leopard™.

3) Connect the new module to the Ocelot™ or Leopard™ as shown in the module manuals with no other modules connected.

4) Remove the covers from the new ADICON™ module if needed to access the Auto Address button.

It is not necessary to remove the cover from the Ocelot™, Bobcat™ or SPEAKEZ™ modules.

5) Start C-Max. Click **Comms** with your left mouse button. A window will appear. Click **Attach to Controller** with your left mouse button. The **Controller Access** window will appear.

6) Click **Controller Utility** with your left mouse button. A window will appear (fig. 105).

7) Click Auto Address Modules with your left mouse button. The following window will appear:

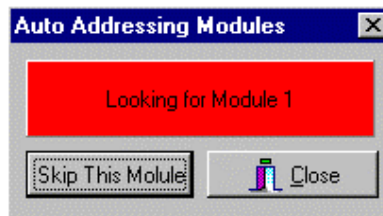


Fig. 105

The Active light on your module will flash rapidly.

8) Press the “Skip This Module” button until the desired address appears next to Looking for Module in the Auto Addressing Modules window that you determined in Step 1. Make sure that the address you choose is not the same as an address on an existing module.

9) Press the Auto Address button on the new module. The number after “Looking for Module” will increment once. If the number does not increment, wait 10 seconds and press the button again. The Active light will stop flashing.

10) Exit the **Controller Access** screen.

- 11) Repeat Steps 2-10 for each new module you want to add. When finished click the Close button.
- 12) Remove power from and Ocelot™ and Module and reconnect the new module in the daisy chain in the desired location and reconnect the other modules to the Ocelot™ or Leopard™. Reapply power to the modules first, then the Ocelot™ or Leopard™.
- 13) Re-enter the **Controller Access** screen. You current and new modules will be displayed in the **Controller Access** screen.

Note: If the new module is addressed using the second method then CPU parameter 3 must be changed to show the highest module number. For example, if your system has 6 modules and you replace module 2, then after auto addressing, CPU parameter must be changed to 6.

5.5 Using Expansion Modules

In this application note on programming, we will look at using some of the ADI expansion modules. Several different models of expansion modules are available. There are temperature, humidity and light sensors, relay modules, a sound module, IR module, and some modules with input capabilities too. One of the most widely used modules is the SECU16, which offers 8 output relays as well as 8 inputs that can be configured and read in various ways. It is this module that we will use for our programming examples.

When connecting a home automation (HA) controller to real-world devices, it quickly becomes obvious that X10 alone will not cover everything if we want to progress beyond turning lights on and off. Many devices do not offer the capability to control them with other than their built-in controls, but with some knowledge of electricity and electronics it is often possible to buy or make an interfacing device that will provide some remote input and/or output capabilities. Often the interfacing device's output will consist of a dry contact closure or produces a low voltage signal to give us a signal that represents the appliance's status. Similarly, you might be able to control an appliance by tying into a low voltage control signal for the appliance. A frequently automated appliance is the automatic garage door. This provides us with a good example of how a module like the SECU16 can be used to provide new capabilities like automatically close it at a certain time of the day, or any other parameter that we often consider in our HA system.

The typical garage door opener provides a manual pushbutton to open or close the door from inside the garage. To keep controls simple, this button is usually a low voltage control, using voltages like 12 or 24 volts. This is the perfect place to tie into a our HA system for actuating the door. The only problem is that you typically have only one button, which will close the door if its open or open it if its closed. This means that to create a routine that closes the door, your system will need to "know" it's current open or closed state to decide if an actuation is needed. In its simplest implementation, our system will need one input for the current door position and one output to actuate the door.

A SECU16 input can be configured for supervised digital input, analog voltage input, or analog current loop input (used in industrial sensors). Since we only want the open/closed status of the door, the supervised digital input is the simplest mode to use. To sense the door's position, a normally closed magnetic reed switch will be used, like the ones typically used for doors in alarm systems. The reed switch will be mounted so that when the door is closed, the magnet is in proximity to the switch and the contact closes. As specified in the SECU16's instructions, a 1k resistor is connected across the magnetic switch's contacts. To actuate the door's operation, a SECU16 output relay is wired in parallel with the door's wall mounted pushbutton. Fig. 106 shows a schematic diagram of the wiring for both the input and the output circuits to interface to the garage door system.

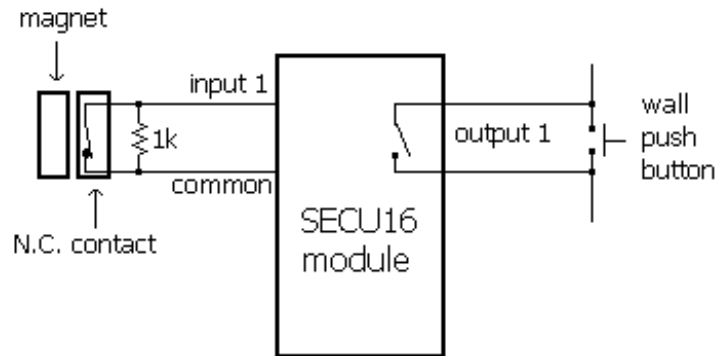


Fig. 106

As for implementing the routine in C-Max, here is a sample program to initiate an automatic closure of the garage door at 8:00 PM every day (Fig. 107):

```

0001 - IF Time of Day becomes = 20:00 // if time = 8:00 PM
0002 - AND Module #1 -SECU16 Input #1 Is OFF // and door not in closed position
0003 - THEN Module #1 -SECU16 Relay #9 Turns ON // "press" the door button
0004 - THEN Timer #3 = 1 // and start interval timer
0005 - IF Timer #3 becomes > 2 // two seconds later
0006 - THEN Module #1 -SECU16 Relay #9 Turns OFF // "release" the door button
0007 - THEN Timer #3 = 0 // and stop the interval timer

```

Fig. 107

Note the use of a timer to create the time interval during which the manual activation button is held down. Given that we have a logic processor controlling the garage door, it becomes easy to add additional features to the system. Since it is an unsupervised system that is closing the door at 8:00 PM, you might want to be sure that it is actually closed before retiring for the night. If a foreign object or other fault prevented the door from closing, we would like to be warned of that fact. Here is an addition to our program that verifies if the door is closed one minute later, at 8:01 PM (Fig. 108):

```

0008 - IF Time of Day becomes = 20:01 // if time = 8:01 PM
0009 - AND Module #1 -SECU16 Input #1 Is OFF // and door is open
0010 - THEN Send Module #2 -SPEAK-EZ Audio Message #6 // announce that door is open

```

Fig. 108

Line 10 uses a Speak Easy sound module to announce that the door is still open. As you can see, it becomes easy to customize our system since the expansion modules become an integral part of the system and are accessed directly with C-Max instructions.

The SECU16 module can also have its inputs configured for analog mode. Configured this way, the input will return a numerical value between 0 and 255, proportional to the voltage applied across its input. Zero volts will give an analog value of zero, while +5 Vdc will give a reading of 255. This is useful for acquiring analog values like tank levels, light levels, etc.; anything that can be converted to a 0 to +5 Vdc signal. As a second and final example, we will show how a tank level sensor can display (for example, on a Leopard screen) the level in the tank as a percentage of “full”. Fig. 109 shows how the sensor would be connected to the SECU16 input, and illustrates the sensor as a potentiometer giving a voltage proportional to liquid level.

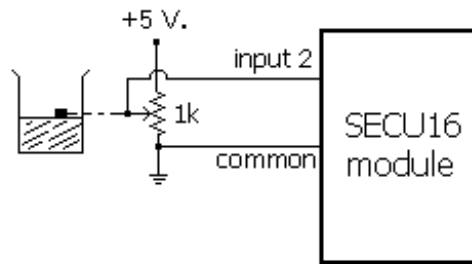


Fig. 109

The code needed to read the input and convert it to a percentage is shown in Fig. 110. Because the controller does not have decimal math, we cannot just divide the 0 to 255 analog scale by 2.55 to get a percentage (0 to 100) so instead, we multiply the reading to increase its divisibility and then divide to get a value from 0 to 100. In our example, we multiply the reading by 255 to get a value from 0 to 65025 (we cannot exceed 65535, the maximum a variable can be set to) and then divide by 650, which will produce a maximum of 100 due to integer division:

```

0001 - IF Module #1 -SECU16 Analog #2 is < 256 // read analog input 2 if < 256 (always)
0002 - THEN Load Data to: Variable #2 // and capture into variable #2
0003 - THEN Variable #2 * 255 // multiply by 255
0004 - THEN Variable #2 / 650 // then divide by 650
0005 - THEN Variable #3 = Variable #2 // and copy to screen display variable

```

Fig. 110

These two examples demonstrate how adding digital and/or analog input/output devices to your ADI system can be easily accomplished using the available modules and by adapting the device’s controls to be interfaced within the modules’ specifications. A user familiar with standard interface types such as open collector outputs, dry contacts, etc. will be able to select the right interfacing hardware or even build his own.

5.6 Ocelot/Leopard Parameters

<u>Param Number</u>	<u>Description of Parameter</u>	<u>Values / Usage</u>
1	Bobcat parameter display	* 0 = Off * 15 = Bobcat address at 5 and data will display in variable 20, Bobcat address at 6 and data will display in variable 21, etc.
2	Power Mode	* 1 = low power mode * 0 = normal mode
3	Max Module address	* Highest module # to scan for. This is automatically set after auto addressing modules.
4	Daylight Saving Time Status	* 1 = DST in force * 0 = DST not in force
5	Daylight Saving Time Enabled	* 1 = Check for DST * 0 = Do not check for DST
6	Enable touch response	* 0,3 = do not respond * 1 = respond with virtual button match * 2 = respond with grid location (0-59) 6 Rows of 10 0 = upper left, 59 = lower right When enabled, sends 0xF2,00,button #,00
7	Rescan Time	* Auto rescan in minutes (0 = OFF)
8	Net Timeout	* Time in 1/10ths of sec to wait for module to respond
9	Retry Counter	* Flag "I/O Error" after this many retries
10	Reserved	*
11	Reserved	*
12	Reserved	*
13	Not Used	*
14	Master or Slave Address	* 0 = Master * 1 or higher = Slave address (Note: All slaves must have an address unique from the other slaves or modules)
15	Auto X-10	* 0 = Off * 1 = On Send X-10 0xFE (RX),hc,kc when X-10 is received. 0xFB (TX),hc,kc when X-10 is transmitted. (hc = house code, kc = key code or unit code)
16	Auto I/O	* 0 = Off * 1 = On Send 0xFF if the remote I/O status has changed.

17	Auto IR	* 0 = Off * 1 = On Send IR number when a comparison match exists, 0xFD (RX) or 0xFC (TX), IR_ number
18	Send ASCII IR	* 0 = Off * 1 = Send ASCII string on IR recognize + T000xxx xxx = IR number recognized.
19	Controller Type	* 0 = Ocelot * 1 = Leopard
20	Max IR received	* Default = 80 The higher the number the longer the time required to check for an IR match
21	RCS X-10 Thermostat Display	* 0 = Off * 1 = On RCS X-10 Thermostat temperature is displayed in variables 64-79 for House codes A-P
22	Reset Variables on Power Up	* 0 = All variables reset * 1 or higher = Variable not to be reset on power up Example: 40 = 40-127 not reset on power up
23	Internal Use	* Always leave at Zero (subject to change)
24	Reserved	*
25	Serial Port Sleep Time	* Time in 1/10ths of sec that any input from the serial port will be queued but ignored after a serial transmission from the controller.
26	Touch Button Queue Mode	* 0 = return touch object numbers only 1 = return touch object number followed by physical grid location number