Ocelot Manual

Table of Contents

1 Overview of the Ocelot

The Ocelot is Applied Digital Inc's (ADI) second generation of the original CPUXA home automation (HA) controller. The Ocelot features a complete programming language allowing the user to execute tasks based on complex "If/Then" logic. Tasks can be activated 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. An external computer can be connected to the Ocelot 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 Ocelot 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 it's resources. The Leopard (and Leopard II) is a controller with the same capabilities as the Ocelot but with the addition of a programmable touch screen that can trigger events and display information under program control.

1.1 Hardware Description

The Ocelot 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 along the top edge 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 Ocelot (with the built-in IR receiver disabled).
- Serial port. The RS-232 serial port is used to load programs and other data into the Ocelot and also serves as an interface to any external computer program that supports the Ocelot as an interface to a HA system.
- RS-485 expansion bus. The bus is used by the proprietary Adnet protocol to allow the Ocelot to communicate with the various expansion modules and slave controllers available from ADI. Like all other ADI controllers, the Ocelot can be configured as either a master or slave controller.

1.2 Hardware Specifications

Size: 6.5"L x 3.75"W x 1.38"H

I/O: RS232 - DBSF w/6' Cable
X-10 -RJ11 w/6' Cable
ADNET: 2 Screw Terminals*
Power: 2 Screw Terminals*
*(up to 14 AWG)
Infra-Red In - Stereo 3.5mm Jack
Infra-Red Out - Mono 3.5mm Jack

Each Unit Includes: User's Guide and C-Max 2.0 Control Wizard Software
Comms Cable for OCELOT - PC communication
25' cable for TW523 communication (TW523 not included)
9-12V @ 1A power supply

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 Ocelot

1.4.1 Hardware Installation

The Ocelot's modular style case allows it to be installed practically anywhere. 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 Ocelot. This might mean planning ahead to avoid cable routing obstacles such as studs and horizontal braces in a wall. Cables should always run from the Ocelot to an easily accessible area. Also consider the accessibility of the built-in Infrared receiver if you will be using it to learn Infrared codes or to send Infrared commands to the Ocelot to control it.

Fig. 1 shows the physical location of the various connectors and jacks for the Ocelot. The screw terminals for the power and communications bus are actually on a plug in connector that can be removed (by pulling it out) to make connections easier to perform.

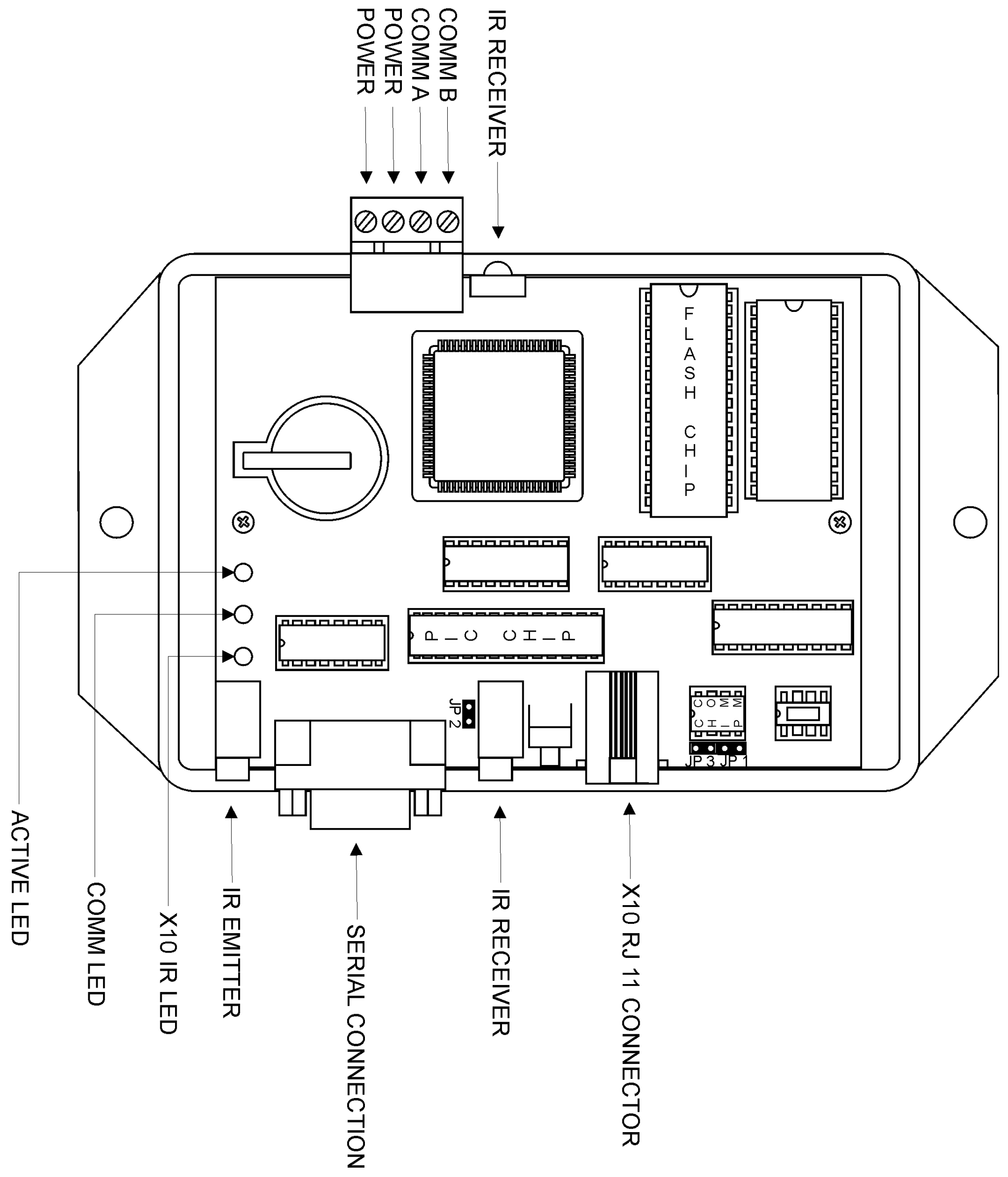

IR RECEIVER
COMM B
COMM A
POWER
POWER
FLASH CHIP
PIC CHIP
JP 2
COMM CHIP
JP 3
JP 1
ACTIVE LED
COMM LED
X10 IR LED
IR EMITTER
SERIAL CONNECTION
IR RECEIVER
X10 RJ 11 CONNECTOR

Fig. 1

1.4.1.1 Power

The Ocelot requires either a 12-volt AC or DC, 200mA power supply (included with unit). The 12-volt power source is connected to the 4-position terminal plug at the top of the case. Make sure you correctly identify the power and bus connections before applying power. Polarity of the power wires is not important for this unit.

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

X-10

The RJ11 (telephone type) jack along the bottom is for the X-10 communications. Connect the supplied RJ11 cable to this jack, and then plug the other end into a PSC05 or TW523 X-10 interface (not included). Note that the supplied cable is wired pin to pin, and does not "roll over" the wires from one end to the other like most telephone cables. Be careful in observing this if you decide to use an extension or another cable for this connection.

Infrared

The IR out jack is located at the bottom left of the Ocelot. It is for a standard 5v mini IR emitter. To the right of the serial connector is the IR in jack. This can be used with the optional IR receiver dongle as an alternate way to receive IR commands (instead of the built-in IR receiver).

Serial port

This is a standard DB-9 connector. Use the supplied serial cable to connect this port to the PC that will be running C-Max to program the Ocelot. Note for alternate or extension wiring: only 3 wires are needed in the serial cable. These correspond to pins 2,3 and 5.

RS-485 bus (Adnet)

On the same connector as the power input. These terminals are used to connect to one or more of our Adicon 2500 modules or slaves. Connect Comm A to Comm A and Comm B to Comm B in a daisy chain manner.

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, event viewing tool, and many other functions. In order to program and use the Ocelot 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 Ocelot 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 program 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 other 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 then 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. 2)

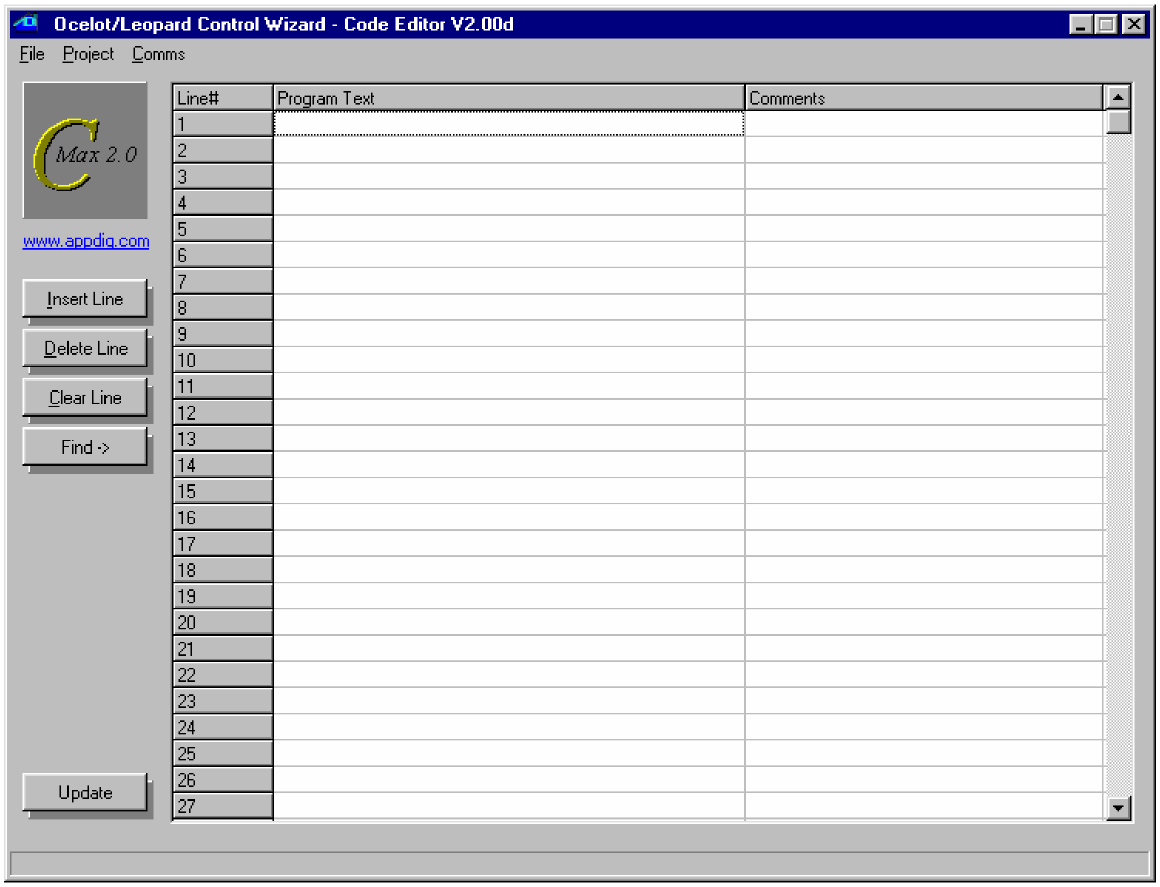
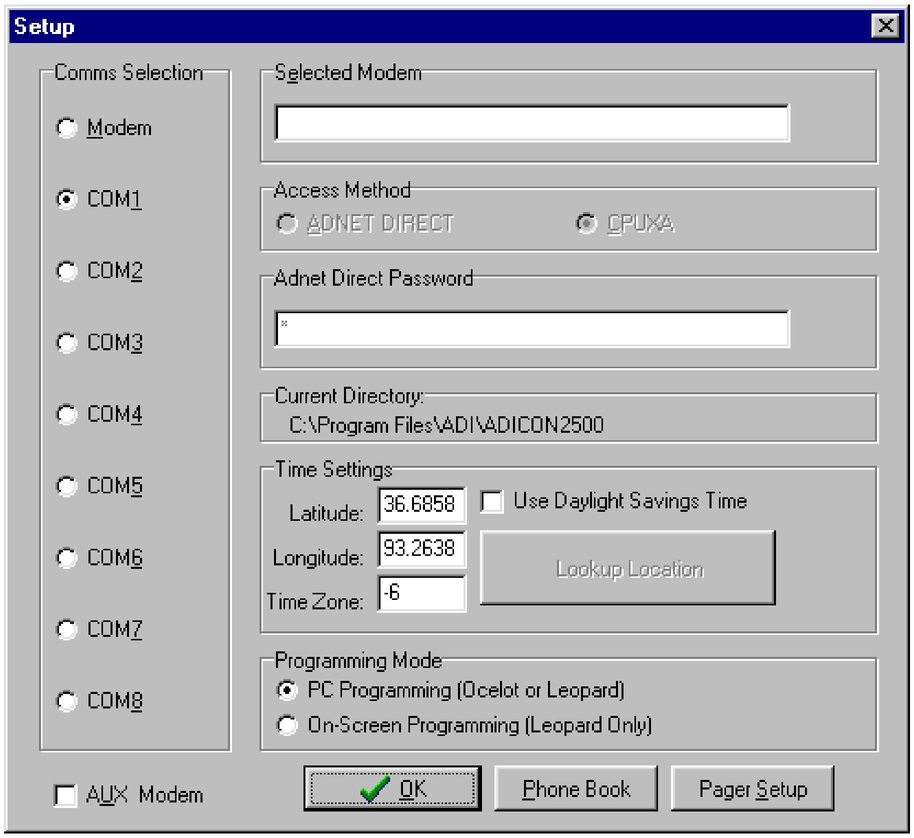

Ocelot/Leopard Control Wizard - Code Editor V2.00d
File
Project
Comms
Max 2.0
www.appdig.com
Insert Line
Delete Line
Clear Line
Find ->
Update
Line#
Program Text
Comments
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27

Fig 2

Click on Comms → Comms Setup and a screen similar to the following will appear (fig 3):


Setup

Comms Selection: ( ) Modem [✓ COM1] ( ) COM2 ( ) COM3 ( ) COM4 ( ) COM5 ( ) COM6 ( ) COM7 ( ) COM8

Selected Modem:

Access Method: ( ) ADNET DIRECT [✓ CPUXA]

Adnet Direct Password: *

Current Directory: C:\Program Files\ADI\ADICON2500

Time Settings

Latitude: 36.6858 ☐ Use Daylight Savings Time

Longitude: 93.2638

Time Zone: -6

Lookup Location

Programming Mode: [✓ PC Programming (Ocelot or Leopard)] ( ) On-Screen Programming (Leopard Only)

☐ AUX Modem

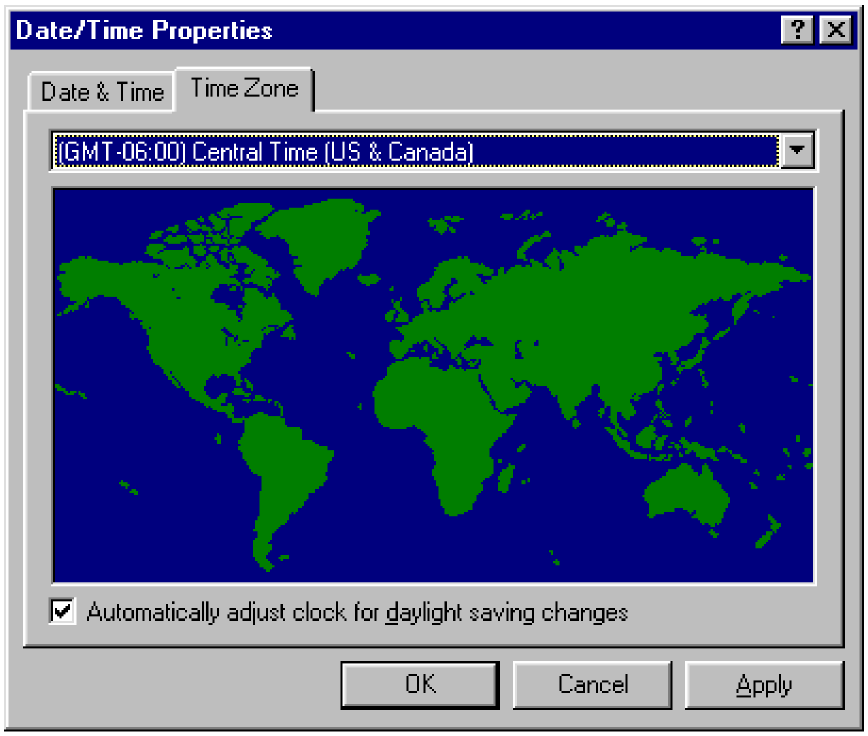
OK Phone Book Pager Setup

Fig. 3

In the "Comms Selection" area, choose your computer's serial port to which you connected the Ocelot'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. 4):



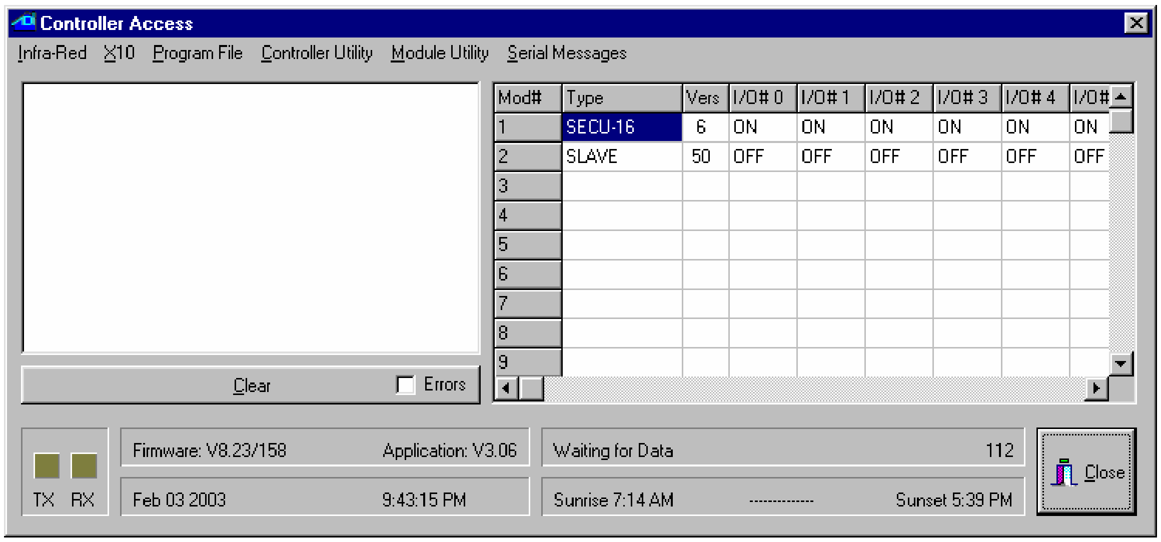
(GMT-06:00) Central Time (US & Canada)

[✓] Automatically adjust clock for daylight saving changes

Fig. 4

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. 3

These settings allow the Ocelot 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. 5:



Mod#	Type	Vers	I/O# 0	I/O# 1	I/O# 2	I/O# 3	I/O# 4	I/O#
1	SECU-16	6	ON	ON	ON	ON	ON	ON
2	SLAVE	50	OFF	OFF	OFF	OFF	OFF	OFF
3								
4								
5								
6								
7								
8								
9								

Firmware: V8.23/158 Application: V3.06

Waiting for Data 112

Feb 03 2003 9:43:15 PM

Sunrise 7:14 AM ------------ Sunset 5:39 PM

Fig. 5

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 Ocelot. You should also look at the “Firmware” and “Application” fields to see that a value has been read from the Ocelot. The actual values may differ from the ones shown in fig. 5 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 then the version that the Ocelot 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 Ocelot.

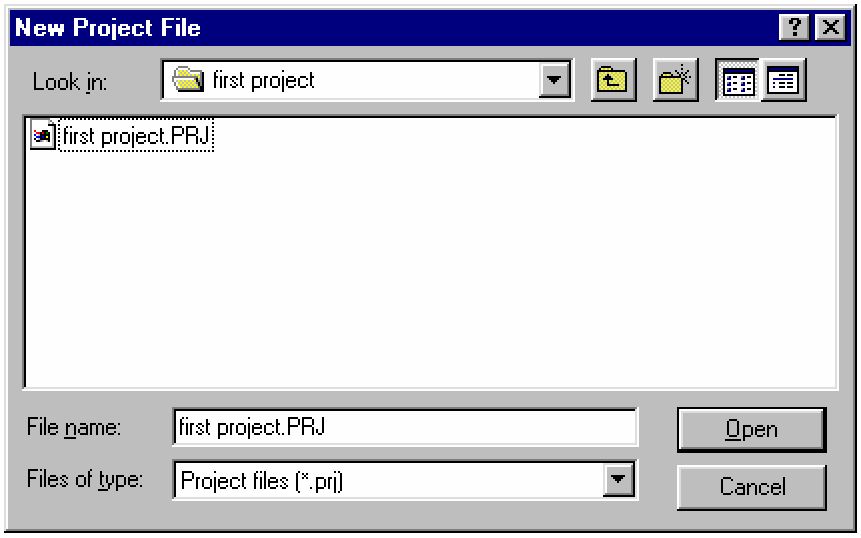
If C-Max programming is totally new to you and you would like to have a feel for how to proceed with creating a working application, or you would just like to have the immediate satisfaction of seeing it “do something” then the next chapter will be of interest to you. It literally holds your hand through 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 Ocelot 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 an X10 "macro", used to turn X-10 controlled lights On and Off. We will begin by starting up the C-Max program and looking at the main screen (see fig. 2). 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. 6



File name: first project.PRJ

Files of type: Project files (*.prj)

Fig. 6

It is a good idea to keep each project in it's 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. 6, 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 Ocelot 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. 7:

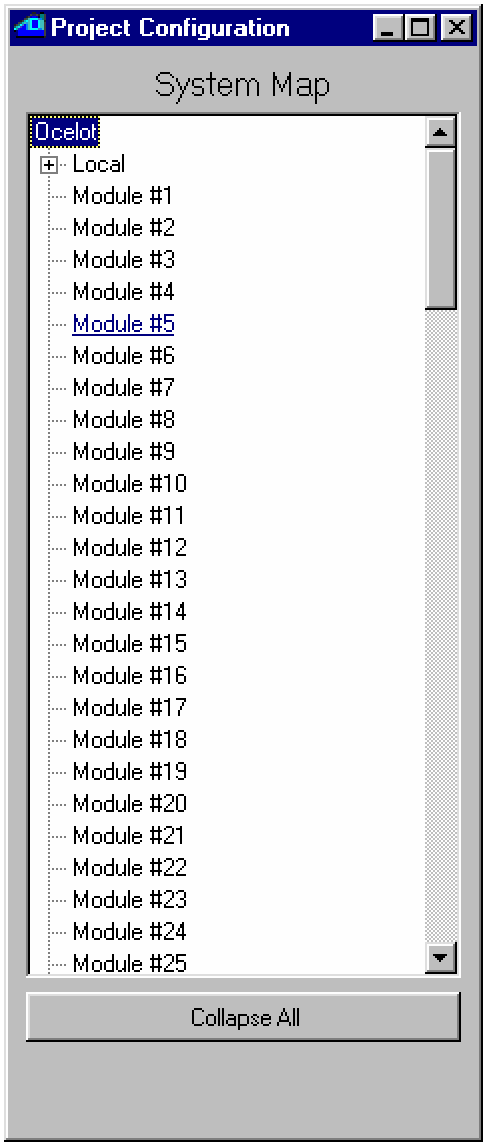


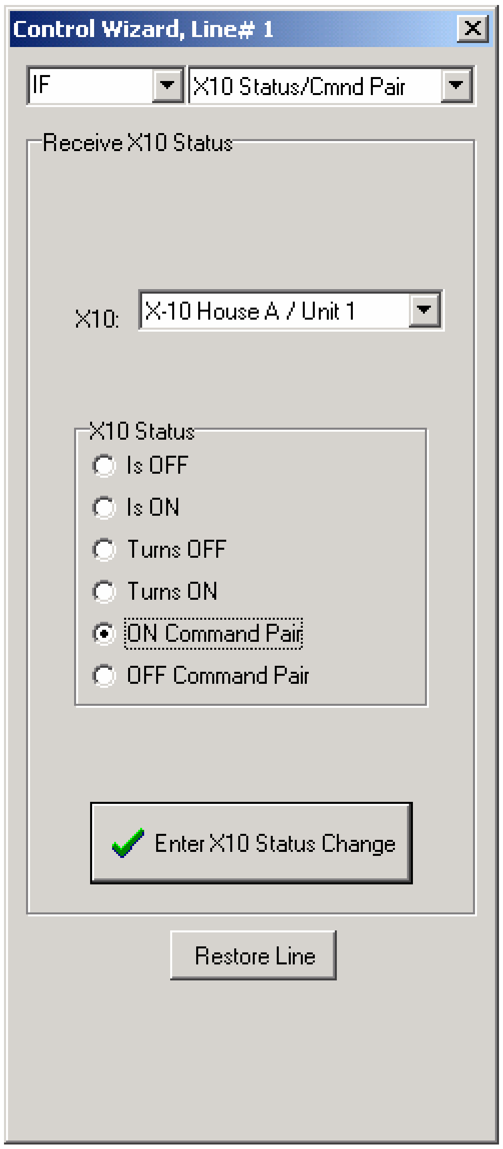
Fig. 7

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. Right clicking on the word "Ocelot" will show that you can choose between Ocelot and Leopard. The Leopard is another ADI controller similar to the Ocelot but with a touch screen. Since we have an Ocelot, there is no need to modify this default value, but if you do right click on it, then make sure you choose Ocelot. You may now close the System Map window by clicking on the "x" in the upper right hand corner.

1.6.3 Writing the Program Code

Now we will write the actual program code for our project. To keep this project simple, we will create a program that looks for an X10 "A/1, A/On" command pair, and respond by sending X10 commands to turn devices B/1, B/2, and B/3 On. Of course, feel free to substitute actual X10 house and unit codes that you have if you want to actually test the completed application.

Under the **Program Text** heading, double click on line #1. You will get the control wizard screen as shown in fig. 8. Select the **IF** instruction type and then **X10 Status/Cmnd Pair** instruction (use the vertical scroll bar to find this instruction.). The window will update its available controls to show the ones that apply to this command. Select the desired house and unit codes from the list box. Next, click on the radio button next to **ON Command Pair**. Once all these selections have been made click on the large **Enter Program Line** button at the bottom of the control wizard and you will see your newly created program line in the main window.


Control Wizard, Line# 1

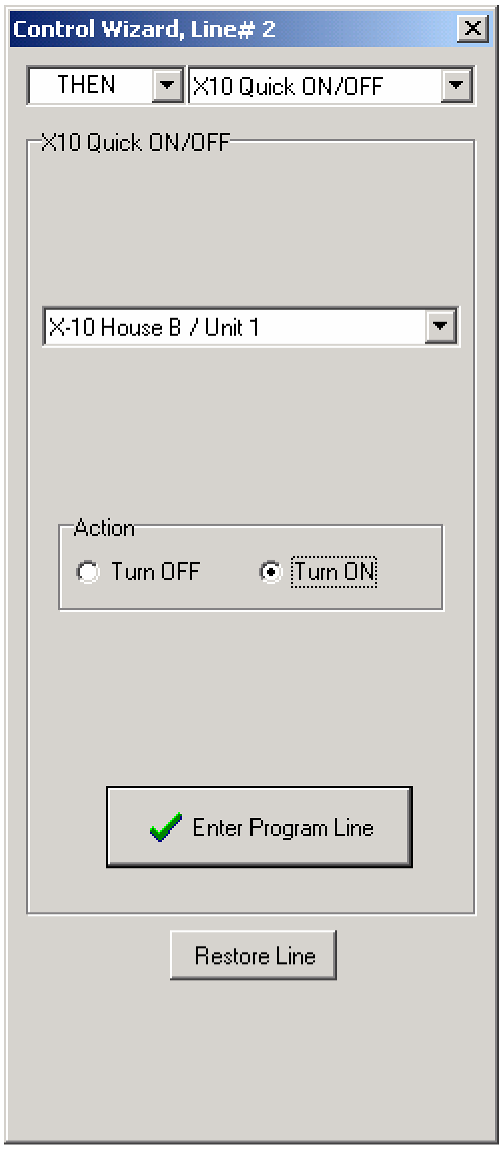
IF | X10 Status/Cmnd Pair

Receive X10 Status

X10: X-10 House A / Unit 1

X10 Status: Is OFF, Is ON, Turns OFF, Turns ON, [✓ ON Command Pair], OFF Command Pair

Fig. 8


Control Wizard, Line# 2

THEN | X10 Quick ON/OFF

X10 Quick ON/OFF

X-10 House B / Unit 1

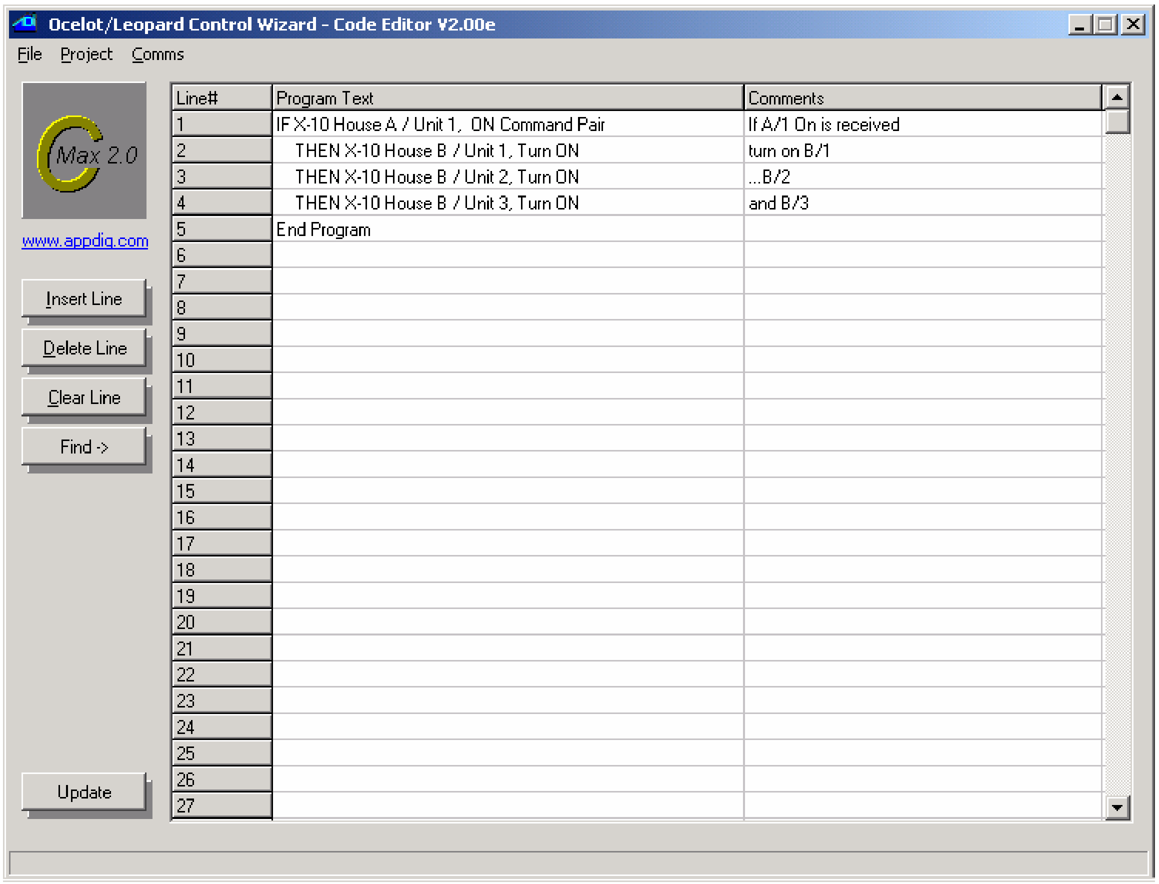
Action: Turn OFF [✓ Turn ON]

Fig. 9

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. 9. 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 house/unit codes **B/1**, or 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 on codes B/2 and B/3 (or whatever you like). Just double click on the line you want to create or edit and make the appropriate choices.

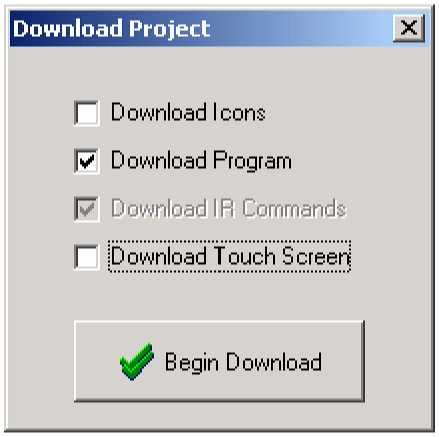
We're almost done! We have the test statement to look for an X10 command and action statements to turn some X10 devices ON. All we need now is an **END** statement. Double click on the next blank line line and in the control wizard, select the **END** command type. There are no other options to this 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. 10 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. 10. 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**.



Line#	Program Text	Comments
1	IF X-10 House A / Unit 1, ON Command Pair	If A/1 On is received
2	THEN X-10 House B / Unit 1, Turn ON	turn on B/1
3	THEN X-10 House B / Unit 2, Turn ON	...B/2
4	THEN X-10 House B / Unit 3, Turn ON	and B/3
5	End Program	
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		
17		
18		
19		
20		
21		
22		
23		
24		
25		
26		
27		

Fig. 10

The only thing left to do at this point is to get the program code into the Ocelot 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. 11. Using the check boxes, select just the **Download Program** option.

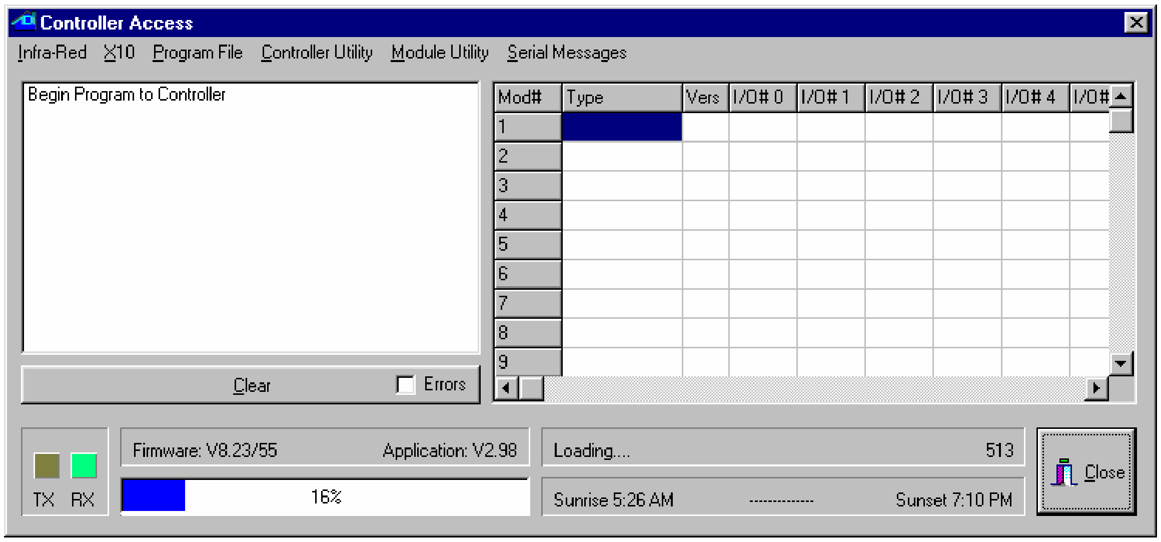


Download Project

- [ ] Download Icons
- [x] Download Program
- [x] Download IR Commands
- [ ] Download Touch Screen

Begin Download

Fig. 11



Controller Access

Infra-Red X10 Program File Controller Utility Module Utility Serial Messages

Begin Program to Controller

Clear ☐ Errors

Mod#	Type	Vers	I/O# 0	I/O# 1	I/O# 2	I/O# 3	I/O# 4	I/O#
1								
2								
3								
4								
5								
6								
7								
8								
9								

TX RX

Firmware: V8.23/55 Application: V2.98

16%

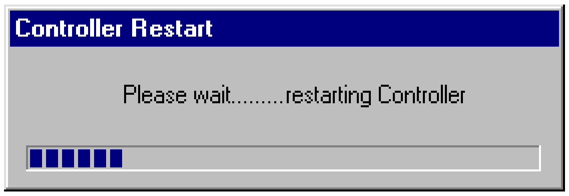
Loading.... 513

Sunrise 5:26 AM ------------ Sunset 7:10 PM

Close

Fig. 12

Click on **Begin Download**. The controller access window will appear and show the progress of the operation (see fig. 12). 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. 13) indicating that the controller is being restarted.



Controller Restart

Please wait.........restarting Controller

Fig. 13

Once program loading has finished, wait about 10 seconds and then try sending an A/1 On command pair using an X10 transmitter (mini controller, palm pad, etc.) and you should see the macros execute, turning on the programmed devices. Congratulations, you have completed your first Ocelot 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 Ocelot

2.1 The C-Max programming model

The C-Max utility that comes with the Ocelot 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 Ocelot 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 Ocelot 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 it's adoption for the Ocelot.

2.2 Ladder Logic

The programming model used for the Ocelot 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 Ocelot 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 a 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 Infrared code #1 is received. 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 Receive IR #1, from Ocelot
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 Receive IR #1, from Ocelot
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 (often called the "OR trap"…). If your intent was to allow the A/1 ON command or the IR code to work only if past sunset, then you'll have the unexpected behavior that the IR code 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/1 On Command Pair
0004  OR Receive IR #1, from Ocelot
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 then 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 a button connected to a SECU16 input 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 Leopard 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 then 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 +....+ 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 as such in an ASCII string by using a formatting option that instructs the formatting routine 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 a 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 Ocelot** 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 on a Leopard 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. Although some examples use Leopard touch button presses as triggering events, any single triggering event available with the Ocelot or Leopard can be substituted. 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 Ocelot 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 then 40 such commands in your program code, the 41st and any subsequent such lines will be ignored. Also, the scan for **IF Module/Param** commands stops as soon as the first **END** instruction is encountered. This means that if you are using more then one **END** statement, any **IF Module/Param** commands beyond it will not be added to the table, even if the table is not yet full. 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 Ocelot. 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.	Lowest 8 bits = Command/Unit Code:
00 = House code A	00 = unit code 1
01 = House code B	01 = unit code 2
…	…
0F = House code P	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 Ocelot will receive any X10 commands it transmits itself, and will process them like commands from any other source.

The Ocelot 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 a Leopard 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 then 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 Ocelot, 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 then a long one), the time between a timer being set to a value other then 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 then 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 then 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 Ocelot 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 Ocelot'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 Ocelot'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 Ocelots 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 Ocelot’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 Ocelot'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 Ocelot'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 Ocelot's RS-2332 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 Ocelot'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  Ocelot              // 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 Ocelot. 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  Ocelot      // 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 Ocelot'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 then the actual amount of codes that need to be recognized).

The Ocelot 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 Ocelot. 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 Ocelot maintains an in-memory status table of all 256 X10 addresses, and memorizes that the device is either o*n* 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 o*n* 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 Ocelot itself, but can come from any X10 device (the Ocelot 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 Ocelot will receive any X10 commands it transmits itself, and will process them like commands from any other source.

The Ocelot 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 then 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 a master or 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 a Leopard. Whenever a touch button object is pressed on the screen, it's 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 Ocelot. 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 Ocelot will receive any X10 commands it transmits itself, and will process them like commands from any other source.

The Ocelot 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 then 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 then a long one), the time between a timer being set to a value other then 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 then 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 then 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 Ocelot'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 then the **END** statement) that causes the program flow to continue with a program line other then 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 it's 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 in the Leopard 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 Ocelot. 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 Ocelot 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 Ocelot 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.

THEN Transmit IR *Module#, Zone#, IR#*

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 Ocelot 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 then 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. Also, do not use a mid program **END** statement if there are any **IF Module/Param** statements beyond it because the first **END** statement causes the interpreter to cease searching the program for additional **IF Module/Param** statements when it is building up its analog input queue (see reference for **IF Module/Param** statement).

4 C-Max User's Guide

This section of the manual describes the various utilities available in C-Max for programming and using your Ocelot. 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. 14), create a project by clicking on **Project → New Project**.

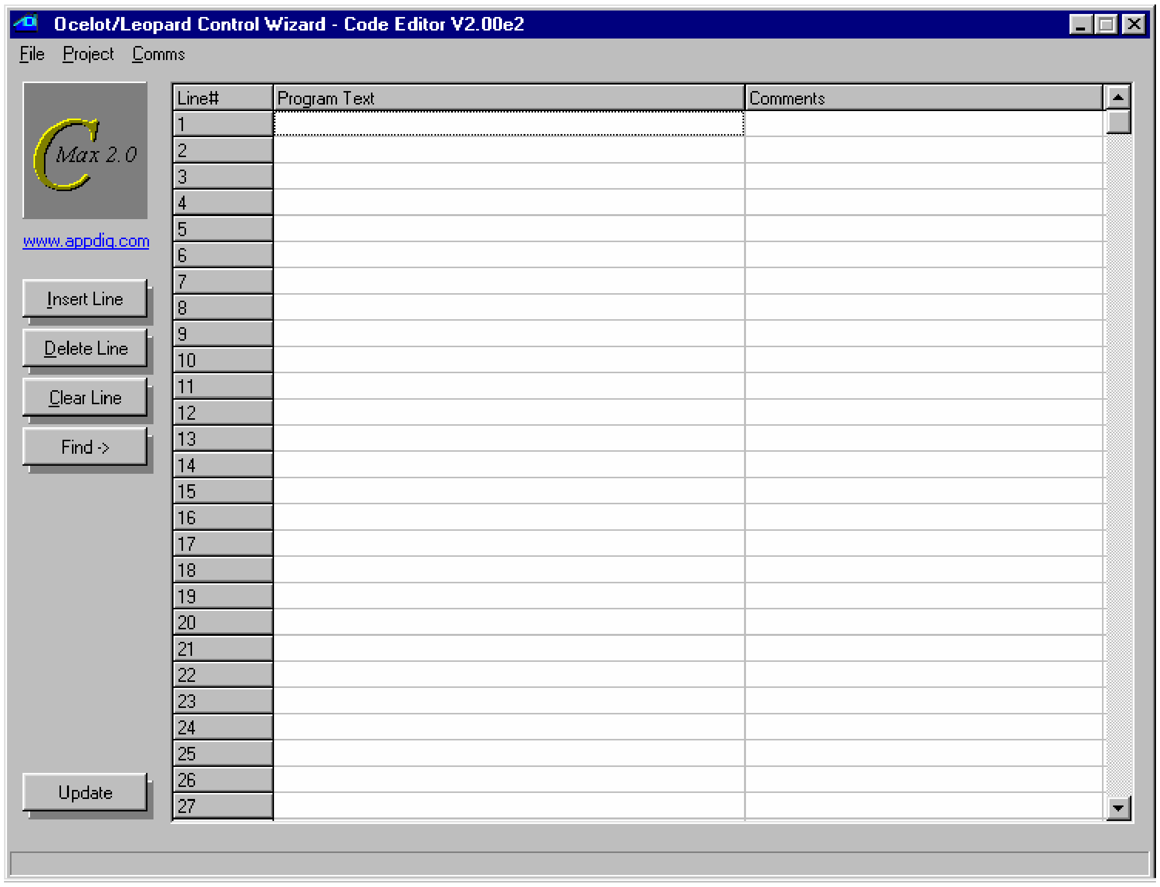


Fig. 14

You will see a file browsing window (fig 15).

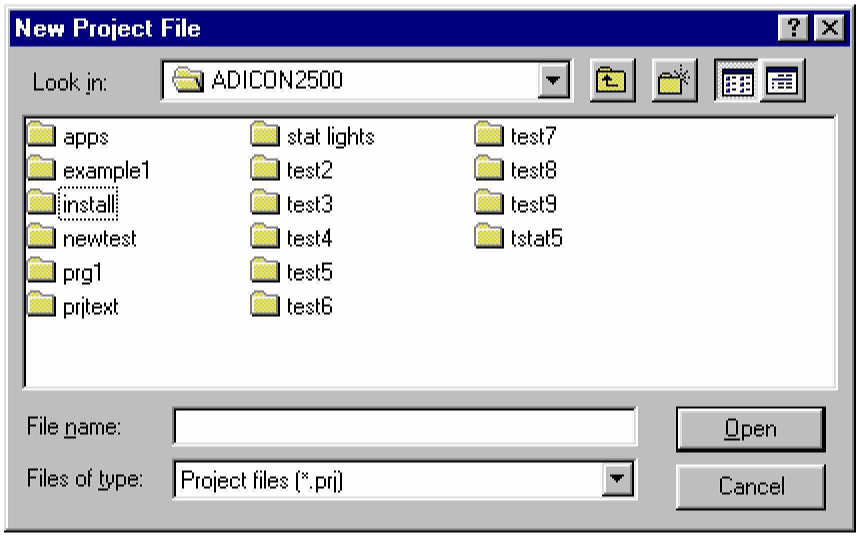


Fig. 15

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 seperate 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 16).

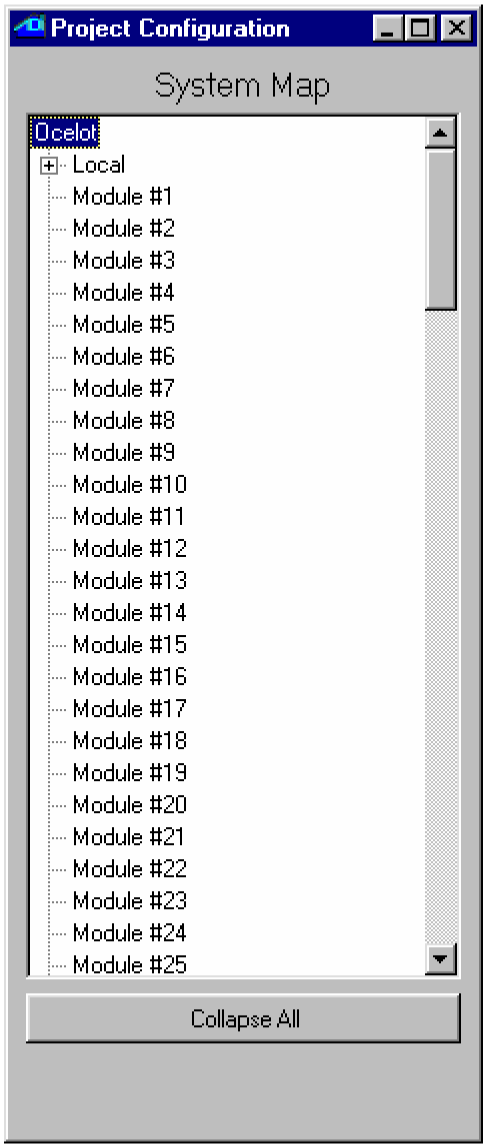


Fig. 16

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. 17). 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**.

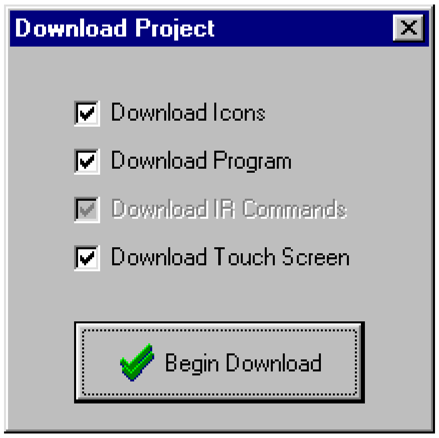


Fig. 17

4.3 The Program Editor

The program editor is the main screen that you see when you start up C-Max (fig. 14). 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 18), 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.

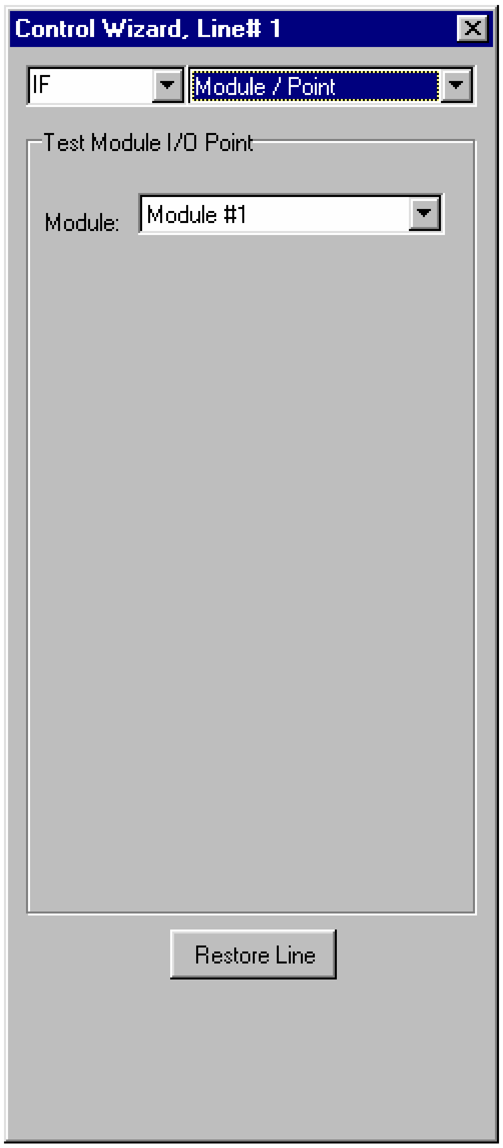


Fig. 18

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 by 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 then 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.

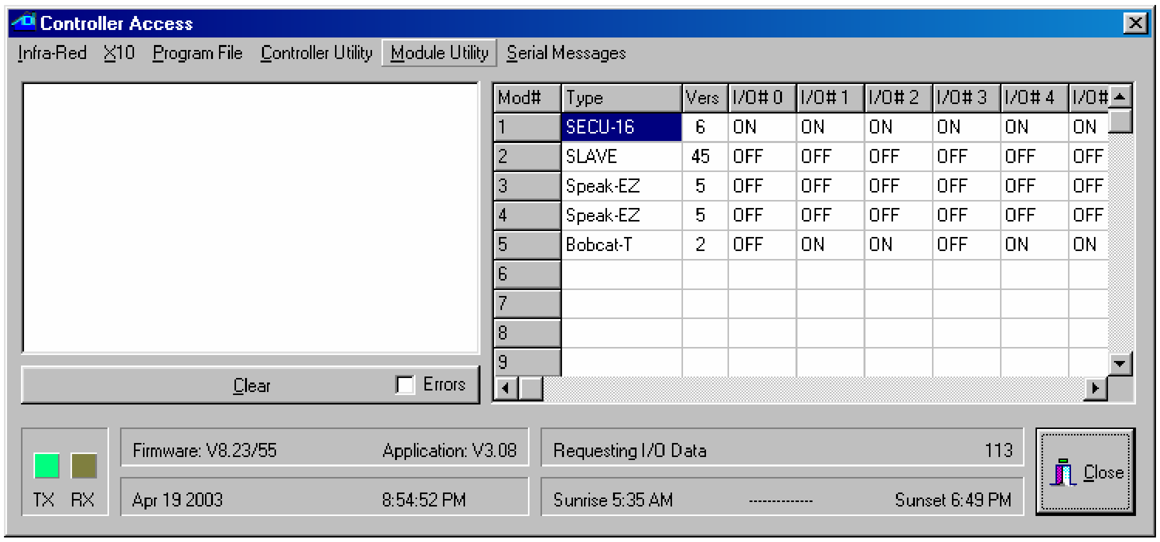
Save Program As – Saves the current program code in the editor to a file of your choice, using a file browser to allow you to choose the location and name of the program file. A warning window will appear if you are about to overwrite an existing program file. Like the **Open Program** command above, 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 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. 19):



Mod#	Type	Vers	I/O# 0	I/O# 1	I/O# 2	I/O# 3	I/O# 4	I/O#
1	SECU-16	6	ON	ON	ON	ON	ON	ON
2	SLAVE	45	OFF	OFF	OFF	OFF	OFF	OFF
3	Speak-EZ	5	OFF	OFF	OFF	OFF	OFF	OFF
4	Speak-EZ	5	OFF	OFF	OFF	OFF	OFF	OFF
5	Bobcat-T	2	OFF	ON	ON	OFF	ON	ON
6								
7								
8								
9								

Firmware: V8.23/55 Application: V3.08
Requesting I/O Data 113
Apr 19 2003 8:54:52 PM
Sunrise 5:35 AM ------------ Sunset 6:49 PM

Fig. 19

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 it's 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.

4.5.1 Infra-Red

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

4.5.1.1 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. 20):

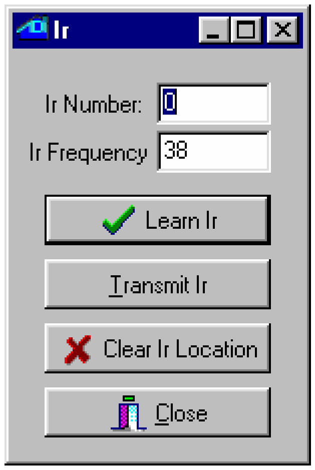


Fig. 20

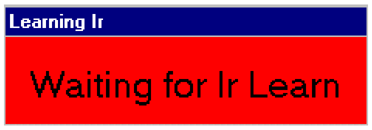


Fig. 21

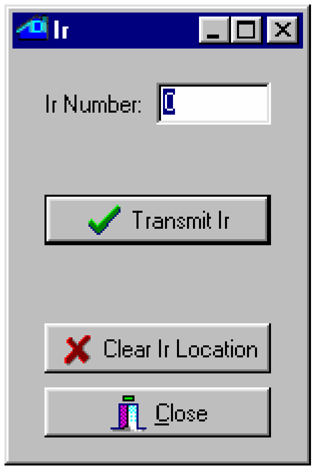


Fig. 22

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 21) prompting you to send the desired IR code to be learned. Point your IR device (TV remote, etc.) to the IR receiver of your Ocelot (the small open slit next to the power and bus connector at the top of the casing) 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. 20.

Choosing the right IR code numbers is important: You can use IR codes in two ways with your Ocelot: They may be learned to be reproduced later by the Ocelot, for the purpose of controlling equipment under C-Max program control. IR codes can also be recognized by the Ocelot itself as input events. 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 Ocelot 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 then 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 Ocelot.

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.

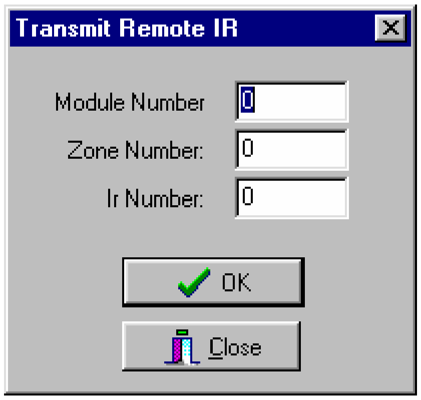
You can erase a learned code using the **Clear Ir Location** button.

4.5.1.2 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. 22) 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.

4.5.1.3 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 23). 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.


Transmit Remote IR

Module Number: 0
Zone Number: 0
Ir Number: 0

Fig. 23

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.

4.5.1.4 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.

4.5.1.5 Upload Infra-Red File from Controller

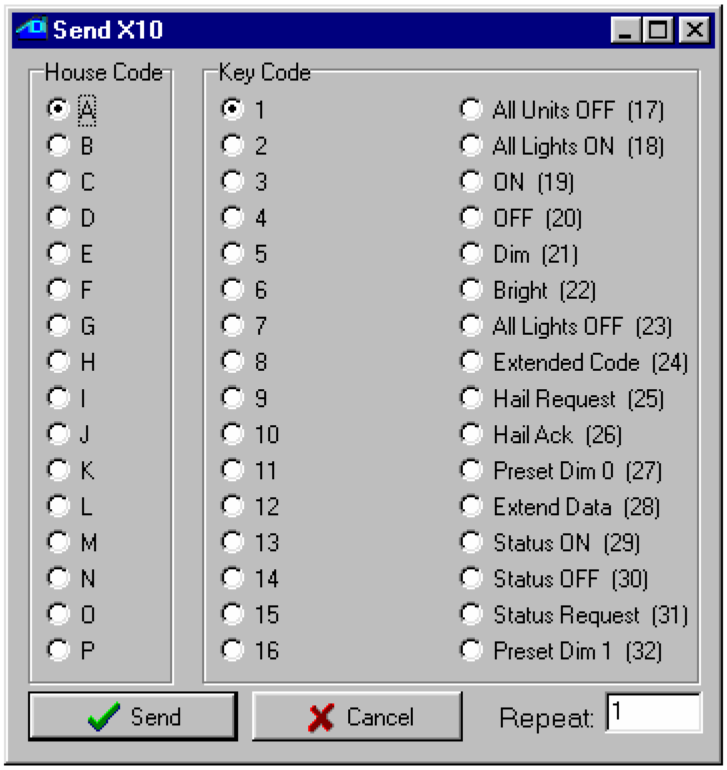
Upload Infra-Red File from Controller does the opposite operation, allowing you to store your learned IR codes to a disk file. This is useful for backing up all the IR codes that you learned manually, and can be used to load into another controller or even sent to another user as a disk file. Once again, 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).

4.5.2 X10

The X10 pull down menu has the following three choices:

4.5.2.1 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 24):


Send X10

House Code

[✓ A] B C D E F G H I J K L M N O P

Key Code

[✓ 1] 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

All Units OFF (17), All Lights ON (18), ON (19), OFF (20), Dim (21), Bright (22), All Lights OFF (23), Extended Code (24), Hail Request (25), Hail Ack (26), Preset Dim 0 (27), Extend Data (28), Status ON (29), Status OFF (30), Status Request (31), Preset Dim 1 (32)

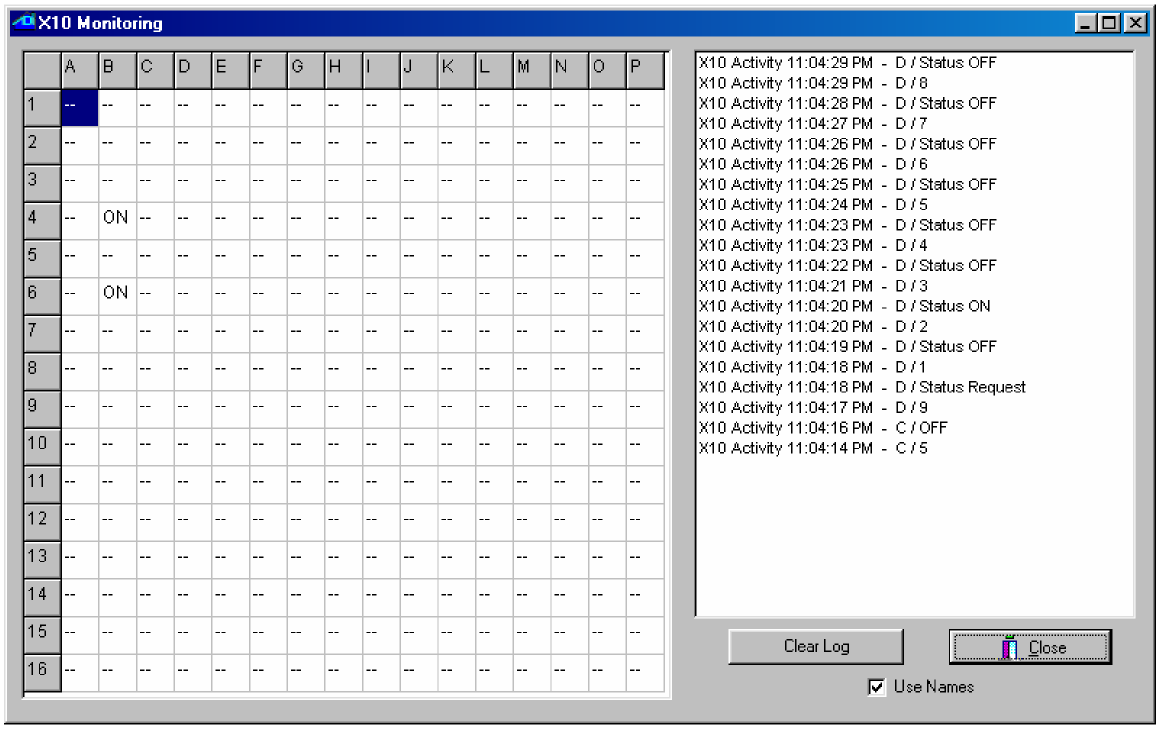
Repeat: 1

Fig. 24

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 then 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.

4.5.2.2 Monitor X10

This menu selection will open an X10 status and activity window (fig 25):



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
2	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
3	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
4	--	ON	--	--	--	--	--	--	--	--	--	--	--	--	--	--
5	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
6	--	ON	--	--	--	--	--	--	--	--	--	--	--	--	--	--
7	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
8	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
9	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
10	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
11	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
12	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
13	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
14	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
15	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
16	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

```
X10 Activity 11:04:29 PM  -  D / Status OFF
X10 Activity 11:04:29 PM  -  D / 8
X10 Activity 11:04:28 PM  -  D / Status OFF
X10 Activity 11:04:27 PM  -  D / 7
X10 Activity 11:04:26 PM  -  D / Status OFF
X10 Activity 11:04:26 PM  -  D / 6
X10 Activity 11:04:25 PM  -  D / Status OFF
X10 Activity 11:04:24 PM  -  D / 5
X10 Activity 11:04:23 PM  -  D / Status OFF
X10 Activity 11:04:23 PM  -  D / 4
X10 Activity 11:04:22 PM  -  D / Status OFF
X10 Activity 11:04:21 PM  -  D / 3
X10 Activity 11:04:20 PM  -  D / Status ON
X10 Activity 11:04:20 PM  -  D / 2
X10 Activity 11:04:19 PM  -  D / Status OFF
X10 Activity 11:04:18 PM  -  D / 1
X10 Activity 11:04:18 PM  -  D / Status Request
X10 Activity 11:04:17 PM  -  D / 9
X10 Activity 11:04:16 PM  -  C / OFF
X10 Activity 11:04:14 PM  -  C / 5
```

☑ Use Names

Fig 25

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 Ocelot "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.

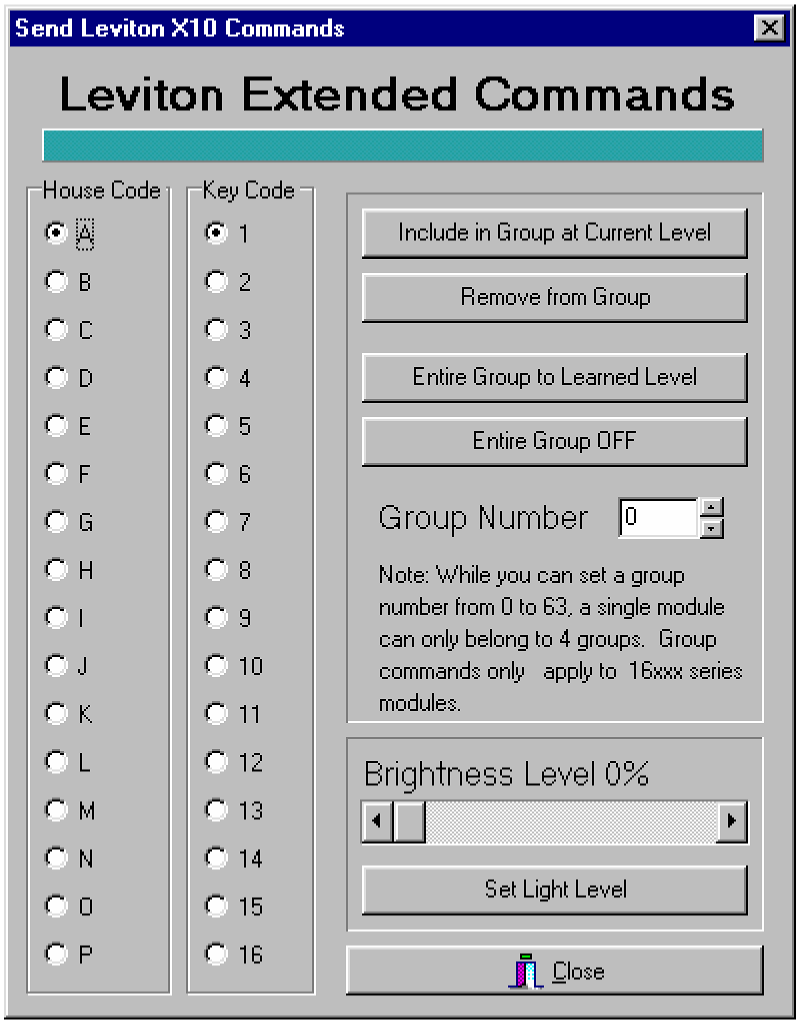
The right side of the window shows X10 events as they happen. Any new messages will be added to the top of the screen and push previous messages down. A scroll bar will appear on the right side of the window if the screen gets full. X10 commands will be shown in their raw form. You can choose to see 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.

4.5.2.3 <u>Send Leviton X10.</u>

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 26):

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.


Send Leviton X10 Commands
Leviton Extended Commands
House Code
Key Code
Include in Group at Current Level
Remove from Group
Entire Group to Learned Level
Entire Group OFF
Close
Set Light Level

House Code: [✓ A] B C D E F G H I J K L M N O P

Key Code: [✓ 1] 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Group Number 0

Note: While you can set a group number from 0 to 63, a single module can only belong to 4 groups. Group commands only apply to 16xxx series modules.

Brightness Level 0%

Fig. 26

4.5.2.3.1 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).

4.5.2.3.2 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.

4.5.2.3.3 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.

4.5.2.3.4 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.

4.5.3 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.

4.5.4 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.

4.5.4.1 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 Ocelot. Make sure the PC's date and time are set to the values you want in the controller before doing this!

4.5.4.2 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.

4.5.4.3 Reload Controller Executive.

Use this command to manually initiate a new download of the controller executive. The executive is the computer code that your Ocelot 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.

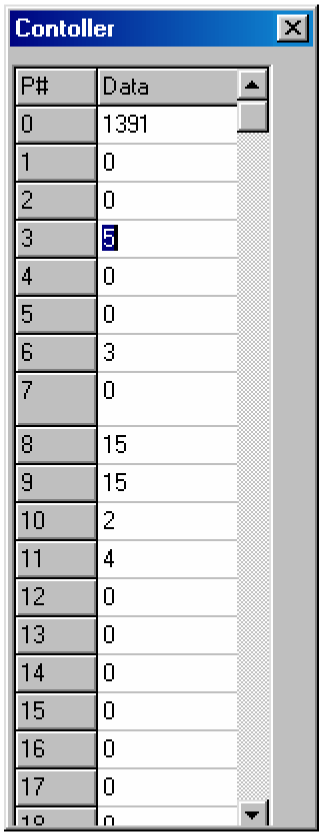
- Sometimes a new executive version is released to correct a very minor bug and the version number is left unchanged. This means that the automatic detection of the available new version will not happen. Use this command to manually do the executive reload.

4.5.4.4 Controller Memory Dump.

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

4.5.4.5 Retrieve Controller Parameters.

This command will read the configuration parameters from the controller and present them in an editable list (fig. 27). 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

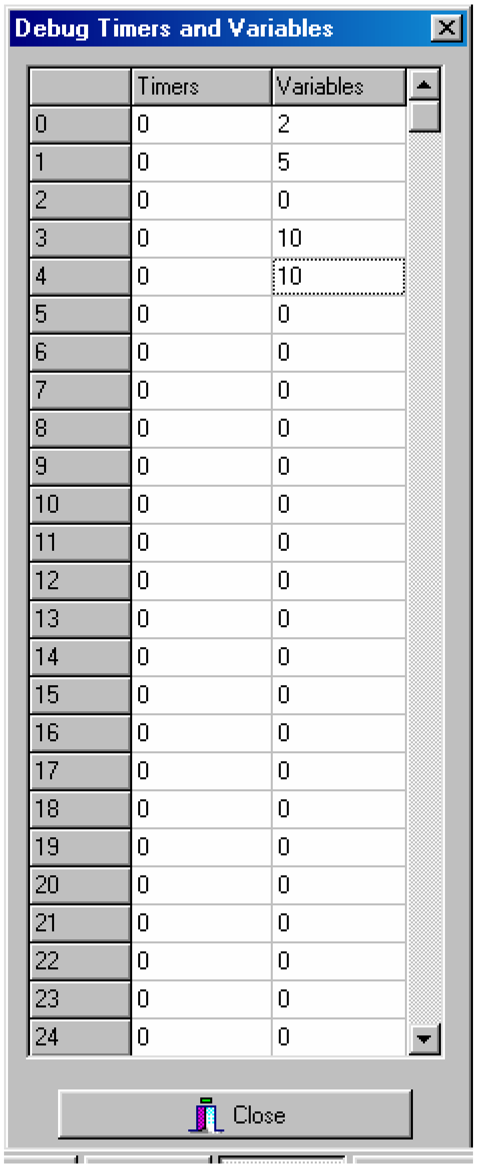
Fig. 27

4.5.4.6 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.

4.5.4.7 Debug Timers and Variables.

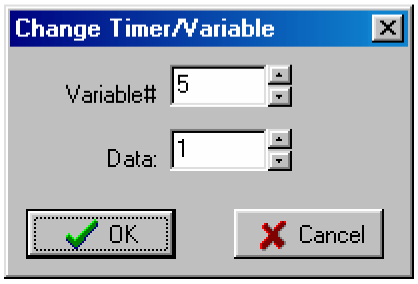
Selecting this menu choice will open a continuously updating window (fig. 28) showing the current value of all the timers and variables in the controller. This is one of the most often used utilities because it is handy to be able to see the immediate progression or changes of timers and variables in a running program. You can literally visualize changes as they happen, with no more then about one second of delay.



Debug Timers and Variables

	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. 28



Change Timer/Variable

Variable# 5

Data: 1

Fig. 29

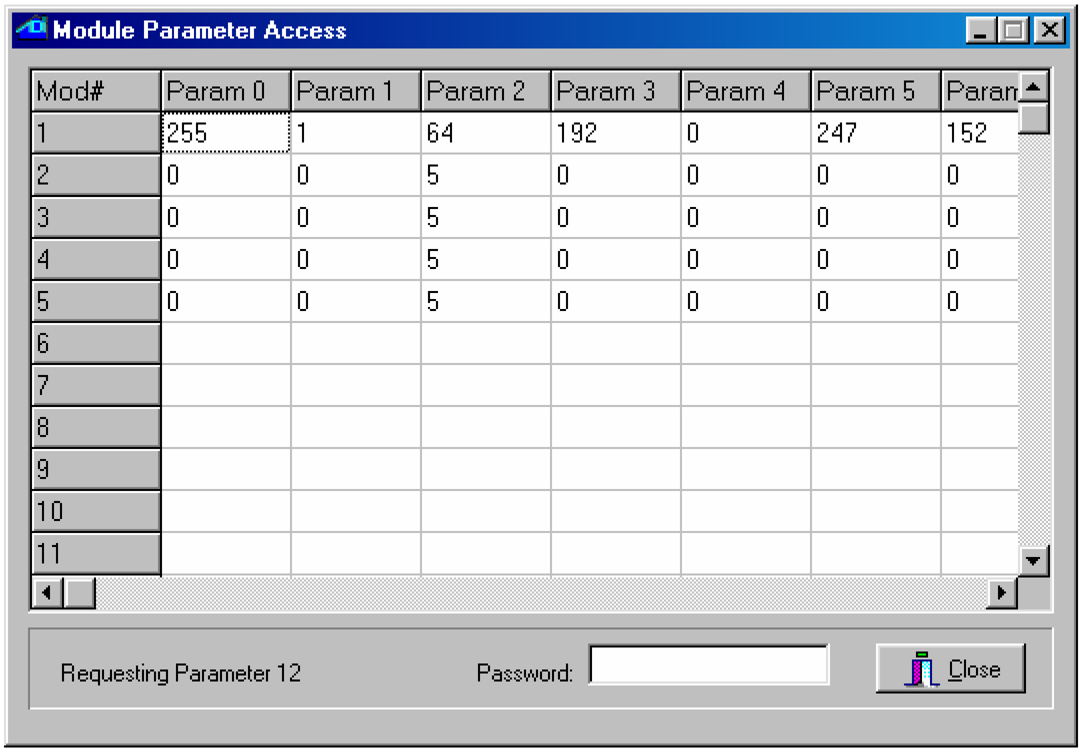
You can also modify the value of a timer or variable on the fly. To do that, click on the timer or variable that you want to change and a small editing window will appear (fig. 29). Enter the new value you want, then click OK.

4.5.5 Module Utility

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

4.5.5.1 Retrieve Module Parameters.

Clicking on this selection will open a new window (fig. 30) 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.



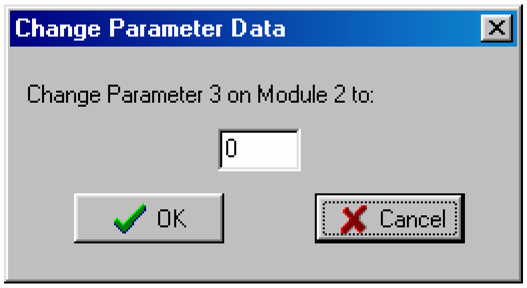
Mod#	Param 0	Param 1	Param 2	Param 3	Param 4	Param 5	Paran
1	255	1	64	192	0	247	152
2	0	0	5	0	0	0	0
3	0	0	5	0	0	0	0
4	0	0	5	0	0	0	0
5	0	0	5	0	0	0	0
6							
7							
8							
9							
10							
11							

Requesting Parameter 12

Password:

Fig. 30

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 31) allowing you the 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.



Change Parameter Data

Change Parameter 3 on Module 2 to:

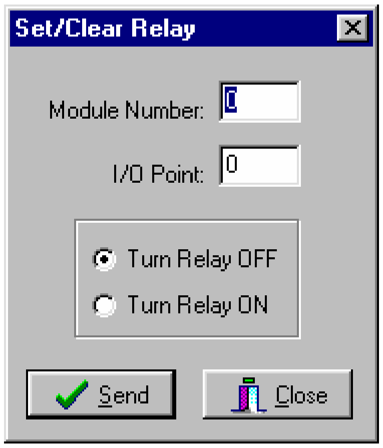
0

Fig. 31

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.

4.5.5.2 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. 32):



Set/Clear Relay

Module Number: 0

I/O Point: 0

[✓ Turn Relay OFF]
Turn Relay ON

Fig. 32

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.

4.5.5.3 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 33):

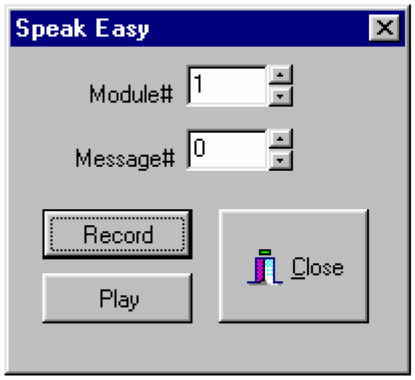


Fig. 33

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:

4.5.5.3.1 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.

4.5.5.3.2 Playing a Message:

· Access the Speak Easy Module Utility as described in the Recording a message section

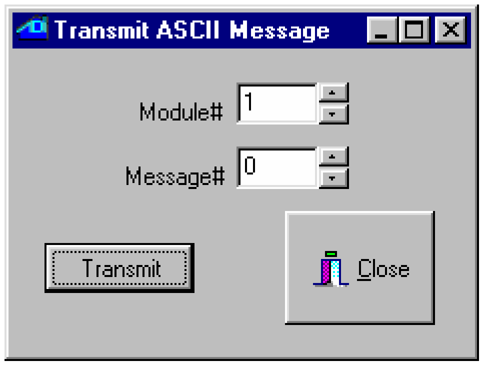
· Select the module number of the Speak Easy and the message number to play then click the PLAY button.

· 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.

4.5.5.4 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 34) in which you can specify the module # of the serial Bobcat and the message number that you want to send.



Module# 1

Message# 0

Fig 34

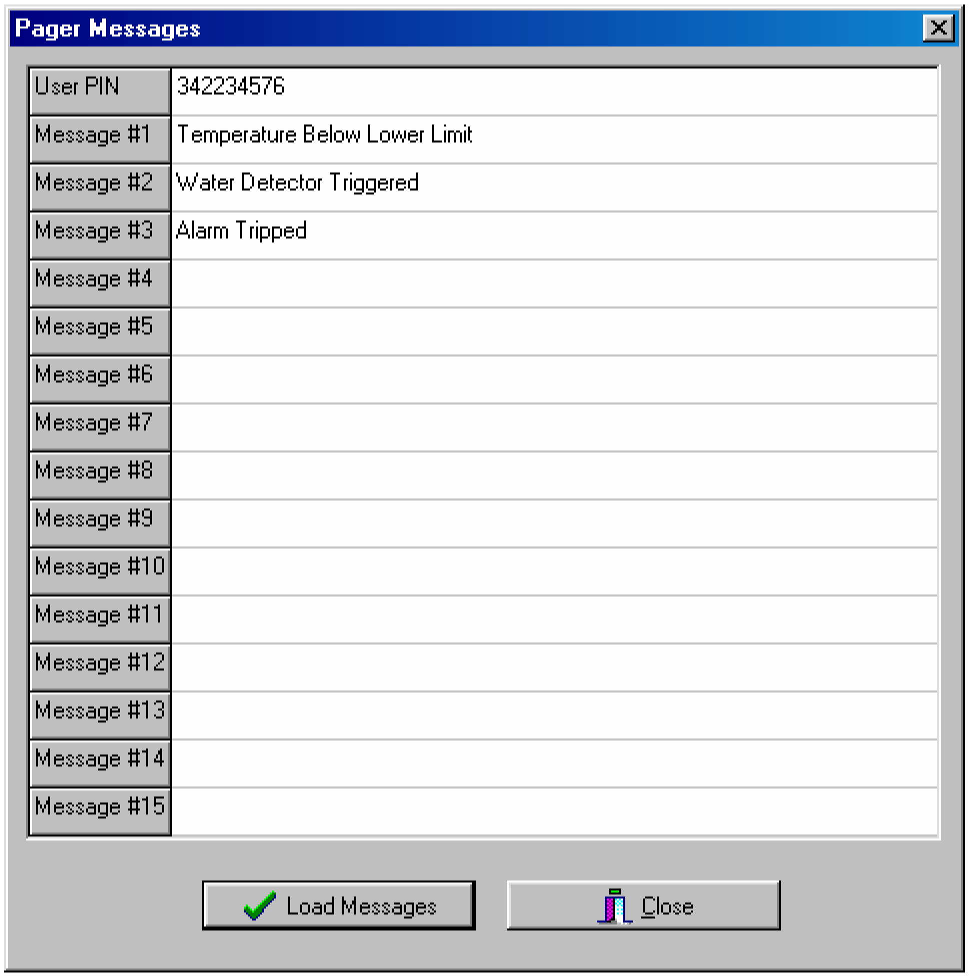
The master controller is also capable of transmitting ASCII messages, but that utility is not provided here. This is because the controller access utilities need the controller's serial port to operate. Transmitting an ASCII message through this same port would conflict with the controller access commands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

4.5.5.5 Serial Messages

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

4.5.5.5.1 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 35):



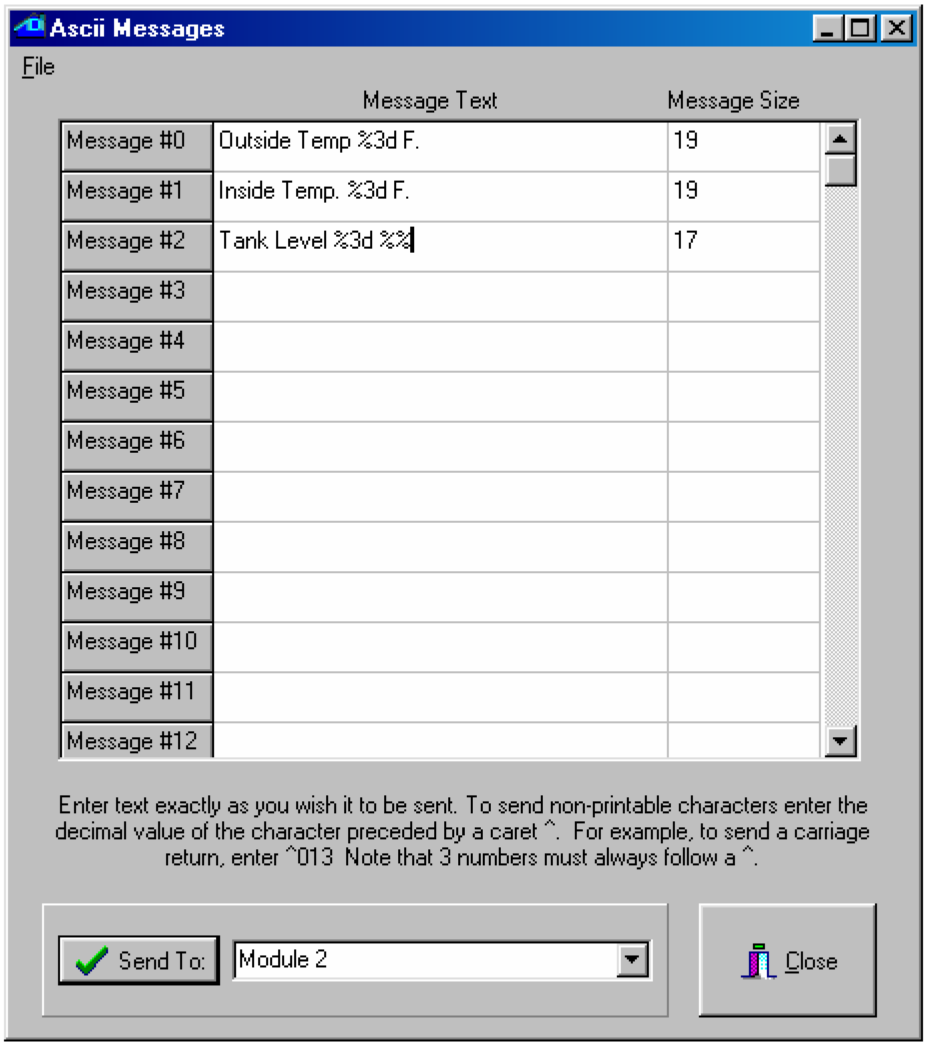
User PIN	342234576
Message #1	Temperature Below Lower Limit
Message #2	Water Detector Triggered
Message #3	Alarm Tripped
Message #4	
Message #5	
Message #6	
Message #7	
Message #8	
Message #9	
Message #10	
Message #11	
Message #12	
Message #13	
Message #14	
Message #15	

Fig. 35

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.

4.5.5.5.2 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 36):



	Message Text	Message Size
Message #0	Outside Temp %3d F.	19
Message #1	Inside Temp. %3d F.	19
Message #2	Tank Level %3d %%	17
Message #3		
Message #4		
Message #5		
Message #6		
Message #7		
Message #8		
Message #9		
Message #10		
Message #11		
Message #12		

Enter text exactly as you wish it to be sent. To send non-printable characters enter the decimal value of the character preceded by a caret ^. For example, to send a carriage return, enter ^013 Note that 3 numbers must always follow a ^.

Send To: Module 2

Fig. 36

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 Ocelot. 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 ASCII messages.

- 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 application note will show a sample program where variables are formatted to obtain the desired representation. The example will show formatting of 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 +....+ 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 3276. 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 in ASCII strings 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 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 ASCII strings (Fig 37). 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. 37

The above formats can be modified with one or more of the following options (these are inserted between the % and the letter) (Fig. 38):

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

Fig. 38

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 then 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. 39).

```
Format      Variable    Appearance

%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. 39

5.1.3 Formatting Variables in ASCII Strings

Our 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. 40):

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

Where:

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

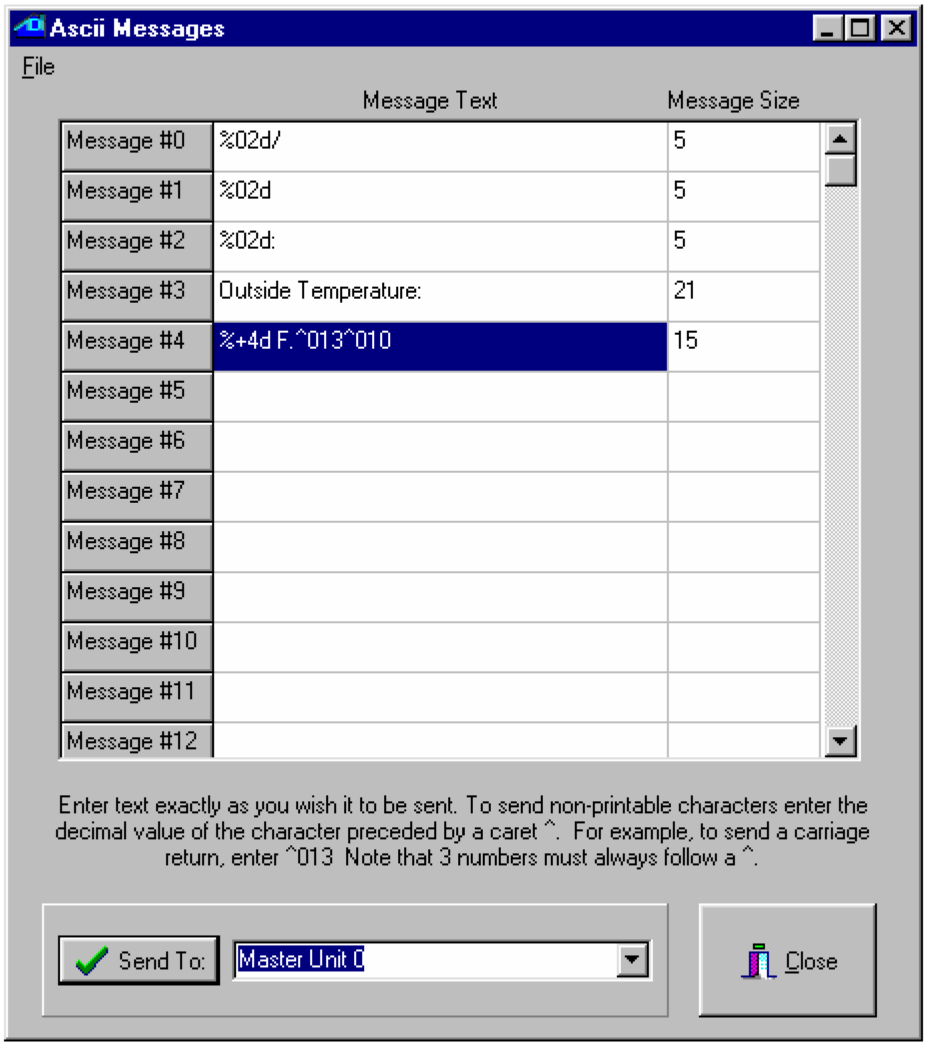
Fig. 40

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 then 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 "     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: "      literal text
"%+4d F.^013^010"      temperature always with sign and line end chars
```

Fig. 41

Here are the strings as they are entered into the ASCII strings definition utility (Fig. 42)



	Message Text	Message Size
Message #0	%02d/	5
Message #1	%02d	5
Message #2	%02d:	5
Message #3	Outside Temperature:	21
Message #4	%+4d F.^013^010	15
Message #5		
Message #6		
Message #7		
Message #8		
Message #9		
Message #10		
Message #11		
Message #12		

Enter text exactly as you wish it to be sent. To send non-printable characters enter the decimal value of the character preceded by a caret ^. For example, to send a carriage return, enter ^013 Note that 3 numbers must always follow a ^.

Fig. 42

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. 43):

```
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. 43

In our programming 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. 44):

```
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. 44

Note that any single triggering event could be used as a trigger, such as a received X-10 command, (slave) 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. 45):

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

Fig. 45

Then line 36 (the END statement) of the program in Fig. 43 was removed and the following code was added (Fig. 46):

```
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. 46

Here is what a log from this new program could look like (Fig. 47):

```
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. 47

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 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.

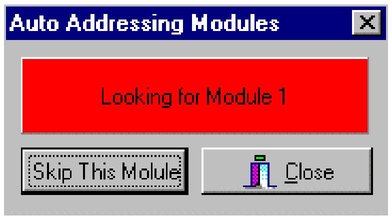
It is not necessary to remove the cover from the Ocelot™, Bobcat™ or SPEAKEZ™ modules.

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 48):



Looking for Module 1

Fig. 48

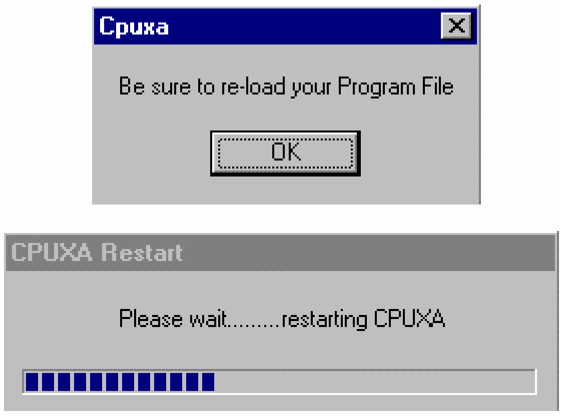
The Active light on each module will flash rapidly.

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 49): Click OK with your left mouse button.



Be sure to re-load your Program File

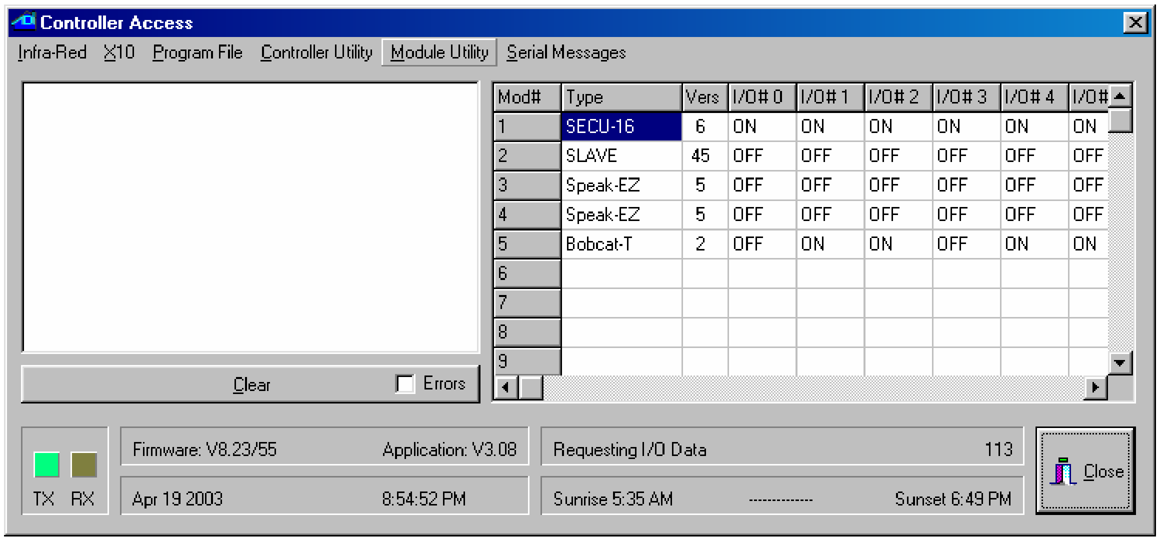
Please wait.........restarting CPUXA

Fig. 49

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 50):



Mod#	Type	Vers	I/O# 0	I/O# 1	I/O# 2	I/O# 3	I/O# 4	I/O#
1	SECU-16	6	ON	ON	ON	ON	ON	ON
2	SLAVE	45	OFF	OFF	OFF	OFF	OFF	OFF
3	Speak-EZ	5	OFF	OFF	OFF	OFF	OFF	OFF
4	Speak-EZ	5	OFF	OFF	OFF	OFF	OFF	OFF
5	Bobcat-T	2	OFF	ON	ON	OFF	ON	ON
6								
7								
8								
9								

Firmware: V8.23/55 Application: V3.08
Apr 19 2003 8:54:52 PM
Requesting I/O Data 113
Sunrise 5:35 AM ------------ Sunset 6:49 PM

Fig. 50

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.

5.2.1 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. 51).

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

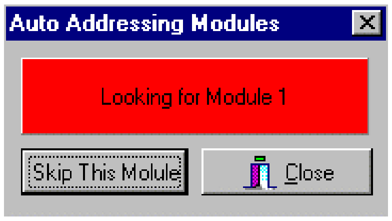


Fig. 51

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.

5.3 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 then 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 tieing 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 siwtch's contacts. To actuate the door's operation, a SECU16 output relay is wired in parallel with the door's wall mounted pushbutton. Fig. 52 shows a schematic diagram of the wiring for both the input and the output circuits to interface to the garage door system.

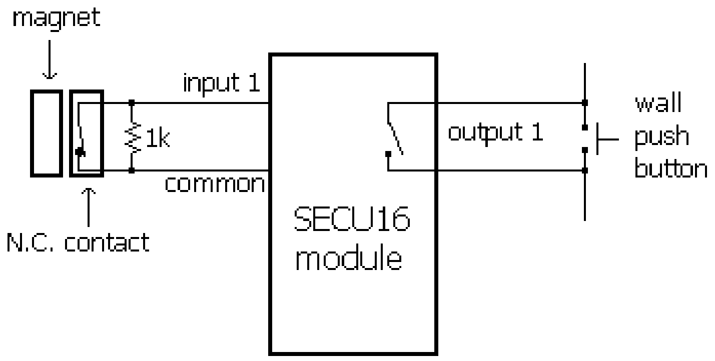


Fig. 52

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. 53):

```
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. 53

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. 54):

```
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. 54

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. 55 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.

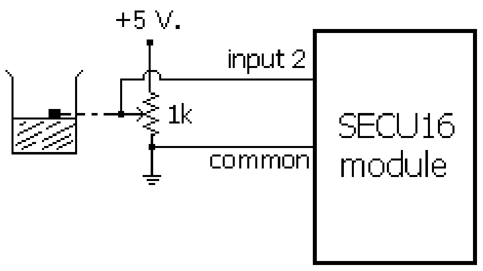


Fig. 55

The code needed to read to read the input and convert it to a percentage is shown in Fig. 56. 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. 56

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.

5.4 Ocelot/Leopard Parameters

Param Number	Description of Parameter	Values / Usage
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 (Leopard)	* 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 may 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 (Leopard)	* 0 = return touch object numbers only * 1 = return touch object number followed by physical grid location number