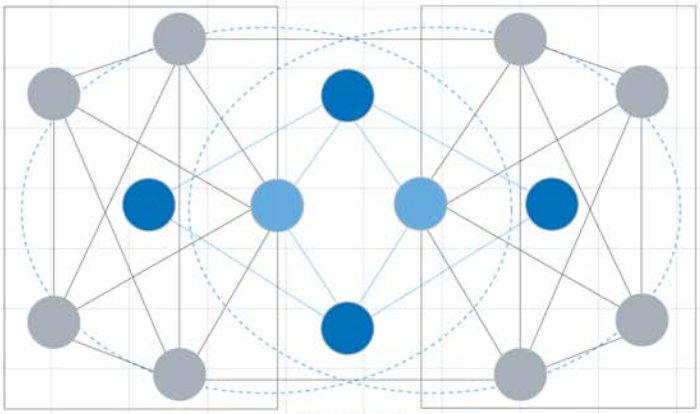
INSTEON

Developer's Guide



SMARTHOME technology

October 14, 2005

Contents at a Glance

Full Table of Contents

Publication Dates

Date	Author	Change
06-21-05	Bill Morgan	Initial release.
10-14-05	Paul Darbee	Release for comment. Major editing and additions.

INTRODUCTION

A TV automatically turns on the surround sound amplifier, a smart microwave oven downloads new cooking recipes, a thermostat automatically changes to its energy saving setpoint when the security system is enabled, bathroom floors and towel racks heat up when the bath runs, an email alert goes out when there is water in the basement. When did the Jetson-style home of the future become a reality? When INSTEON™—the new technology standard for advanced home control—arrived. INSTEON enables product developers to create these distinctive solutions for homeowners, and others yet unimagined, by delivering on the promise of a truly connected 'smart home.'

INSTEON is a cost-effective dual band network technology optimized for home management and control. INSTEON-networked Electronic Home Improvement™ products can interact with one another, and with people, in new ways that will improve the comfort, safety, convenience and value of homes around the world.

For a brief introduction to INSTEON see *INSTEON Overview*.

This Developer's Guide is part of the INSTEON Software and Hardware Development Kits that Smarthome provides to Independent Software Vendors (ISVs) and Original Equipment Manufacturers (OEMs) who wish to create software and hardware systems that work with INSTEON.

In This INSTEON Developer's Guide

INSTEON BASICS

In This Section

Getting Started Quickly
Points out the highlights of this Developer's Guide for those who wish to start coding as quickly as possible.

About This Developer's Guide
Discusses document conventions, where to get support, and legal information.

INSTEON Overview
Familiarizes you with the background, design goals, and capabilities of INSTEON.

INSTEON Application Development Overview
Explains how developers can create applications that orchestrate the behavior of INSTEON-networked devices.

Getting Started Quickly

INSTEON devices communicate by sending INSTEON messages. *The Smarthome PowerLinc Controller* (PLC) is an INSTEON device that also has a serial port that you connect to your PC. You can write applications that run on the PLC using tools documented in the *SALad Integrated Development Environment User's Guide*.

What to Look at First

For an accelerated introduction to using the PLC and SALad to control and program INSTEON devices, follow these steps in sequence:

1. Review the *INSTEON SALad and PowerLinc Controller Architecture* and *INSTEON Device Communication* diagrams to see how things fit together.
2. Review the *IBIOS Serial Communication Protocol and Settings* section to see how the serial protocol works.
3. Review the *IBIOS Serial Command Examples* section to see how to use IBIOS Serial Commands directly.
4. Review the *SALad IDE Quickstart* section.
5. Review the *INSTEON Messages* section for more detailed information on the INSTEON protocol.
6. Review the *INSTEON Network Usage* section for details about INSTEON commands, device classes, device linking using Groups, and security issues.

Summary Tables

Sections of this Developer's Guide that you will reference often are:

1. The *INSTEON Message Summary Table*, which enumerates all possible INSTEON message types.
2. The *INSTEON Common Command Summary Table* and *INSTEON Broadcast Command Summary Table*, which enumerate all of the INSTEON Commands that can appear in INSTEON messages.
3. The *IBIOS Event Summary Table*, which enumerates all of the events that IBIOS can generate.
4. The *IBIOS Serial Command Summary Table*, which enumerates all of the commands for interacting serially with the PLC.
5. The *Flat Memory Map*, which shows where everything is in the PLC's memory.
6. The *SALad Instruction Summary Table*, which lists the SALad instruction set.

About This Developer's Guide

In This Section

About the Documentation

Document Conventions

The following table shows the typographic conventions used in this INSTEON Developer's Guide.

Convention	Description	Example
Monospace	Indicates source code, code examples, code lines embedded in text, and variables and code elements	`DST EQU 0x0580`
Angle Brackets < and >	Indicates user-supplied parameters	<Address MSB>
Vertical Bar |	Indicates a choice of one of several alternatives	<0x06 (ACK) | 0x15 (NAK)>
Ellipsis …	Used to imply additional text	"Text …"
0xNN	Hexadecimal number	0xFF, 0x29, 0x89AB
⇒	Range of values, *including* the beginning and ending values	0x00 ⇒ 0x0D

Getting Help

INSTEON Support

Smarthome is keenly interested in supporting the community of INSTEON developers. If you are having trouble finding the answers you need in this INSTEON Developer's Guide, you can get more help by accessing the INSTEON Developer's Forum or by emailing sdk@insteon.net.

INSTEON Developer's Forum

When you purchased the INSTEON Software Developer's Kit, you received a username and password for accessing the Developer's Forum at http://insteon.net/sdk/forum. The Forum contains a wealth of information for developers, including

- Frequently asked questions
- Software downloads, including the SALad IDE (Integrated Development Environment), and Smarthome Device Manager
- Documentation updates
- Sample code
- Discussion forums

Providing Feedback

To provide feedback about this documentation, go to the INSTEON Developer's Forum or send email to sdk@insteon.net.

Legal Information

Terms of Use

This INSTEON Developer's Guide is supplied to you by Smarthome, Inc. (Smarthome) in consideration of your agreement to the following terms. Your use or installation of this INSTEON Developer's Guide constitutes acceptance of these terms. If you do not agree with these terms, please do not use or install this INSTEON Developer's Guide.

In consideration of your agreement to abide by the following terms, and subject to these terms, Smarthome grants you a personal, non-exclusive license, under Smarthome's intellectual property rights in this INSTEON Developer's Guide, to use this INSTEON Developer's Guide; provided that no license is granted herein under any patents that may be infringed by your works, modifications of works, derivative works or by other works in which the information in this INSTEON Developer's Guide may be incorporated. No names, trademarks, service marks or logos of Smarthome, Inc. or INSTEON may be used to endorse or promote products derived from the INSTEON Developer's Guide without specific prior written permission from Smarthome, Inc. Except as expressly stated herein, no other rights or licenses, express or implied, are granted by Smarthome and nothing herein grants any license under any patents except claims of Smarthome patents that cover this INSTEON Developer's Guide as originally provided by Smarthome, and only to the extent necessary to use this INSTEON Developer's Guide as originally provided by Smarthome. Smarthome provides this INSTEON Developer's Guide on an "AS IS" basis.

SMARTHOME MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THIS INSTEON DEVELOPER'S GUIDE OR ITS USE, ALONE OR IN COMBINATION WITH ANY PRODUCT.

IN NO EVENT SHALL SMARTHOME BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) ARISING IN ANY WAY OUT OF THE USE, REPRODUCTION, MODIFICATION AND/OR DISTRIBUTION OF THIS INSTEON DEVELOPER'S GUIDE, HOWEVER CAUSED AND WHETHER UNDER THEORY OF CONTRACT, TORT (INCLUDING NEGLIGENCE), STRICT LIABILITY OR OTHERWISE, EVEN IF SMARTHOME HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Trademarks and Patents

Smarthome, INSTEON, PowerLinc, ControLinc, LampLinc, SwitchLinc, RemoteLinc, Electronic Home Improvement, Smarthome Device Manager, Home Network Language, and Plug-n-Tap are trademarks of Smarthome, Inc.

INSTEON networking technology is covered by pending U.S. and foreign patents.

Copyright

Revision History

Release Date	Author	Description
10/5/04	BM	Initial Draft.
10/19/04	BM	Alpha release of SDK.
10/19/04	BM	Added Flags to the Memory Map.
10/22/04	BM	Removed SALad NTL documentation and put in external document. Editing changes.
10/27/04	BM	Editorial changes. Fixed version in header to match version on title page.
11/02/04	BM	Updated serial command section. Created index and started adding entries. Updated section 6 (INSTEON Messages). Added Device Types. Removed Examples section, and put example code in external directory.
11/05/04	BM	Editorial Changes.
11/15/04	BM	Editorial Changes.
11/22/04	BM	Editorial Changes. Reformatted document. Removed Warning and Reserved areas from Flat Memory M. Added new memory map for flat / absolute address space. Added serial commands with examples relative to previous serial commands.
11/29/04	BM	Added changes from technical review. Removed CRC from INSTEON message descriptions. Added SALad Assembler documentation. Added USB notes.
1/3/05	BM	Added changes to Peek direct command description to include address auto-increment feature. Updated device types. Reformatted document. Added SALad detailed instructions. Updated Link Data Table. Added table for NAK reasons. Added INSTEON enrollment description. Initial release to select beta reviewers.
1/14/05	BM	Removed CRC columns and rows of message tables and rephrased CRC description near message tables. Added Bit 0 for the Total Hops and rephrased hops descriptions.
1/21/05	BM	Added IBIOS Serial Command Examples.
1/31/05	BM	Added more PowerLinc Serial examples. Updated SALad Event Handling. Updated hypertext bookmark links. Editing changes.
2/3/05	BM	Updated Flat Memory M.
2/10/05	BM	Updated serial command checksum description. Updated Control and RS_Control registers in Flat Memory M.
3/14/05	BM	Redesign of Device Types and Network Management. Document formatting changes. Updated Flat Memory M. Updated SALad SALad Instruction Summary Table and SALad Parameter Encoding. Updated INSTEON Command Tables.
3/15/05	BM	Updated INSTEON Command Tables. Added latest IDE documentation.
3/18/05	BM	Updated SALad Instruction Summary. Removed AJUMP, ACALL, PAUSE, APROC. Updated command codes for SEND$, SENDEXT$, RANDOMIZE. Renamed RJUMP to JUMP, RCALL to CALL, RPROC to PROC. Added new command ENROLLMENT. Editorial Changes

3/21/05	BM	Editorial changes. Updated Flat Memory M. Updated IBIOS Serial Command Examples.
3/23/05	BM	Removed blank topic headings.
3/28/05	BM	Updated Flat Memory M for PowerLinc Controller version 2.6. Added EventMask 0x016F register, and updated I_Control register 0x0142.
4/11/05	BM	Updated SALad Instruction Summary. Updated Flat Memory M.
4/13/05	BM	Added Receive INSTEON example. Updated serial communications sections to remove 0x0d transmission.
4/21/05	BM	Updated data returned by Get Version serial command (0x48). Updated Event Report serial command (0x45).
5/16/05	BM	Updated SALad TEST instruction to "jump if false". Updated SALad FIND instruction description.
6/3/05	BM	Updated SALad Flat Memory M. Added Debug Report to INSTEON Broadcast Commands section. Updated Event Table.
6/15/05	BM	Added INSTEON Hardware Development Documentation.
6/20/05	BV	Updated product names referenced in document. Fixed error in description of SALad FIND instruction for finding Master or Slave device. Fixed error in Peek and Poke Notes (there is only one Set Hi command 0x27).
6/21/05	BV	Updated the FIND instruction (find empty returns a pointer to DB_H, not DB_0).
7/21/05	BV	Updated Device Type 0x0007 to 'Reserved for future use'. Used ⇒, instead of '-' to mean an address range to avoid confusion with the minus symbol. Updated command 0x48. Updated references to Controller and Responder. Updated DB_0 ⇒ DB_8 to clarify message construction.
8/16/05	PVD	Began major rewrite, including additional material and reformatting.
10/7/05	PVD	Pre-release for comment. Major editing and additions, but incomplete. Areas needing attention marked.
10/14/05	PVD	Release for comment. Major editing and additions.

INSTEON Overview

INSTEON enables simple, low-cost devices to be networked together using the powerline, radio frequency (RF), or both. All INSTEON devices are peers, meaning that any device can transmit, receive, or repeat[1] other messages, without requiring a master controller or complex routing software. Adding more devices makes an INSTEON network more robust, by virtue of a simple protocol for communication retransmissions and retries. On the powerline, INSTEON devices are compatible[2] with legacy X10 devices.

This section explains why INSTEON has these properties and explains them further without going into the details.

In this section

Why INSTEON?
Explains why Smarthome undertook the development of INSTEON.

Hallmarks of INSTEON
Gives the 'project pillars' and main properties of INSTEON.

INSTEON Specifications
Shows the main features of INSTEON in table form.

INSTEON Fundamentals
Shows how INSTEON devices communicate using both powerline and radio, how all INSTEON devices repeat[1] INSTEON messages, and how all INSTEON devices are peers.

INSTEON Application Development Overview
Explains how developers can create applications that orchestrate the behavior of INSTEON-networked devices.

Why INSTEON?

INSTEON is the creation of Smarthome, the world's leading authority on electronic home improvement. Smarthome is organized into three divisions—Smarthome.com, "the Amazon of electronic home improvement," Smarthome Design, creators of best-in-class home control products, and Smarthome Technology, the pioneering architects of INSTEON. With Smarthome.com's global distribution channel, Smarthome Design's product development and manufacturing resources, and Smarthome Technology's ongoing innovation, Smarthome is uniquely positioned to support and encourage INSTEON product developers.

But why did Smarthome undertake the complex task of creating an entirely new home-control networking technology in the first place?

Smarthome has been a leading supplier of devices and equipment to home automation installers and enthusiasts since 1992. Now selling over 5,000 products into more than 130 countries, Smarthome has first-hand experience dealing directly with people all over the world who have installed lighting control, whole-house automation, security and surveillance systems, pet care devices, gadgets, and home entertainment equipment. Over the years, by talking to thousands of customers through its person-to-person customer support operation, Smarthome has become increasingly concerned about the mismatch between the dream of living in a responsive, aware, automated home and the reality of existing home-control technologies.

Today's homes are stuffed with high-tech appliances, entertainment gear, computers, and communications gadgets. Utilities, such as electricity, lighting, plumbing, heating and air conditioning are so much a part of modern life that they almost go unnoticed. But these systems and devices all act independently of each other—there still is nothing that can link them all together. Houses don't know that people live in them. Lights happily burn when no one needs them, HVAC is insensitive to the location and comfort of people, pipes can burst without anyone being notified, and sprinklers dutifully water the lawn even while it's raining.

For a collection of independent objects to behave with a unified purpose, the objects must be able to communicate with each other. When they do, new, sometimes-unpredictable properties often emerge. In biology, animals emerged when nervous systems evolved. The Internet emerged when telecommunications linked computers together. The global economy emerges from transactions involving a staggering amount of communication. But there is no such communicating infrastructure in our homes out of which we might expect new levels of comfort, safety and convenience to emerge. There is nothing we use routinely in our homes that links our light switches or our door locks, for instance, to our PCs or our remote controls.

It's not that such systems don't exist at all. Just as there were automobiles for decades before Henry Ford made cars available to everyone, there are now and have been for some time systems that can perform home automation tasks. On the high end, all kinds of customized systems are available for the affluent, just as the rich could buy a Stanley Steamer or a Hupmobile in the late 1800s. At the low end, X10 powerline signaling technology has been around since the 1970s, but its early adoption is its limiting factor—it is too unreliable and inflexible to be useful as an infrastructure network.

Smarthome is a major distributor of devices that use X10 signaling. In 1997, aware of the reliability problems its customers were having with X10 devices available at

the time, Smarthome developed and began manufacturing its own 'Linc' series of improved X10 devices, including controllers, dimmers, switches, computer interfaces and signal boosters. Despite the enhanced performance enjoyed by Linc products, it was still mostly do-it-yourselfers and hobbyists who were buying and installing them.

Smarthome knew that a far more robust and flexible networking standard would have to replace X10 before a truly intelligent home could emerge. Smarthome wanted a technology that would meet the simplicity, reliability, and cost expectations of the masses—mainstream consumers who want immediate benefits, not toys.

In 2001, Smarthome's engineers were well aware of efforts by others to bring about the home of the future. The aging X10 protocol was simply too limiting with its tiny command set and unacknowledged, 'press and pray' signaling over the powerline. CEBus had tried to be everything to everybody, suffering from high cost due to overdesign by a committee of engineers. Although CEBus did become an official standard (EIA-600), developers did not incorporate it into real-world products.

Radio-only communication protocols, such as Z-Wave and ZigBee, not only required complex routing strategies and a confusing array of different types of network masters, slaves, and other modules, but radio alone might not be reliable enough when installed in metal switch junction boxes or other RF-blocking locations.

BlueTooth radio has too short a range, WiFi radio is too expensive, and high-speed powerline protocols are far too complex to be built into commodity products such as light switches, door locks, or thermostats. Overall, it seemed that everything proposed or available was too overdesigned and therefore would cost too much to become a commodity for the masses in the global economy.

So, in 2001, Smarthome decided to take its destiny into its own hands and set out to specify an ideal home control network, one that would be simple, robust and inexpensive enough to link everything to everything else. INSTEON was born.

Hallmarks of INSTEON

These are the project pillars that Smarthome decided upon to guide the development of INSTEON. Products networked with INSTEON had to be:

Instantly Responsive

INSTEON devices respond to commands with no perceptible delay. INSTEON's signaling speed is optimized for home control—fast enough for quick response, while still allowing reliable networking using low-cost components.

Easy to Install

Installation in existing homes does not require any new wiring, because INSTEON products communicate over powerline wires or they use the airwaves. Users never have to deal with network enrollment issues because all INSTEON devices have an ID number pre-loaded at the factory—INSTEON devices join the network as soon as they're powered up.

Simple to Use

Getting one INSTEON device to control another is very simple—just press and hold a button on each device for 10 seconds, and they're linked. Because commands are confirmed, INSTEON products can provide instant feedback to the user, making them straightforward to use and 'guest friendly.'

Reliable

An INSTEON network becomes more robust and reliable as it is expanded because every INSTEON device repeats[1] messages received from other INSTEON devices. Dual band communications using both the powerline and the airwaves ensures that there are multiple pathways for messages to travel.

Affordable

INSTEON software is simple and compact, because all INSTEON devices send and receive messages in exactly the same way, without requiring a special network controller or complex routing algorithms. The cost of networking products with INSTEON is held to at an absolute minimum because INSTEON is designed specifically for home control applications, and not for transporting large amounts of data.

Compatible with X10

INSTEON and X10 signals can coexist with each other on the powerline without mutual interference. Designers are free to create hybrid INSTEON/X10 products that operate equally well in both environments, allowing current users of legacy X10 products to easily upgrade to INSTEON without making their investment in X10 obsolete.

INSTEON Specifications

The most important property of INSTEON is its no-frills simplicity.

INSTEON messages are fixed in length and synchronized to the AC powerline zero crossings. They do not contain routing information beyond a source and destination address. INSTEON is reliable and affordable because it is optimized for command and control, not high-speed data transport. INSTEON allows infrastructure devices like light switches and sensors to be networked together in large numbers, at low cost. INSTEON stands on its own, but can also bridge to other networks, such as WiFi LANs, the Internet, telephony, and entertainment distribution systems. Such bridging allows INSTEON to be part of very sophisticated integrated home control environments.

The following table shows the main features of INSTEON at a glance.

INSTEON Property	Specification	
Network	Dual band (RF and powerline) Peer-to-Peer Mesh Topology Unsupervised No routing tables	
Protocol	All devices are two-way Repeaters[1] Messages acknowledged Retry if not acknowledged Synchronized to powerline	
X10 Compatibility[2]	INSTEON devices can send and receive X10 commands INSTEON devices do not repeat or amplify X10	
Data Rate	Instantaneous	13,165 bits/sec
	Sustained	2,880 bits/sec
Message Types	Standard	10 Bytes
	Extended	24 Bytes
Message Format, Bytes	From Address	3
	To Address	3
	Flags	1
	Command	2
	User Data	14 (Extended Messages only)
	Message Integrity	1
Devices Supported	Unique IDs	16,777,216
	Device Types	65,536
	Commands	65,536
	Groups per Device	256
	Members within a Group	Limited only by memory
INSTEON Engine Memory Requirements	RAM	80 Bytes
	ROM	3K Bytes

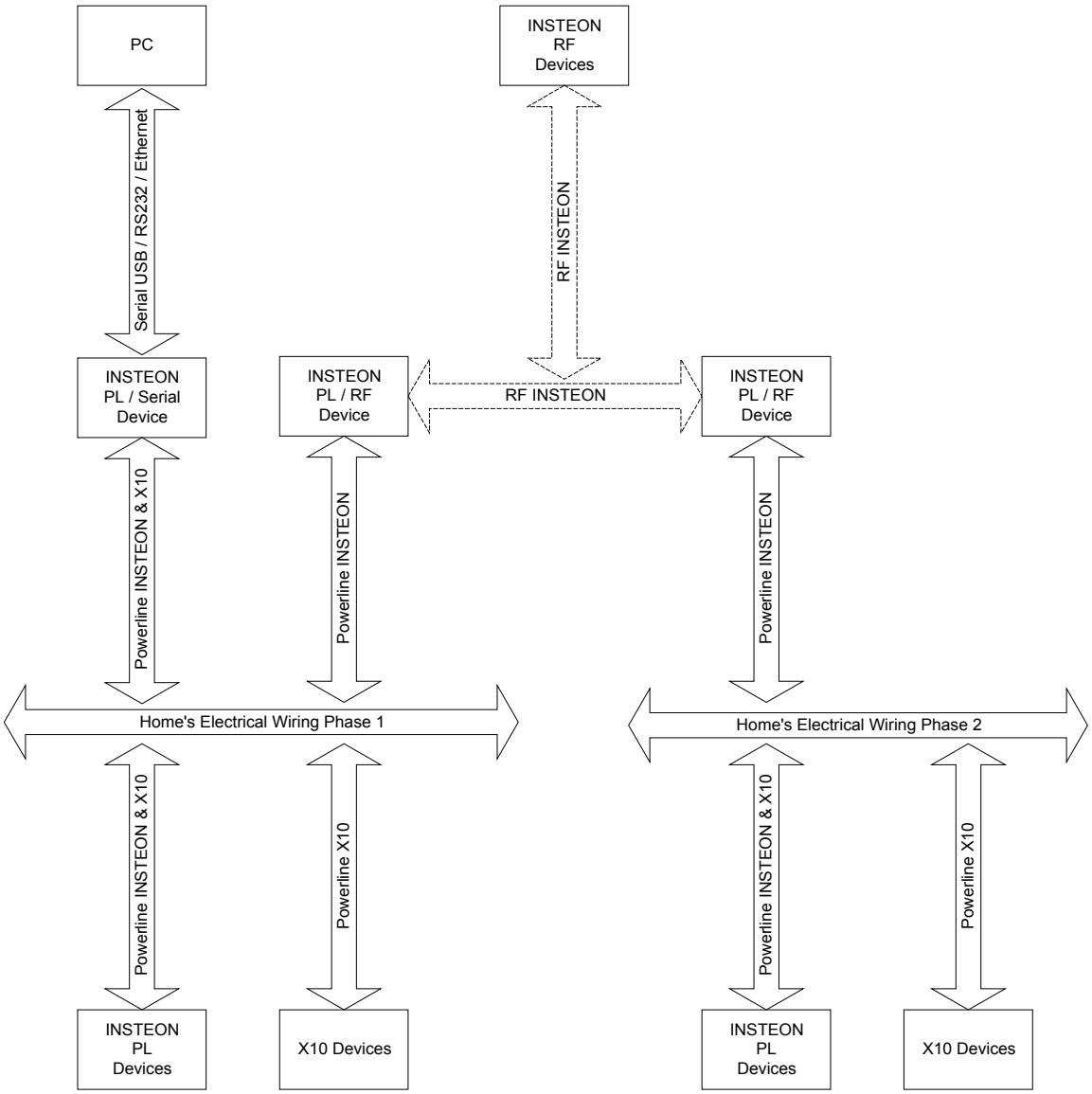
INSTEON Property	Specification	
Typical INSTEON Application Memory Requirements (Light Switch, Lamp Dimmer)	RAM	256 Bytes
	EEPROM	256 Bytes
	Flash	7K Bytes
Device Installation	Plug-in Wire-in Battery Operated	
Device Setup	Plug-n-Tap™ manual linking PC or Controller	
Security	Physical device possession Address masking Encrypted message payloads	
Application Development	IDE (Integrated Development Environment) SALad interpreted language Software and Hardware Development Kits	
Powerline Physical Layer	Frequency	131.65 KHz
	Modulation	BPSK
	Min Transmit Level	3.16 Vpp into 5 Ohms
	Min Receive Level	10 mV
	Phase Bridging	INSTEON RF or hardware
RF Physical Layer	Frequency	904 MHz
	Modulation	FSK
	Sensitivity	-103 dbm
	Range	150 ft unobstructed line-of-sight

INSTEON Fundamentals

In this section

INSTEON Device Communication

Devices communicate with each other using the INSTEON protocol over the air via radio frequency (RF) and over the powerline as illustrated below.



Electrical power is most commonly distributed to homes in North America as split-phase 220-volt alternating current (220 VAC). At the main electrical junction box to the home, the single three-wire 220 VAC powerline is split into a pair of two-wire 110 VAC powerlines, known as Phase 1 and Phase 2. Phase 1 wiring usually powers half the circuits in the home, and Phase 2 powers the other half.

INSTEON devices communicate with each other over the powerline using the INSTEON Powerline protocol, which will be described in detail below (see *INSTEON Messages* and *INSTEON Signaling Details*).

Existing X10 devices also communicate over the powerline using the X10 protocol. The INSTEON Powerline protocol is compatible[2] with the X10 protocol, meaning that designers can create INSTEON devices that can also listen and talk to X10 devices. X10 devices, however, are insensitive to the INSTEON Powerline protocol.

INSTEON devices containing RF hardware may optionally communicate with other INSTEON RF devices using the INSTEON RF protocol.

INSTEON devices that can use *both* the INSTEON Powerline protocol and the INSTEON RF protocol solve a significant problem encountered by devices that can only communicate via the powerline. Powerline signals originating on the opposite powerline phase from a powerline receiver are severely attenuated, because there is no direct circuit connection for them to travel over.

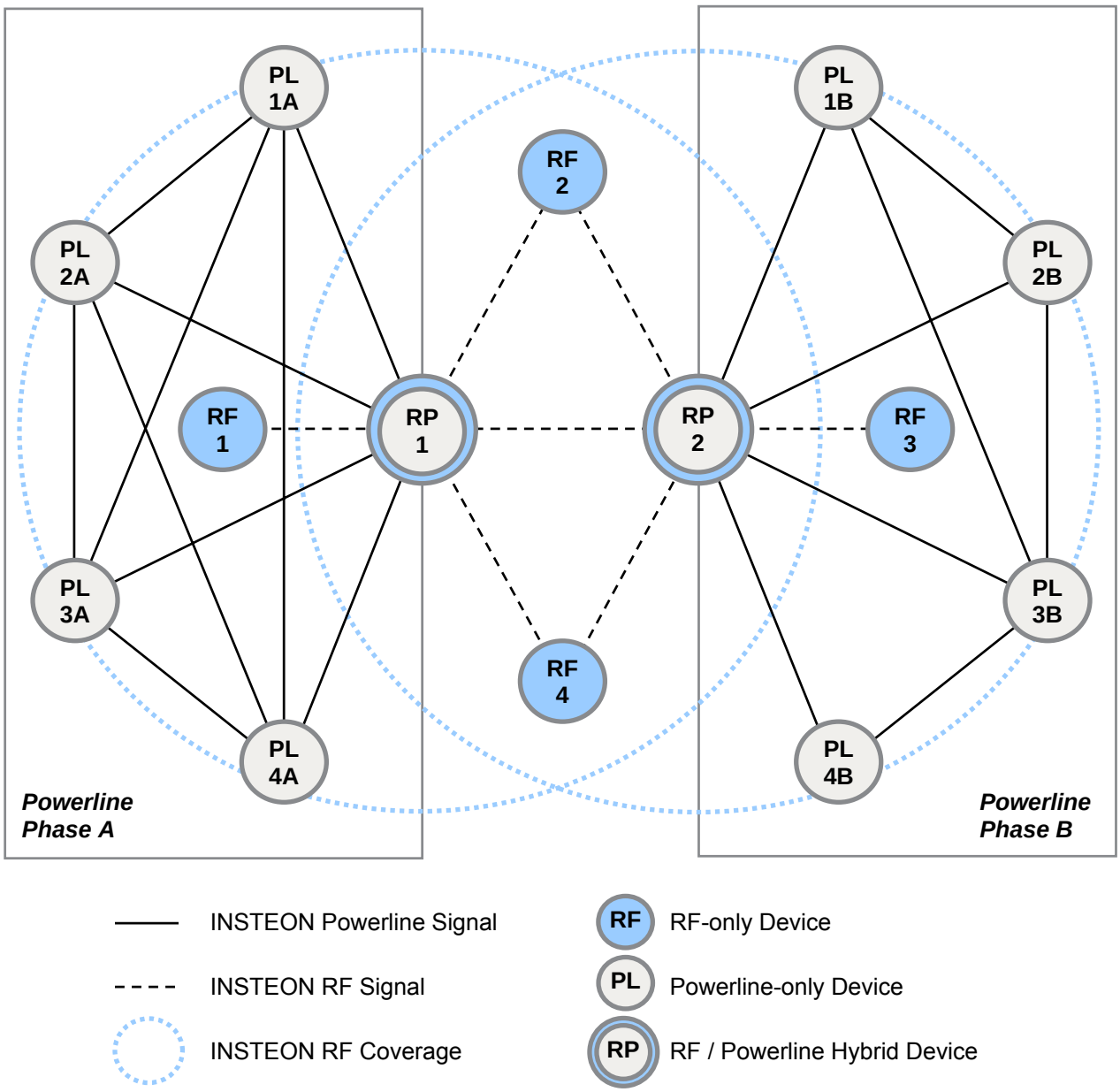
A traditional solution to this problem is to connect a signal coupling device between the powerline phases, either by hardwiring it in at a junction box or by plugging it into a 220 VAC outlet. INSTEON automatically solves the powerline phase coupling problem through the use of INSTEON devices capable of both powerline and RF messaging. INSTEON RF messaging bridges the powerline phases whenever at least one INSTEON PL/RF device is installed on each powerline phase.

When suitably equipped with a dedicated serial interface, such as USB, RS232, or Ethernet, INSTEON devices can also interface with computers and other digital equipment. In the figure above, an INSTEON device is shown communicating with a PC using a serial link. In the Software Developer's Kit, that device is a Smarthome PowerLinc™ V2 Controller with a USB or RS232 interface (see *The Smarthome PowerLinc Controller* and *INSTEON BIOS*).

Serial communications can bridge networks of INSTEON devices to otherwise incompatible networks of devices in a home, to computers, to other nodes on a local-area network (LAN), or to the global Internet. Such connections to outside resources allow networks of INSTEON devices to exhibit complex, adaptive, people-pleasing behaviors. INSTEON devices capable of running downloadable SALad Applications (see *SALad Language Documentation*) can be upgraded to perform very sophisticated functions, including functions not envisioned at the time of manufacture or installation.

INSTEON Message Repeating

The figure below shows how network reliability improves when additional INSTEON devices are added. The drawing shows INSTEON devices that communicate by powerline-only (PL), RF-only (RF), and both (RP).



Every INSTEON device is capable of repeating[1] INSTEON messages. They will do this automatically as soon as they are powered up—they do not need to be specially installed using some network setup procedure. Adding more devices increases the number of available pathways for messages to travel. This *path diversity* results in a higher probability that a message will arrive at its intended destination, so the more devices in an INSTEON network, the better.

As an example, suppose RF device RF1 desires to send a message to RF3, but RF3 is out of range. The message will still get through, however, because devices within range of RF1, say RP1 and RF2, will receive the message and retransmit it to other devices within range of themselves. In the drawing, RP1 might reach RF2, RP2, and RF4, and devices RP2 and RF1 might be within range of the intended recipient, RF3. Therefore, there are many ways for a message to travel: RF1 to RF2 to RF3 (1 retransmission), RF1 to RP1 to RP2 to RF3 (2 retransmissions), and RF1 to RP1 to RF2 to RP2 to RF3 (3 retransmissions) are some examples.

On the powerline, path diversity has a similar beneficial effect. For example, the drawing shows powerline device PL1B without a direct communication path to device PL4B. In the real world, this might occur because of signal attenuation problems or because a direct path through the electric wiring does not exist. But a message from PL1B will still reach PL4B by taking a path through RP2 (1 retransmission), through PL2B to RP2 (2 retransmissions), or through PL2B to RP2 to PL3B (3 retransmissions).

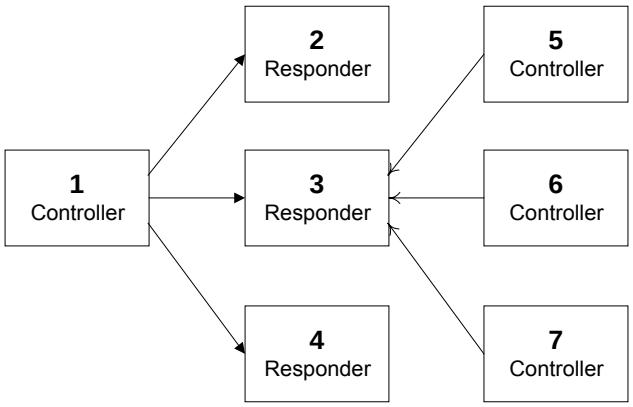
The figure also shows how messages can travel among powerline devices that are installed on different phases of a home's wiring. To accomplish phase bridging, at least one INSTEON hybrid RF/powerline device must be installed on each powerline phase. In the drawing, hybrid device RP1 is installed on phase A and RP2 is installed on phase B. Direct RF paths between RP1 to RP2, or indirect paths using RF2 or RF4 (1 retransmission) allow messages to propagate between the powerline phases, even though there is no direct electrical connection.

With all devices repeating messages, there must be some mechanism for limiting the number of times that a message may be retransmitted, or else messages might propagate forever within the network. Network saturation by repeating messages is known as a 'data storm.' The INSTEON protocol avoids this problem by limiting the maximum number times an individual message may be retransmitted to three (see *INSTEON Message Hopping*).

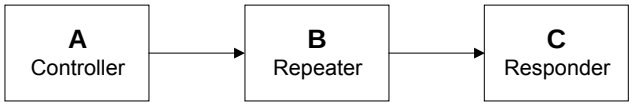
INSTEON Peer-to-Peer Networking

All INSTEON devices are peers, meaning that any device can act as a Controller (sending messages), Responder (receiving messages), or Repeater[1] (relaying messages).

This relationship is illustrated in the figure below, where INSTEON device **1**, acting as a Controller, sends messages to multiple INSTEON devices **2**, **3**, and **4** acting as Responders. Multiple INSTEON devices **5**, **6**, and **7** acting as Controllers can also send messages to a single INSTEON device **3** acting as a Responder.



Any INSTEON device can repeat[1] messages, as with device **B**, below, which is shown relaying a message from device **A** acting as a Controller to device **C** acting as a Responder.



INSTEON Application Development Overview

INSTEON, with its no-nonsense emphasis on simplicity, reliability, and low cost, is optimized as an *infrastructure* network. Common devices in the home, such as light switches, door locks, thermostats, clocks, and entertainment systems currently do not communicate with one another. INSTEON can change all that.

When devices are networked together, there is a potential for coordinated, adaptive behavior that can bring a new, higher level of comfort, safety, and convenience to living. But networking devices together cannot by itself change the behavior of the devices. It is application-level software, created by developers, that transforms a network of previously unrelated devices into a coordinated, adaptive, lifestyle-enhancing system.

There are two basic kinds of applications that developers can create for INSTEON-networked devices: External Applications and Internal Applications.

External Applications run on a computing device such as a PC or PDA. A special type of INSTEON module called an INSTEON *Bridge* connects the computing device to an INSTEON network. *Manager Apps* are External Applications that exchange INSTEON messages directly with INSTEON devices via a Bridge.

Internal Applications run on INSTEON devices themselves. Smarthome has developed an embedded language interpreter, called SALad, which resides in the firmware of SALad-enabled INSTEON devices. Developers can create and debug *SALad Apps* in a Smarthome Integrated Development Environment (IDE) that communicates with INSTEON devices via an INSTEON Bridge. Devices running SALad Apps can exhibit very sophisticated behavior. Moreover, devices that have already been installed in the home can be upgraded by downloading new SALad Apps to them. With INSTEON upgradeability, the world of home control can dynamically adapt to people's expectations and needs as the marketplace evolves.

In this section

Interfacing to an INSTEON Network

An INSTEON device that connects an INSTEON network to the outside world is called an *INSTEON Bridge*. There can be many kinds of INSTEON Bridges. One kind, an INSTEON-to-Serial Bridge, connects an INSTEON network to a computing device like a PC, a PDA, or to a dedicated user interface device with a serial port. Another Bridge, INSTEON-to-IP, connects an INSTEON network to a LAN or the Internet, either with wires (like Ethernet) or wirelessly (like WiFi). Still other INSTEON Bridges could connect to other networks such as wired or wireless telephony, Bluetooth, ZigBee, WiMax, or whatever else emerges in the future.

The Smarthome PowerLinc Controller

The PowerLinc™ V2 Controller (PLC) from Smarthome is an example of an INSTEON-to-Serial Bridge for connecting an INSTEON network to a computing device. PLCs are currently available with either a USB or an RS232 serial interface. An Ethernet interface, for connecting to a LAN or the Internet, is under development. For comprehensive information about the firmware capabilities of the PLC, see the *INSTEON BIOS* section below.

Using the PLC, application developers can create high-level user interfaces to devices on an INSTEON network. *Manager Apps* are External Applications that run on a computing device and use the PLC to directly send and receive INSTEON messages to INSTEON devices. *SALad Apps* are Internal Applications that run on SALad-enabled INSTEON devices themselves. The PLC is a SALad-enabled INSTEON device, having a SALad language interpreter embedded in its firmware.

As shipped by Smarthome, the PLC contains a 1200-byte *SALad coreApp Program* that performs a number of useful functions:

- When coreApp receives messages from INSTEON devices, it sends them to the computing device via its serial port, and when it receives INSTEON-formatted messages from the computing device via the serial port, it sends them out over the INSTEON network.
- CoreApp handles linking to other INSTEON devices and maintains a Link Database.
- CoreApp is event-driven, meaning that it can send messages to the computing device based on the time of day or other occurrences.
- CoreApp can send and receive X10 commands.

Source code for coreApp is available to developers to modify for their own purposes. Once programmed with an appropriately modified SALad App, the PLC can operate on its own without being connected to a computing device.

As described in the section *Masking Non-linked Network Traffic*, the PLC hides the full addresses contained within INSTEON messages that it sees, unless the messages are from devices that it is already linked to. In particular, SALad Apps that the PLC may be running cannot discover the addresses of previously unknown INSTEON devices, so a hacker cannot write a SALad App that violates INSTEON security protocols.

Manager Applications

An INSTEON Manager App is an External Application program that runs on a computing device, like a PC or PDA, connected to an INSTEON network via an INSTEON Bridge. Manager Apps can provide sophisticated user interfaces for INSTEON devices, they can interact in complex ways with the outside world, and they can orchestrate system behaviors that bring real lifestyle benefits to people.

A Manager App exchanges INSTEON messages directly with INSTEON devices, so it must contain a software module that can translate between a user's intentions and the rules for composing and parsing INSTEON messages.

An example of a Manager App that encapsulates these functions is Smarthome's *Device Manager* (SDM) (see *Smarthome Device Manager Reference*), a Windows program that connects to an INSTEON network via a PowerLinc™ Controller (PLC). SDM handles all the intricacies involved with sending and receiving INSTEON messages via a PLC. To the outside world, it exposes an interface that developers can connect their own custom top-level application layers to.

This topmost layer, often a user interface, communicates with SDM using the Internet HTTP protocol or Microsoft's ActiveX, so it can run on an Internet browser or within a Windows program. SDM and the top layer communicate using a simple text-based scripting language developed by Smarthome called *Home Network Language™* (HNL).

SDM allows designers to concentrate on rapid application development of their end products without having to deal directly with INSTEON messaging issues. Product developers are encouraged to contact Smarthome at info@insteon.net for more information about acquiring and using SDM.

SALad Applications

SALad is a language interpreter embedded in the firmware of SALad-enabled INSTEON devices (see *SALad Overview*). By writing and debugging SALad programs in Smarthome's *SALad Integrated Development Environment*, developers can create INSTEON Internal Applications that run directly on SALad-enabled devices.

SALad Overview

Because the SALad instruction set is small and addressing modes for the instructions are highly symmetrical, SALad programs run fast and SALad object code is very compact.

SALad is event driven. Events are triggered when a device receives an INSTEON message, a user pushes a button, a timer expires, an X10 command is received, and so forth. As events occur, firmware in a SALad-enabled device posts event handles to an event queue, and then starts the SALad program. The SALad program determines what action to take based on the event that started it.

SALad programs can be downloaded into nonvolatile memory of INSTEON devices using the INSTEON network itself, or via a serial link if the device has one. SALad also contains a small debugger that allows programs to be started, stopped, and single-stepped directly over the INSTEON network.

SALad programming mostly consists of writing event handlers. By following examples in the INSTEON Software Development Kit, or by modifying Smarthome's *SALad coreApp Program*, developers can rapidly create INSTEON devices with wide-ranging capabilities. For more information about the SALad Language, consult the *SALad Language Documentation* in this Developer's Guide, or contact Smarthome at info@insteon.net.

SALad Integrated Development Environment

The SALad Integrated Development Environment (IDE) is a comprehensive, user-friendly tool for creating and debugging Internal Applications that run directly on SALad-enabled INSTEON devices. Using this tool, programmers can write, compile, download, and debug SALad programs without ever having to leave the IDE. The IDE is a Windows program that connects to an INSTEON network using a Smarthome PowerLinc™ Controller (see *The Smarthome PowerLinc Controller*).

The SALad IDE includes:

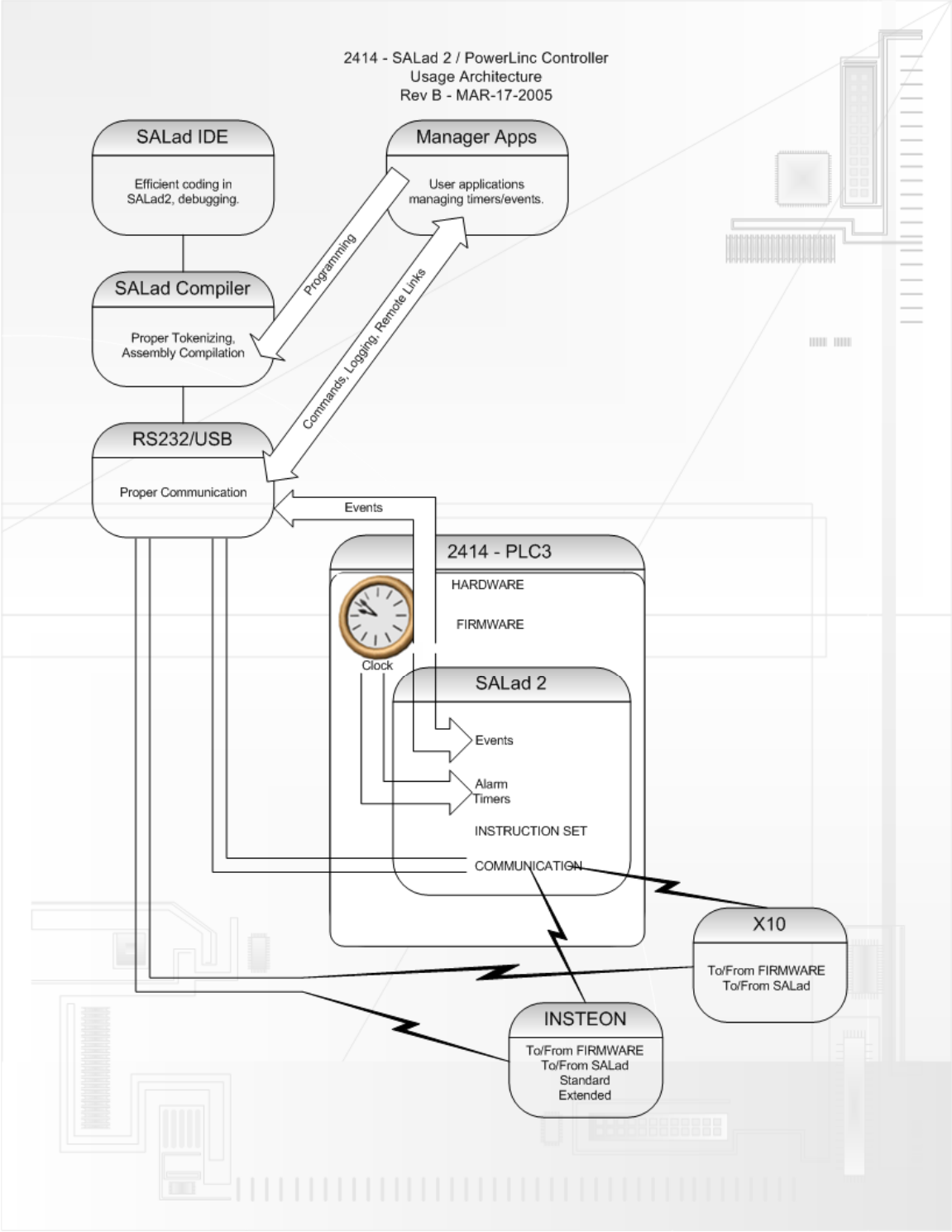
- A SALad Compiler that reads SALad language files and writes SALad object code, error listings, and variable maps
- A communications module that can download SALad object code to an INSTEON device via USB, RS232, or the INSTEON network itself
- A multiple-file, color-contextual source code editor that automatically compiles SALad programs on the fly
- Code templates for common tasks
- A real-time debugger based upon instantaneous feedback from a SALad-enabled device
- A program tracer

- An interactive device conversation window for sending and receiving INSTEON, X10, or ASCII messages
- A raw data window
- A PLC simulator for writing and debugging SALad Apps without actually being connected to an INSTEON network
- INSTEON device diagnostics
- INSTEON network diagnostics
- A device Link Database manager
- A program listing formatter

For complete information on installing and using the SALad IDE, consult the *SALad Integrated Development Environment User's Guide* below.

INSTEON SALad and PowerLinc Controller Architecture

This diagram shows how software solutions can be implemented using *The Smarthome PowerLinc Controller* as an INSTEON gateway/controller device.



INSTEON Developer's Kits

Smarthome is committed to making the development process as easy as possible for those who create products that can profit from INSTEON networking. For designers who will be crafting new INSTEON devices, adding INSTEON networking to existing devices, or developing External Applications for a network of INSTEON devices, Smarthome offers both a Software Developer's Kit (SDK) and a series of Hardware Development Modules, as well as extensive technical support.

Software Developer's Kit

To encourage as many developers as possible to join the community of INSTEON product creators, Smarthome offers a comprehensive Software Developer's Kit (SDK) for $99. The INSTEON SDK includes:

- The INSTEON Integrated Development Environment (IDE)
- A Smarthome PowerLinc™ V2 Controller (PLC) with either a USB or RS232 serial interface
- A Smarthome LampLinc™ V2 Dimmer module
- This INSTEON Developer's Guide in both text and compiled help formats
- Access to technical support and peer networking on the INSTEON Internet Forum
- Source code to the *SALad coreApp Program* that runs on the PLC
- Sample SALad Applications
- Version maps for product upgrades
- Header files

Hardware Development Modules

Smarthome will be releasing a series of Hardware Development Modules. The module that is currently available is an isolated powerline module ($99), to be followed soon by a non-isolated powerline module and an RF development module.

The isolated powerline module is essentially a PowerLinc™ Controller (PLC) with an extender board that has a prototyping area and a hardware interface to internal circuitry, including the microcontroller. With this module, designers can build and debug hardware interfaces to controllers, sensors, or actuators that connect to an INSTEON network. The isolated power supply for this module ensures that no dangerous voltages are exposed.

The non-isolated version of the powerline development module is only intended for those experts who are developing products that must achieve the lowest possible cost while still communicating over the powerline. To reduce the part count, the power supply is directly connected to the 110-volt mains, so potentially lethal voltages are exposed. Use of this module requires signing a liability waiver.

The RF development module contains a special RF daughter board extending from a PLC. With this module, developers can create products that communicate via RF, and only optionally communicate via the powerline. RF-only devices can be battery operated, so this module is particularly designed with developers of handheld INSTEON devices in mind.

INSTEON REFERENCE

In This Section

INSTEON Messages

INSTEON devices communicate by sending messages to one another. In the interest of maximum simplicity, there are only two kinds of INSTEON messages: 10-byte Standard messages and 24-byte Extended messages. The only difference between the two is that Extended messages carry 14 bytes of arbitrary User Data. They both carry a From Address, a To Address, a Flag byte, two Command bytes, and a Message Integrity byte.

In this section

INSTEON Message Structure

INSTEON devices communicate with each other by sending fixed-length messages. This section describes the two *Message Lengths* (Standard and Extended) and explains the contents of the *Message Fields* within the messages.

Message Lengths

There are only two kinds of INSTEON messages, 10-byte Standard Length Messages and 24-byte Extended Length Messages.

The only difference between the two is that the Extended message contains 14 User Data Bytes not found in the Standard message. The remaining information fields for both types of message are identical.

INSTEON Standard Message – 10 Bytes				
3 Bytes	3 Bytes	1 Byte	2 Bytes	1 Byte
From Address	To Address	Flags	Command 1, 2	CRC[3]

INSTEON Extended Message – 24 Bytes					
3 Bytes	3 Bytes	1 Byte	2 Bytes	14 Bytes	1 Byte
From Address	To Address	Flags	Command 1, 2	User Data	CRC[3]

Standard Message

Standard messages are designed for direct command and control. The payload is just two bytes, Command 1 and Command 2.

Data		**Bits**	**Contents**
From Address		24	Message Originator's address
To Address		24	For Direct messages: Intended Recipient's address For Broadcast messages: Device Type, Subtype, Firmware Version For Group Broadcast messages: Group Number [0 - 255]
Message Flags	Message Type	1	Broadcast/NAK
		1	Group
		1	Acknowledge
	Extended Flag	1	**0 (Zero) for Standard messages**
	Hops Left	2	Counted down on each retransmission
	Max Hops	2	Maximum number of retransmissions allowed
Command 1		8	Command to execute
Command 2		8	
CRC[3]		8	Cyclic Redundancy Check

Extended Message

In addition to the same fields found in Standard messages, Extended messages carry 14 bytes of arbitrary User Data for downloads, uploads, encryption, and advanced applications.

Data		Bits	Contents
From Address		24	Message Originator's address
To Address		24	For Direct messages: Intended Recipient's address For Broadcast messages: Device Type, Subtype, Firmware Version For Group Broadcast messages: Group Number [0 - 255]
Message Flags	Message Type	1	Broadcast/NAK
		1	Group
		1	Acknowledge
	Extended Flag	1	**1 (One) for Extended messages**
	Hops Left	2	Counted down on each retransmission
	Max Hops	2	Maximum number of retransmissions allowed
Command 1		8	Command to execute
Command 2		8	
User Data 1		8	User defined data
User Data 2		8	
User Data 3		8	
User Data 4		8	
User Data 5		8	
User Data 6		8	
User Data 7		8	
User Data 8		8	
User Data 9		8	
User Data 10		8	
User Data 11		8	
User Data 12		8	
User Data 13		8	
User Data 14		8	
CRC[3]		8	Cyclic Redundancy Check

Message Fields

All INSTEON messages contain source and destination *Device Addresses*, a *Message Flags* byte, a 2-byte *Command 1 and 2* payload, and a *Message Integrity Byte*. INSTEON Extended messages also carry 14 bytes of *User Data*.

Device Addresses

The first field in an INSTEON message is the *From Address*, a 24-bit (3-byte) number that uniquely identifies the INSTEON device originating the message being sent. There are 16,777,216 possible INSTEON devices identifiable by a 3-byte number. This number can be thought of as an ID Code or, equivalently, as an address for an INSTEON device. During manufacture, a unique ID Code is stored in each device in nonvolatile memory.

The second field in an INSTEON message is the *To Address*, also a 24-bit (3-byte) number. Most INSTEON messages are of the *Direct*, or *Point-to-Point* (P2P), type, where the intended recipient is another single, unique INSTEON device.

If the message is indeed Direct (as determined by the Flags Byte), the To Address contains the 3-byte unique ID Code for the intended recipient. However, INSTEON messages can also be sent to all recipients within range, as *Broadcast* messages, or they can be sent to all members of a group of devices, as *Group Broadcast* messages. In the case of Broadcast messages, the To Address field contains a 2-byte *Device Type* and a *Firmware Version* byte. For Group Broadcast messages, the To Address field contains a Group Number. Group Numbers only range from 0 to 255, given by one byte, so the two most-significant bytes of the three-byte field will be zero.

Message Flags

The third field in an INSTEON message, the Message Flags byte, not only signifies the Message Type but it also contains other information about the message. The three most-significant bits, the Broadcast/NAK flag (bit 7), the Group flag (bit 6), and the ACK flag (bit 5) together indicate the Message Type. Message Types will be explained in more detail in the next section (see *Message Type Flags*). Bit 4, the Extended flag, is set to one if the message is an Extended message, i.e. contains 14 User Data bytes, or else it is set to zero if the message is a Standard message. The low nibble contains two two-bit fields, Hops Left (bits 3 and 2) and Max Hops (bits 1 and 0). These two fields control message retransmission as explained below (see *Message Retransmission Flags*).

The table below enumerates the meaning of the bit fields in the Message Flags byte. The Broadcast/NAK flag (bit 7, the most-significant byte), the Group flag (bit 6), and the ACK flag (bit 5) together denote the eight possible Message Types.

Bit Position	Flag	Meaning
Bit 7 (Broadcast /NAK) (MSB)	Message Type	100 = Broadcast Message 000 = Direct Message 001 = ACK of Direct Message 101 = NAK of Direct Message 110 = Group Broadcast Message 010 = Group Cleanup Direct Message 011 = ACK of Group Cleanup Direct Message 111 = NAK of Group Cleanup Direct Message
Bit 6 (Group)		
Bit 5 (Acknowledge)		
Bit 4	Extended	1 = Extended Message 0 = Standard Message
Bit 3	Hops Left	00 = 0 message retransmissions remaining 01 = 1 message retransmission remaining 10 = 2 message retransmissions remaining 11 = 3 message retransmissions remaining
Bit 2		
Bit 1	Max Hops	00 = Do not retransmit this message 01 = Retransmit this message 1 time maximum 10 = Retransmit this message 2 times maximum 11 = Retransmit this message 3 times maximum
Bit 0 (LSB)		

Message Type Flags

There are eight possible INSTEON Message Types given by the three Message Type Flag Bits.

Message Types

To fully understand the eight Message Types, consider that there are four basic classes of INSTEON messages: *Broadcast*, *Group Broadcast*, *Direct*, and *Acknowledge*.

Broadcast messages contain general information with no specific destination. Directed to the community of all devices within range, they are used extensively during device linking (see *Device Identification Broadcast*, below). Broadcast messages are not acknowledged.

Group Broadcast messages are directed to a group of devices that have previously been linked to the message originator (see *INSTEON Groups*, below). Group Broadcast messages are not acknowledged directly. They only exist as a means for speeding up the response to a command intended for multiple devices. After sending a Group Broadcast message to a group of devices, the message originator then sends a Direct 'Group Cleanup' message to each member of the group individually, and waits for an acknowledgement back from each device.

Direct messages, also referred to as Point-to-Point (P2P) messages, are intended for a single specific recipient. The recipient responds to Direct messages by returning an Acknowledge message.

Acknowledge messages (ACK or NAK) are messages from the recipient to the message originator in response to a Direct message. There is no acknowledgement to a Broadcast or Group Broadcast message. An ACK or NAK message may contain status information from the acknowledging device.

Message Type Flag Bits

The Broadcast/NAK flag will be set whenever the message is a Broadcast message or a Group Broadcast message. In those two cases the Acknowledge flag will be clear. If the Acknowledge flag is set, the message is an Acknowledge message. In that case the Broadcast/NAK flag will be set when the Acknowledge message is a NAK, and it will be clear when the Acknowledge message is an ACK.

The Group flag will be set to indicate that the message is a Group Broadcast message or part of a Group Cleanup conversation. This flag will be clear for general Broadcast messages and Direct conversations.

Now all eight Message Types can be enumerated as follows, where the three-bit field is given in the order Bit 7, Bit 6, Bit 5.

- Broadcast messages are Message Type 100.
- Direct (P2P) messages are 000.
- An ACK of a Direct message is 001
- A NAK of a Direct message is 101
- A Group Broadcast message is 110.
- Group Broadcasts are followed up by a series of Group Cleanup Direct messages of Message Type 010 to each member of the group.
- Each recipient of a Group Cleanup Direct message will return an acknowledgement with a Group Cleanup ACK of Message Type 011 or a Group Cleanup NAK of Message Type 111.

Extended Message Flag

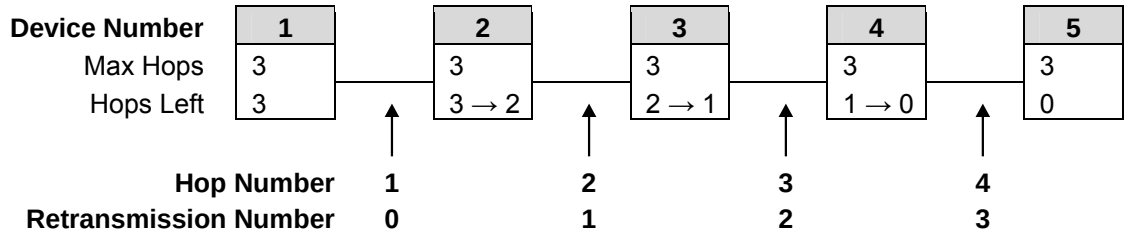
Bit 4 is the Extended Message flag. This flag is set for 24-byte Extended messages that contain a 14-byte User Data field, and the flag is clear for 10-byte Standard messages that do not contain User Data.

Message Retransmission Flags

The remaining two flag fields, Max Hops and Hops Left, manage message retransmission. As described above, all INSTEON devices are capable of repeating[1] messages by receiving and retransmitting them. Without a mechanism for limiting the number of times a message can be retransmitted, an uncontrolled 'data storm' of endlessly repeated messages could saturate the network. To solve this problem, INSTEON message originators set the 2-bit Max Hops field to a value of 0, 1, 2, or 3, and they also set the 2-bit Hops Left field to the same value. A Max Hops value of zero tells other devices within range not to retransmit the message. A higher Max Hops value tells devices receiving the message to retransmit it depending on the Hops Left field. If the Hops Left value is one or more, the receiving device decrements the Hops Left value by one, then retransmits the message with the new Hops Left value. Devices that receive a message with a Hops Left value of zero will not retransmit that message. Also, a device that is the intended recipient of a message will not retransmit the message, no matter what the Hops Left value is. See *INSTEON Message Hopping* for more information.

Note that the designator 'Max Hops' really means maximum *retransmissions* allowed. All INSTEON messages 'hop' at least once, so the value in the Max Hops field is one less than the number of times a message actually hops from one device to another. Since the maximum value in this field is three, there can be four actual hops,

consisting of the original transmission and three retransmissions. Four hops can span a chain of five devices. This situation is shown schematically below.



Command 1 and 2

The fourth field in an INSTEON message is a two-byte Command, made up of *Command 1* and *Command 2*. The usage of this field depends on the Message Type as explained below (see *INSTEON Commands*).

User Data

Only if the message is an Extended message, with the Extended Flag set to one, will it contain the fourteen-byte *User Data* field. User Data can be arbitrarily defined.

If more than 14 bytes of User Data need to be transmitted, multiple INSTEON Extended messages will have to be sent. Users can define a packetizing method for their data so that a receiving device can reliably reassemble long messages. Encrypting User Data can provide private, secure communications for sensitive applications such as security systems. See *INSTEON Extended Messages*, below, for more information.

Message Integrity Byte

The last field in an INSTEON message is a one-byte CRC, or Cyclic Redundancy Check. The INSTEON transmitting device computes the CRC over all the bytes in a message beginning with the From Address. INSTEON uses a software-implemented 7-bit linear-feedback shift register with taps at the two most-significant bits. The CRC covers 9 bytes for Standard messages and 23 bytes for Extended messages. An INSTEON receiving device computes its own CRC over the same message bytes as it receives them. If the message is corrupt, the receiver's CRC will not match the transmitted CRC.

Firmware in the INSTEON Engine handles the CRC byte automatically, appending it to messages that it sends, and comparing it within messages that it receives. Applications post messages to and receive messages from the INSTEON Engine without the CRC byte being appended.

Detection of message integrity allows for highly reliable, verified communications. The INSTEON ACK/NAK (acknowledge, non-acknowledge) closed-loop messaging protocol based on this detection method is described below (see *INSTEON Message Retrying*).

INSTEON Message Summary Table

The table below summarizes all the fields in every type of INSTEON message. Standard messages are enumerated at the top and Extended messages are enumerated at the bottom. The figure clearly shows that the only difference between Standard and Extended messages is that the Extended Flag is clear for Standard messages and set for Extended messages, and Extended messages possess a 14-byte User Data field. The From Address, the To Address, the Message Flags, and the CRCs are as explained above.

Message		3 Bytes			3 Bytes			1 Byte						1 Byte	1 Byte	1 Byte
		From Address			To Address			Message Flags						Cmd 1	Cmd 2	CRC[3]
								Type			X	HL	MH			
Standard	Broadcast	ID1_2	ID1_1	ID1_0	Device Type		Revision	1	0	0	0	Message retransmissions left	Maximum message retransmissions allowed	Broadcast Cmd		CRC
	Broadcast Group	ID1_2	ID1_1	ID1_0	0	0	Group #	1	1	0	0			Gp Cmd	Param	CRC
	P2P Cleanup	ID1_2	ID1_1	ID1_0	ID2_2	ID2_1	ID2_0	0	1	0	0			Gp Cmd	Group #	CRC
	P2P Cleanup ACK	ID1_2	ID1_1	ID1_0	ID2_2	ID2_1	ID2_0	0	1	1	0			Gp Cmd	Status	CRC
	P2P Cleanup NAK	ID1_2	ID1_1	ID1_0	ID2_2	ID2_1	ID2_0	1	1	1	0			Gp Cmd	Reason	CRC
	P2P	ID1_2	ID1_1	ID1_0	ID2_2	ID2_1	ID2_0	0	0	0	0			Direct Cmd		CRC
	P2P ACK	ID1_2	ID1_1	ID1_0	ID2_2	ID2_1	ID2_0	0	0	1	0			ACK Status		CRC
	P2P NAK	ID1_2	ID1_1	ID1_0	ID2_2	ID2_1	ID2_0	1	0	1	0			NAK Reason		CRC
															14 Bytes	1 Byte
															D1 ⇒ D14	CRC[3]
Extended	Broadcast	ID1_2	ID1_1	ID1_0	Device Type		Revision	1	0	0	1			Broadcast Cmd		D1 ⇒ D14	CRC
	Broadcast Group	ID1_2	ID1_1	ID1_0	0	0	Group #	1	1	0	1			Gp Cmd	Param	D1 ⇒ D14	CRC
	P2P Cleanup	ID1_2	ID1_1	ID1_0	ID2_2	ID2_1	ID2_0	0	1	0	1			Gp Cmd	Group #	D1 ⇒ D14	CRC
	P2P Cleanup ACK	ID1_2	ID1_1	ID1_0	ID2_2	ID2_1	ID2_0	0	1	1	1			Gp Cmd	Status	D1 ⇒ D14	CRC
	P2P Cleanup NAK	ID1_2	ID1_1	ID1_0	ID2_2	ID2_1	ID2_0	1	1	1	1			Gp Cmd	Reason	D1 ⇒ D14	CRC
	P2P	ID1_2	ID1_1	ID1_0	ID2_2	ID2_1	ID2_0	0	0	0	1			Direct Cmd		D1 ⇒ D14	CRC
	P2P ACK	ID1_2	ID1_1	ID1_0	ID2_2	ID2_1	ID2_0	0	0	1	1			ACK Status		D1 ⇒ D14	CRC
	P2P NAK	ID1_2	ID1_1	ID1_0	ID2_2	ID2_1	ID2_0	1	0	1	1			NAK Reason		D1 ⇒ D14	CRC
								Broadcast / NAK	Group	Acknowledge	Extended	Hops Left, 2 bits	Max Hops, 2 bits				

The Command 1 and Command 2 fields contain different information for each of the eight types of INSTEON messages. In the case of Broadcast messages, the two fields together contain a 2-byte command chosen from a possible 65,536 commands suitable for sending to all devices at once. For example, a device can identify itself to other devices by sending a *SET Button Pushed* Broadcast command (see *Device Identification Broadcast*). Every receiving device contains a database of Broadcast commands that it is capable of executing.

In the case of Point-to-Point (Direct) messages, the two Command fields together comprise a 2-byte command chosen from a possible 65,536 commands suitable for sending to a single device. For example, a Direct command could tell a lamp control device to turn on the lamp plugged into it. Every receiving device contains a database of Direct commands that it is capable of executing (see *INSTEON Commands*).

In the interest of maximum system reliability, the INSTEON protocol requires that Direct messages be acknowledged. A receiving device can issue an acknowledgement of successful communication and completion of a task, i.e. an ACK, or it can issue a NAK to indicate some kind of failure. If a receiving device fails to send an ACK or a NAK back to the originating device, the originating device will retry the message (see *INSTEON Message Retrying*).

To respond with an ACK or a NAK, firmware in a receiving device swaps the From Address and the To Address in the message it received, and sets the Message Type bits to 001 for an ACK or 101 for a NAK. Depending on the command received in the Command fields, the receiving device composes a two-byte status response code for an ACK or else a two-byte reason code for a NAK, which it inserts in the Command fields. For example, if a lamp dimmer receives a command to set the lamp to a certain brightness level, issued as a Set Brightness code in the Command 1 field and the desired brightness level as one of 256 values in the Command 2 field, the dimmer will respond with an ACK message containing the same two bytes in the Command fields to indicate successful execution of the command.

The remaining INSTEON message types are for dealing with groups of devices (see *INSTEON Groups*). Group Broadcast messages exist as a performance enhancement. While it is true that all the members of a group of devices could be sent individual Direct messages with the same command (to turn on, for example), it would take a noticeable amount of time for all the messages to be transmitted in sequence. The members of the group would not execute the command all at once, but rather in the order received. INSTEON solves this problem by first sending a Group Broadcast message, then following it up with individual Direct 'Group Cleanup' messages.

Group Broadcast messages contain a Group Number in the To Address field, a one-byte Group Command in the Command 1 field, and an optional one-byte parameter in the Command 2 field. During the Direct Group Cleanup messages that will follow, the Group Command will be sent in the Command 1 field and the Group Number will be sent in the Command 2 field. These are both one-byte fields, so there can only be 256 Group Commands and only 256 Group Numbers. This is a reasonable limitation given that Group Broadcasts only need to be used where rapid, synchronous response of multiple devices is an issue. In any case, the numerical limitation can be overcome by using Extended messages and embedding additional commands or group membership criteria in the User Data field.

Recipients of a Group Broadcast message check the Group Number in the To Address field against their own group memberships recorded in a Link Database (see *INSTEON Link Database*). This database, stored in nonvolatile memory, is established during a prior group enrollment, or *linking*, process (see *Methods for Linking INSTEON Devices*). If the recipient is a member of the Group being broadcast to, it executes the command in the Command 1 field. Since the Group Command only occupies one byte, the other byte in field can be a parameter or a subcommand.

Group Broadcast command recipients can expect a Direct individually-addressed Group Cleanup message to follow. If the recipient has already executed the Group Command, it will not execute the command a second time. However, if the recipient missed the Group Broadcast command for any reason, it will not have executed it, so it will execute the command after receiving the Direct Group Cleanup message.

After receiving the Direct Group Cleanup message and executing the Group Command, the recipient device will respond with a Group Cleanup ACK message, or else, if something went wrong, it will respond with a Group Cleanup NAK message. In both cases the Command 1 field will contain the same one-byte Group Command received during the Direct Group Cleanup message. The other byte in the Command 2 field will contain a one-byte ACK Status code in the case of an ACK, or a one-byte NAK Reason code in the case of a NAK. These one-byte codes are a subset of the corresponding two-byte codes used in Direct ACK and Direct NAK messages.

INSTEON Message Repetition

To maximize communications reliability, the INSTEON messaging protocol includes two kinds of message repetition: message hopping and message retrying.

INSTEON Message Hopping is the mechanism whereby INSTEON devices, all of which can retransmit INSTEON messages, aid each other in delivering a message from a message originator to a message recipient.

INSTEON Message Retrying occurs when the originator of a Direct message does not receive a proper acknowledgement message from the intended recipient.

INSTEON Message Hopping

In order to improve reliability, the INSTEON messaging protocol includes message retransmission, or hopping. Hopping enables other INSTEON devices, all of which can repeat[1] messages, to help relay a message from an originator to a recipient.

When INSTEON devices repeat messages, multiple devices can end up simulcasting the same message, meaning that they can repeat the same message at the same time. To ensure that simulcasting is synchronous (so that multiple devices do not jam each other), INSTEON devices adhere to specific rules given below (see *Timeslot Synchronization*).

Message Hopping Control

Two 2-bit fields in the Message Flags byte manage INSTEON message hopping (see *Message Retransmission Flags*, above). One field, *Max Hops*, contains the maximum number of hops, and the other, *Hops Left,* contains the number of hops remaining.

To avoid 'data storms' of endless repetition, messages can be retransmitted a maximum of three times only. A message originator sets the Max Hops for a message. The larger the number of Max Hops, the longer the message will take to complete being sent, whether or not the recipient hears the message early.

If the Hops Left field in a message is nonzero, every device that hears the message synchronously repeats it, thus increasing the signal strength, path diversity, and range of the message. An INSTEON device that repeats a message decrements Hops Left before retransmitting it. When a device receives a message with zero Hops Left, it does not retransmit the message.

Timeslot Synchronization

There is a specific pattern of transmissions, retransmissions and acknowledgements that occurs when an INSTEON message is sent, as shown in the examples below.

An INSTEON message on the powerline occupies either six or thirteen zero crossing periods, depending on whether the message is Standard or Extended. This message transmission time, six or thirteen powerline half-cycles, is called a *timeslot* in the following discussion.

During a single timeslot, an INSTEON message can be transmitted, retransmitted, or acknowledged. The entire process of communicating an INSTEON message, which may involve retransmissions and acknowledgements, will occur over integer multiples of timeslots.

The following examples show how INSTEON messages propagate in a number of common scenarios. The examples use these symbols:

Legend		
	T	Transmission by Message Originator
	R	Message Retransmission
	A	Acknowledgement by Intended Recipient
	C	Confirmation received by Message Originator
	L	Listening State
	W	Waiting State

	Max Hops	Timeslot	1	2	3	4	5	6	7	8
Example 1	**0**	Sender	**T**							

Example 1, the simplest, shows a Broadcast message with a Max Hops of zero (no retransmissions). The **T** indicates that the Sender has originated and transmitted a single message. There is no acknowledgement that intended recipients have heard the message. The message required one timeslot of six or thirteen powerline zero crossings to complete.

	Max Hops	Timeslot	1	2	3	4	5	6	7	8
Example 2	**1**	Sender	**T**							
		Repeater 1	L	**R**						

Example 2 shows a Broadcast message with a Max Hops of one. Max Hops can range from zero to three as explained above. The Sender transmits a Broadcast message as signified by the **T**. Another INSTEON device, functioning as a Repeater, listens to the message, as signified by an **L**, and then retransmits it in the next timeslot as indicated by the **R**.

	Max Hops	Timeslot	1	2	3	4	5	6	7	8
Example 3	**3**	Sender	**T**	L	L	L	L			
		Repeater 1	L	**R**	L	**R**	L			
		Repeater 2	L	L	**R**	L	L			
		Repeater 3	L	L	L	**R**	L			

Up to three retransmissions are possible with a message. **Example 3** shows the progression of the message involving an originating Sender and three repeating devices, with a Max Hops of three. Example 3 assumes that the range between Repeaters is such that only adjacent Repeaters can hear each other, and that only Repeater 1 can hear the Sender. Note that the Sender will not retransmit its own message.

	Max Hops	Timeslot	1	2	3	4	5	6	7	8
Example 4	**0**	Sender	**T**	**C**						
		Recipient	L	**A**						

When a Sender transmits a Direct (Point-to-Point) message, it expects an acknowledgement from the Recipient. **Example 4** shows what happens if the Max Hops value is zero. The **A** designates the timeslot in which the Recipient acknowledges receipt of the Direct message. The **C** shows the timeslot when the Sender finds that the message is confirmed.

	Max Hops	Timeslot	1	2	3	4	5	6	7	8
Example 5	**1**	Sender	**T**	L	L	**C**				
		Repeater 1	L	**R**	L	**R**				
		Recipient	L	L	**A**	L				

When Max Hops is set to one, a Direct message propagates as shown in **Example 5**. Repeater 1 will retransmit both the original Direct message and the acknowledgement from the Recipient.

	Max Hops	Timeslot	1	2	3	4	5	6	7	8
Example 6	**1**	Sender	**T**	L	**C**	W				
		Repeater 1	L	**R**	L	**R**				
		Recipient	L	W	**A**	L				

If Max Hops is set to one, but no retransmission is needed because the Recipient is within range of the Sender, messages flow as shown in **Example 6**. The **W** in the Sender and Recipient rows indicates a wait. The Recipient immediately hears the Sender since it is within range. However, the Recipient must wait one timeslot before sending its acknowledgement, because it is possible that a repeating device will be retransmitting the Sender's message. Repeater 1 is shown doing just that in the example, although the Recipient would still have to wait even if no Repeaters were present. Only when all of the *possible* retransmissions of the Sender's message are complete, can the Recipient send its acknowledgement. Being within range, the Sender hears the acknowledgement immediately, but it must also wait until possible retransmissions of the acknowledgement are finished before it can send another message.

	Max Hops	Timeslot	1	2	3	4	5	6	7	8
Example 7	3	Sender	**T**	L	L	L	L	L	L	**C**
		Repeater 1	L	**R**	L	**R**	L	**R**	L	**R**
		Repeater 2	L	L	**R**	L	L	L	**R**	L
		Repeater 3	L	L	L	**R**	L	**R**	L	**R**
		Recipient	L	L	L	L	**A**	L	**R**	L

Example 7 shows what happens when Max Hops is three and three retransmissions are in fact needed for the message to reach the Recipient. Note that if the Sender or Recipient were to hear the other's message earlier than shown, it still must wait until Max Hops timeslots have occurred after the message was originated before being free to send its own message. If devices did not wait, they would jam each other by sending different messages in the same timeslot. A device can calculate how many timeslots have passed prior to receiving a message by subtracting the Hops Left number in the received message from the Max Hops number.

All seven of the above examples are given again in the table below in order to show the patterns more clearly.

	Max Hops	Timeslot	1	2	3	4	5	6	7	8
Example 1	**0**	Sender	**T**							
Example 2	**1**	Sender	**T**							
		Repeater 1	L	**R**						
Example 3	**3**	Sender	**T**	L	L	L	L			
		Repeater 1	L	**R**	L	**R**	L			
		Repeater 2	L	L	**R**	L	L			
		Repeater 3	L	L	L	**R**	L			
Example 4	**0**	Sender	**T**	**C**						
		Recipient	L	**A**						
Example 5	**1**	Sender	**T**	L	L	**C**				
		Repeater 1	L	**R**	L	**R**				
		Recipient	L	L	**A**	L				
Example 6	**1**	Sender	**T**	L	**C**	W				
		Repeater 1	L	**R**	L	**R**				
		Recipient	L	W	**A**	L				
Example 7	**3**	Sender	**T**	L	L	L	L	L	L	**C**
		Repeater 1	L	**R**	L	**R**	L	**R**	L	**R**
		Repeater 2	L	L	**R**	L	L	L	**R**	L
		Repeater 3	L	L	L	**R**	L	**R**	L	**R**
		Recipient	L	L	L	L	**A**	L	**R**	L

Legend		
	T	Transmission by Message Originator
	R	Message Retransmission
	A	Acknowledgement by Intended Recipient
	C	Confirmation received by Message Originator
	L	Listening State
	W	Waiting State

INSTEON Message Retrying

If the originator of an INSTEON Direct message does not receive an acknowledgement from the intended recipient, the message originator will automatically try resending the message up to five times.

In case a message did not get through because Max Hops was set too low, each time the message originator retries a message, it also increases Max Hops up to the limit of three. A larger number of Max Hops can achieve greater range for the message by allowing more devices to retransmit it.

Firmware in the INSTEON Engine handles message retrying. After using the INSTEON Engine to send a Direct message, applications will either receive the expected acknowledgement message or an indication that the intended recipient did not receive the Direct message after five retries.

Because message retrying is automatic, it is important to unlink INSTEON Responder devices from INSTEON Controller devices when a linked device is removed from an INSTEON network. See *INSTEON Link Database*, below, for more information.

INSTEON Signaling Details

This section gives complete information about how the data in INSTEON messages actually travels over the powerline or the airwaves. Unlike other mesh networks, INSTEON does not elaborately route its traffic in order to avoid data collisions—instead, INSTEON devices *simulcast* according to simple rules explained below. Simulcasting by multiple devices is made possible because INSTEON references a global clock, the powerline zero crossing.

In this section

INSTEON Packet Structure
: Shows how messages are packetized for powerline and RF transmission.

INSTEON Signaling
: Covers bit encoding for powerline and RF transmission, packet synchronizing, X10 compatibility[2], message timeslots, and data rates.

Simulcasting
: Explains how allowing multiple INSTEON devices to talk at the same time makes an INSTEON network more reliable as more devices are added, and eliminates the need for complex, costly message routing.

INSTEON Packet Structure

This section describes *Powerline Packets* and *RF Packets*.

Powerline Packets

Messages sent over the powerline are broken up into packets, with each packet sent in conjunction with a zero crossing of the AC voltage on the powerline. Standard Messages use five packets and Extended Messages use eleven packets, as shown below.

Standard Message – 5 Packets

120 total bits = 15 bytes
84 Data bits = 10½ bytes, 10 usable

SP	BP	BP	BP	BP

Extended Message - 11 Packets

264 total bits = 33 bytes
192 Data bits = 24 bytes

SP	BP	BP	BP	BP	BP	BP	BP	BP	BP	BP

A Start Packet appears as the first packet in an INSTEON message, as shown by the symbol **SP** in both the Standard and Extended Messages. The remaining packets in a message are Body Packets, as shown by the symbols **BP**.

Each packet contains 24 bits of information, but the information is interpreted in two different ways, as shown below.

SP Start Packet

1	0	1	0	1	0	1	0	1	0	0	1	x	x	x	x	x	x	x	x	x	x	x	x
8 Sync bits								4 Start Code bits				12 Data bits											

BP Body Packet

1	0	1	0	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
2 Sync bits		4 Start Code bits				18 Data Bits																	

Powerline packets begin with a series of *Sync Bits*. There are eight Sync Bits in a Start Packet and there are two Sync Bits in a Body Packet. The alternating pattern of ones and zeros allows the receiver to detect the presence of a signal.

Following the Sync Bits are four *Start Code Bits*. The 1001 pattern indicates to the receiver that Data bits will follow.

The remaining bits in a packet are *Data Bits*. There are twelve Data Bits in a Start Packet, and there are eighteen Data Bits in a Body Packet.

The total number of Data Bits in a Standard Message is 84, or 10½ bytes. The last four data bits in a Standard Message are ignored, so the usable data is 10 bytes. The total number of Data Bits in an Extended Message is 192, or 24 bytes.

RF Packets

The figure below shows the contents of INSTEON messages sent using RF. Because INSTEON RF messaging is much faster than powerline messaging, there is no need to break up RF messages into smaller packets. An RF Standard message and an RF Extended message are both shown. In both cases the message begins with two Sync Bytes followed by one Start Code Byte. RF Standard messages contain 10 Data Bytes (80 bits), and RF Extended messages contain 24 Data Bytes (192 bits).

RF Standard Message – 1 Packet

112 total bits = 14 bytes
80 Data bits = 10 bytes

AA	AA	C3	x	x	x	x	x	x	x	x	x	x	n
2 Sync bytes		1 Start Code byte	80 Data Bits (10 Data bytes)										CRC[3]

RF Extended Message – 1 Packet

224 total bits = 28 bytes
192 Data bits = 24 bytes

AA	AA	C3	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	n
2 Sync bytes		1 Start Code byte	192 Data Bits (24 Data bytes)																								CRC[3]

INSTEON Signaling

This section explains bit encoding for powerline and RF transmission, packet synchronization to the powerline, X10 compatibility[2], message timeslots, and data rates.

Powerline Signaling

INSTEON devices communicate on the powerline by adding a signal to the powerline voltage. In the United States, powerline voltage is nominally 110 VAC RMS, alternating at 60 Hz.

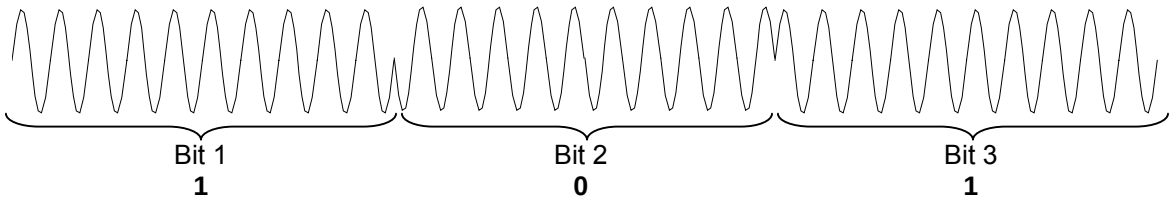
An INSTEON powerline signal uses a carrier frequency of 131.65 KHz, with a nominal amplitude of 4.64 volts peak-to-peak into a 5 ohm load. In practice, the impedance of powerlines varies widely, depending on the powerline configuration and what is plugged into it, so measured INSTEON powerline signals can vary from sub-millivolt to more than 5 volts.

INSTEON data is modulated onto the 131.65 KHz carrier using binary phase-shift keying, or BPSK, chosen for reliable performance in the presence of noise.

The bytes in an INSTEON powerline message are transmitted most-significant byte first, and the bits are transmitted most-significant bit first.

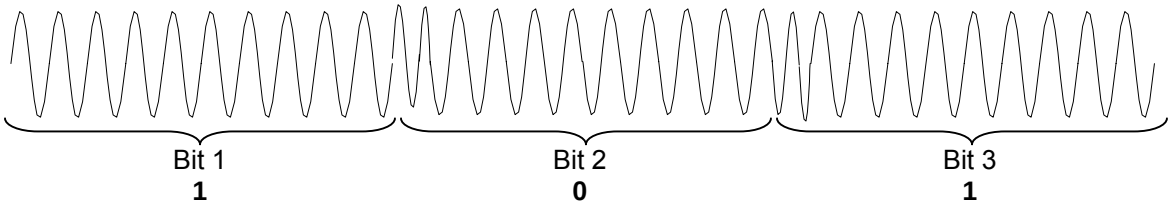
BPSK Modulation

The figure below shows an INSTEON 131.65 KHz powerline carrier signal with alternating binary phase-shift keying (BPSK) bit modulation.



INSTEON uses 10 cycles of carrier for each bit. Bit 1, interpreted as a one, begins with a positive-going carrier cycle. Bit 2, interpreted as a zero, begins with a negative-going carrier cycle. Bit 3 begins with a positive-going carrier cycle, so it is interpreted as a one. Note that the sense of the bit interpretations is arbitrary. That is, ones and zeros could be reversed as long as the interpretation is consistent. Phase transitions only occur when a bitstream changes from a zero to a one or from a one to a zero. A one followed by another one, or a zero followed by another zero, will not cause a phase transition. This type of coding is known as NRZ, or non-return to zero.

Note the abrupt phase transitions of 180 degrees at the bit boundaries. Abrupt phase transitions introduce troublesome high-frequency components into the signal's spectrum. Phase-locked detectors can have trouble tracking such a signal. To solve this problem, INSTEON uses a gradual phase change to reduce the unwanted frequency components.

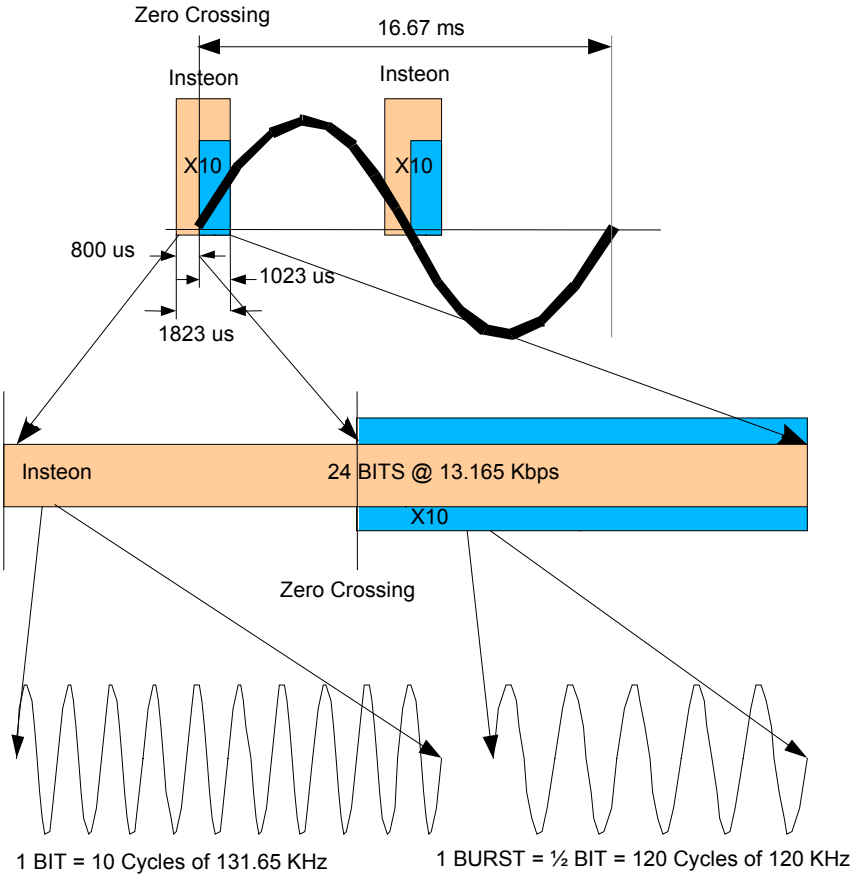


The figure above shows the same BPSK signal with gradual phase shifting. The transmitter introduces the phase change by inserting 1.5 cycles of carrier at 1.5 times the 131.65 KHz frequency. Thus, in the time taken by one cycle of 131.65 KHz, three half-cycles of carrier will have occurred, so the phase of the carrier will be reversed at the end of the period due to the odd number of half-cycles. Note the smooth transitions between the bits.

Packet Timing

All INSTEON powerline packets contain 24 bits. Since a bit takes 10 cycles of 131.65 KHz carrier, there are 240 cycles of carrier in an INSTEON packet. An INSTEON powerline packet therefore lasts 1.823 milliseconds.

The powerline environment is notorious for uncontrolled noise, especially high-amplitude spikes caused by motors, dimmers and compact fluorescent lighting. This noise is minimal during the time that the current on the powerline reverses direction, a time known as the powerline zero crossing. Therefore, INSTEON packets are transmitted during the zero crossing quiet time, as shown in the figure below.



The top of the figure shows a single powerline cycle, which possesses two zero crossings. An INSTEON packet is shown at each zero crossing. INSTEON packets begin 800 microseconds before a zero crossing and last until 1023 microseconds after the zero crossing.

X10 Compatibility

The figure also shows how X10 signals are applied to the powerline. X10 is the signaling method used by many devices already deployed on powerlines around the world. Compatibility[2] with this existing population of legacy X10 devices is an important feature of INSTEON. At a minimum, X10 compatibility means that INSTEON and X10 signals can coexist with each other, but compatibility also allows

designers to create hybrid devices that can send and receive both INSTEON and X10 signals.

The X10 signal uses a burst of approximately 120 cycles of 120 KHz carrier beginning at the powerline zero crossing and lasting about 1000 microseconds. A burst followed by no burst signifies an X10 one bit and no burst followed by a burst signifies an X10 zero bit. An X10 message begins with two bursts in a row followed by a one bit, followed by nine data bits. The figure shows an X10 burst at each of the two zero crossings.

The X10 specification also allows for two copies of the zero crossing burst located one-third and two-thirds of the way through a half-cycle of power. These points correspond to the zero crossings of the other two phases of three-phase power. INSTEON is insensitive to those additional X10 bursts and does not transmit them when sending X10.

The middle of the figure shows an expanded view of an INSTEON packet with an X10 burst superimposed. The X10 signal begins at the zero crossing, 800 microseconds after the beginning of the INSTEON packet. Both signals end at approximately the same time, 1023 microseconds after the zero crossing.

INSTEON devices achieve compatibility with X10 by listening for an INSTEON signal beginning 800 microseconds before the zero crossing. INSTEON receivers implemented in software can be very sensitive, but at the cost of having to receive a substantial portion of a packet before being able to validate that a true INSTEON packet is being received. Reliable validation may not occur until as much as 450 microseconds after the zero crossing, although an INSTEON device will still begin listening for a possible X10 burst right at the zero crossing. If at the 450-microsecond mark the INSTEON receiver validates that it is not receiving an INSTEON packet, but that there *is* an X10 burst present, the INSTEON receiver will switch to X10 mode and listen for a complete X10 message over the next 11 powerline cycles. If instead the INSTEON device detects that it is receiving an INSTEON packet, it will remain in INSTEON mode and not listen for X10 until it receives the rest of the complete INSTEON message.

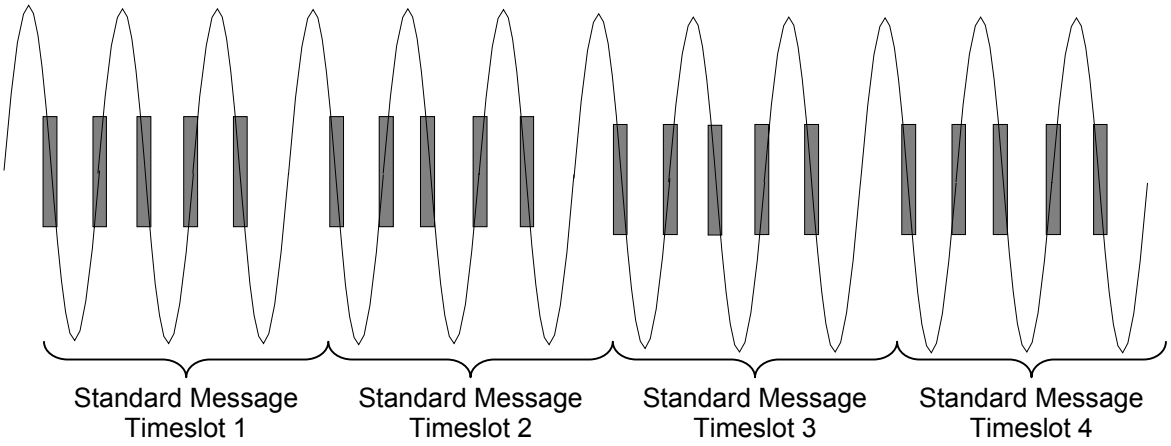
The bottom of the figure shows that the raw bitrate for INSTEON is much faster for INSTEON than for X10. An INSTEON bit requires ten cycles of 131.65 KHz carrier, or 75.96 microseconds, whereas an X10 bit requires *two* 120-cycle bursts of 120 KHz. One X10 burst takes 1000 microseconds, but since each X10 burst is sent at a zero crossing, it takes 16,667 microseconds to send the two bursts in a bit, resulting in a sustained bitrate of 60 bits per second. INSTEON packets consist of 24 bits, and an INSTEON packet can be sent during each zero crossing, so the nominal raw sustained bitrate for INSTEON is 2880 bits per second, 48 times faster than X10. Note that this nominal INSTEON bitrate must be derated to account for packet and message overhead, as well as message retransmissions. See *INSTEON Powerline Data Rates*, below, for details.

Message Timeslots

To allow time for potential retransmission of a message by INSTEON RF devices, an INSTEON transmitter waits for one additional zero crossing after sending a Standard message, or for two zero crossings after sending an Extended message. Therefore, the total number of zero crossings needed to send a Standard message is 6, or 13 for an Extended message. This number, 6 or 13, constitutes an INSTEON message timeslot.

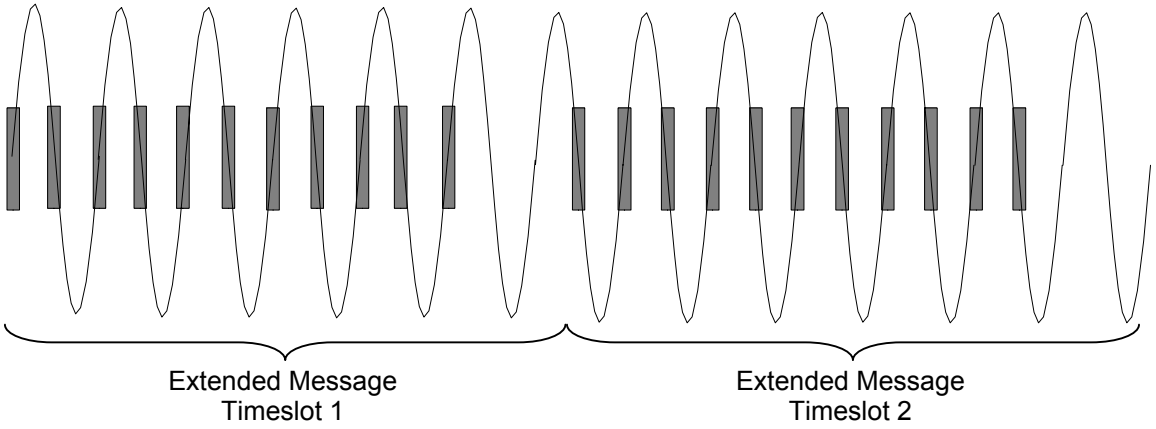
Standard Message Timeslots

The figure below shows a series of 5-packet Standard INSTEON messages being sent on the powerline. INSTEON transmitters wait for one zero crossing after each Standard message before sending another message, so the Standard message timeslot is 6 zero crossings, or 50 milliseconds, in length.



Extended Message Timeslots

The next figure shows a series of 11-packet Extended INSTEON messages being sent on the powerline. INSTEON transmitters wait for two zero crossings after each Extended message before sending another message, so the Extended message timeslot is 13 zero crossings, or 108.33 milliseconds, in length.



INSTEON Powerline Data Rates

INSTEON Standard messages contain 120 raw data bits and require 6 zero crossings, or 50 milliseconds to send. Extended messages contain 264 raw data bits and require 13 zero crossings, or 108.33 milliseconds to send. Therefore, the actual raw bitrate for INSTEON is 2400 bits per second for Standard messages, or 2437 bits per second for Extended messages, instead of the 2880 bits per second it would be without waiting for the extra zero crossings.

INSTEON Standard messages contain 9 bytes (72 bits) of usable data, not counting packet sync and start code bits, nor the message CRC byte. Extended messages contain 23 bytes (184 bits) of usable data using the same criteria. Therefore, the bitrates for usable data are further reduced to 1440 bits per second for Standard messages and 1698 bits per second for Extended messages. If one only counts the 14 bytes (112 bits) of User Data in Extended messages, the User Data bitrate is 1034 bits per second.

These data rates assume that messages are sent with Max Hops set to zero and that there are no message retries. They also do not take into account the time it takes for a message to be acknowledged. The table below shows net data rates when multiple hops and message acknowledgement are taken into account. To account for retries, divide the given data rates by one plus the number of retries (up to a maximum of 5 possible retries).

Condition			Bits per Second		
Max Hops	**ACK**	**Retries**	**Standard Message Usable Data**	**Extended Message Usable Data**	**Extended Message User Data Only**
0	No	0	1440	1698	1034
1	No	0	720	849	517
2	No	0	480	566	345
3	No	0	360	425	259
0	Yes	0	720	849	517
1	Yes	0	360	425	259
2	Yes	0	240	283	173
3	Yes	0	180	213	130

RF Signaling

RF INSTEON devices can send and receive the same messages that appear on the powerline. Unlike powerline messages, however, messages sent by RF are not broken up into smaller packets sent at powerline zero crossings, but instead are sent whole, as was shown in the section *RF Packets*. As with powerline, there are two RF message lengths: Standard 10-byte messages and Extended 24-byte messages.

The table below gives the specifications for INSTEON RF signaling.

RF Specification	Value
Center Frequency	904 MHz
Data Encoding Method	Manchester
Modulation Method	FSK
FSK Deviation	64 KHz
FSK Symbol Rate	76,800 symbols per second
Data Rate	38,400 bits per second
Range	150 feet outdoors

The center frequency lies in the band 902 to 924 MHz, which is permitted for unlicensed operation in the United States. Each bit is Manchester encoded, meaning that two symbols are sent for each bit. A one-symbol followed by a zero-symbol designates a one-bit, and a zero-symbol followed by a one-symbol designates a zero-bit. Symbols are modulated onto the carrier using frequency-shift keying (FSK), where a zero-symbol modulates the carrier half the FSK deviation frequency downward and a one-symbol modulates the carrier half the FSK deviation frequency upward. The FSK deviation frequency chosen for INSTEON is 64 KHz. Symbols are modulated onto the carrier at 76,800 symbols per second, resulting in a raw data rata of half that, or 38,400 bits per second. The typical range for free-space reception is 150 feet, which is reduced in the presence of walls and other RF energy absorbers.

INSTEON devices transmit data with the most-significant bit sent first. Referring to the figures below, RF messages begin with two sync bytes consisting of AAAA in hexadecimal, followed by a start code byte of C3 in hexadecimal. Ten data bytes follow in Standard messages, or twenty-four data bytes in Extended messages. The last data byte in a message is a CRC[3] over the data bytes as explained above (see *Message Integrity Byte*).

RF Standard Message – 1 Packet

112 total bits = 14 bytes
80 Data bits = 10 bytes

AA	AA	C3	x	x	x	x	x	x	x	x	x	x	n
2 Sync bytes		1 Start Code byte	80 Data Bits (10 Data bytes)										CRC[3]

RF Extended Message – 1 Packet

224 total bits = 28 bytes
192 Data bits = 24 bytes

AA	AA	C3	x	x	x	x	x	x	x	X	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	n
2 Sync bytes		1 Start Code byte	192 Data Bits (24 Data bytes)																								CRC[3]

It takes 2.708 milliseconds to send a 104-bit Standard message, and 5.625 milliseconds to send a 216-bit Extended message. Zero crossings on the powerline occur every 8.333 milliseconds, so a Standard or Extended RF message can be sent during one powerline half-cycle. The waiting times after sending powerline messages, as shown in the section *Powerline Packets*, are to allow sufficient time for INSTEON RF devices, if present, to retransmit a powerline message.

Simulcasting

By following the above rules for message propagation, INSTEON systems achieve a marked increase in the reliability of communications. The reason is that multiple INSTEON devices can transmit the same message at the same time within a given timeslot. INSTEON devices within range of each other thus "help each other out." Most networking protocols for shared physical media prohibit multiple devices from simultaneously transmitting within the same band by adopting complex routing algorithms. In contrast, INSTEON turns what is usually a problem into a benefit by ensuring that devices transmitting simultaneously will be sending the same messages in synchrony with each other.

Powerline Simulcasting

One might think that multiple INSTEON devices transmitting on the powerline could easily cancel each other out rather than boost each other. In practice, even if one were *trying* to nullify one signal with another, signal cancellation by multiple devices would be extremely difficult to arrange. The reason is that for two signals to cancel at a given receiver, the two transmitters would have to send carriers such that the receiver would see them as exactly equal in amplitude and very nearly 180 degrees out of phase. The probability of this situation occurring and persisting for extended periods is low.

The crystals used on typical INSTEON devices to generate the powerline carrier frequency of 131.65 KHz run independently of each other with a frequency tolerance of a few tenths of a percent. Phase relationships among multiple powerline carriers therefore will drift, although slowly with respect to the 1823 microsecond duration of an INSTEON packet. Even if the phases of two transmitters happened to cancel, it is very unlikely that the amplitudes would also be equal at the location of a receiver, so a receiver would very likely still see some signal even in the worst-case transient phase relationship. INSTEON receivers have a wide dynamic range, from millivolts to five volts or so, which will allow them to track signals even if they fade temporarily. Adding more transmitters reduces the probability of signal cancellation even more. Rather, the probability that the sum of all the signals will increase in signal strength becomes much greater with source diversity.

The INSTEON powerline carrier is modulated using binary phase-shift keying (BPSK), meaning that receivers are looking for 180-degree phase shifts in the carrier to detect changes in a string of bits from a one to a zero or vice-versa. Multiple transmitters, regardless of the absolute phase of their carriers, will produce signals whose sum still possesses 180-degree phase reversals at bit-change boundaries, so long as their relative carrier frequencies do not shift more than a few degrees over a packet time. Of course, bit timings for each transmitter need to be fairly well locked, so INSTEON transmitters are synchronized to powerline zero crossings. An INSTEON bit lasts for ten cycles of the 131.65 KHz powerline carrier, or 76 microseconds. The powerline zero crossing detector should be accurate within one or two carrier periods so that bits received from multiple transmitters will overlay each other.

In practice, multiple INSTEON powerline transmitters simulcasting the same message will improve the strength of the powerline signal throughout a building.

RF Simulcasting

Since RF signaling is used as an extension to powerline signaling, it also is based on simulcasting. However, because of the short wavelength of 900 MHz RF carrier signals, standing wave interference patterns can form where the RF carrier signal is reduced, even when the carrier and data are ideally synchronized.

As with powerline, for a cancellation to occur, two carriers must be 180 degrees out of phase and the amplitudes must be the same. Perfect cancellation is practically impossible to obtain. In general, two co-located carriers on the same frequency with random phase relationships and the same antenna polarization will sum to a power level greater than that of just one transmitter 67% of the time. As one of the transmitters is moved away from a receiver, the probability of cancellation drops further because the signal amplitudes will be unequal. As the number of transmitters increases, the probability of cancellation becomes nearly zero.

Mobile INSTEON RF devices, such as handheld controllers, are battery operated. To conserve power, mobile devices are not configured as RF Repeaters, but only as message originators, so RF simulcasting is not an issue for them. INSTEON devices that do repeat RF messages are attached to the powerline, so most of them will not be moved around after initial setup. During setup, such RF devices can be located, and their antennas adjusted, so that no signal cancellation occurs. With the location of the transmitters fixed, the non-canceling configuration will be maintained indefinitely.

RF/Powerline Synchronization

INSTEON RF devices attached to the powerline use the zero crossing for message synchronization. These devices receive INSTEON messages synchronously on the powerline, synchronously via RF from RF Repeaters, or possibly asynchronously via RF from mobile RF devices.

Messages that need to be retransmitted will have a Hops Left count greater than zero. If the INSTEON device receives such a message from the powerline, it will first retransmit the message using RF as soon as it has received the last packet of the powerline message, then it will retransmit the message on the powerline in the next timeslot. If the device receives the message via RF, it will first retransmit the message on the powerline in the next timeslot, then it will retransmit the message using RF immediately after sending the last packet of the powerline message. In this way, RF message received asynchronously will be resynchronized to the powerline zero crossing at the earliest opportunity.

INSTEON Network Usage

INSTEON messaging technology can be used in many different ways in many kinds of devices. To properly utilize the full set of possible INSTEON message types, devices must share a common set of specific, preassigned number values for the one- and two-byte Commands, two-byte Device Types, one-byte Device Attributes, one- or two-byte ACK statuses, and one- or two-byte NAK reasons. Smarthome maintains the database of allowable values for these parameters.

Because INSTEON devices are individually preassigned a three-byte Address at the time of manufacture, complex procedures for assigning network addresses in the field are not needed. Instead, INSTEON devices are logically linked together in the field using a simple, uniform procedure.

INSTEON Extended messages allow programmers to devise all kinds of meanings for the User Data that can be exchanged among devices. For example, many INSTEON devices include an interpreter for an application language, called SALad, which is compiled into token strings and downloaded into devices using Extended messages. Also, secure messaging can be implemented by sending encrypted payloads in Extended messages.

In this section

INSTEON Commands

INSTEON's simplicity stems from the fact that Standard messages are all 10 bytes in length, and they contain just two payload bytes, Command 1 and Command 2.

Smarthome maintains the table of possible INSTEON Commands. This table is currently very sparsely populated. Designers who wish to create INSTEON devices that implement new Commands should contact Smarthome at info@insteon.net. The current INSTEON Commands are enumerated in the *INSTEON Command Tables*.

The basic rules for handling INSTEON Commands depend on whether a device is currently acting as a Controller or a Responder.

Controllers have a repertoire of Commands that they can send, usually set by the firmware in the device. Examples for a lighting controller might include *On*, *Off*, *Bright*, *Dim*, *Fast On*, and *Fast Off*. Obviously, a Controller can only send the Commands it knows about, and no others.

Responders likewise have a repertoire of Commands that they can act upon. For example, a lamp dimmer's firmware might contain procedures to respond to *On*, *Off*, *Bright*, and *Dim* Commands, but not *Fast On* and *Fast Off*. A Responder will only act on the Commands *it* knows about, and no others.

Smarthome maintains a cross-reference between Device Classes and Commands that a device must implement for INSTEON conformance certification. Contact Smarthome at info@insteon.net for more information.

In this section

INSTEON Command 1 and 2

INSTEON Command 1

Command 1 holds an 8-bit number representing the INSTEON Primary Command to execute.

INSTEON Command 2

The interpretation of the Command 2 field depends on the Primary Command in the Command 1 field.

Parameter

The Command 2 field can be a parameter for the Primary Command. For example, Command 0x11 (*On*) has a parameter in Command 2 ranging from 0x00 to 0xFF representing the On-Level.

Subcommand

Command 2 can act as a Subcommand for certain blocks of Primary Commands. Taken together, the 2-byte Primary-plus-Subcommands allow for expansion of the command space.

Group Number

For Groups of linked devices, Controllers first send a Group Broadcast message containing a Primary Command to all devices in the Group at once. Responders in the Group will execute the Primary Command right away, but they will not reply with an acknowledgement. To ensure reliability, a Controller follows up a Group Broadcast with a Group Cleanup message sent individually to each member of the Group. In Group Cleanup messages Command 2 contains the Group Number.

Acknowledgement

Command 2 can return data to the sending device in an acknowledgement message. For example, an ACK message responding to Commands *On*, *Off*, *Bright*, *Dim*, *Fast On*, *Fast Off*, *Start Manual Change*, and *Stop Manual Change* will hold the On-Level in the Command 2 field. If one of these Commands is sent and the response is a NAK message, then the Command 2 field will hold the NAK reason code.

INSTEON Command Tables

There are two classes of INSTEON Commands:

INSTEON Common Commands
Used in Direct (Point-to-Point), Group Broadcast, and Group Cleanup messages.

INSTEON Broadcast Commands
Used in Broadcast (but not Group Broadcast) messages.

The Command Numbers are reused in each class. For example, a Command Number **0x01** in a Broadcast message denotes *SET Button Pressed Slave*, but a Command Number **0x01** in a Direct message denotes *Assign to Group*. See *INSTEON Message Summary Table* for more information on INSTEON message types.

In This Section

INSTEON Common Commands

These INSTEON Commands can be used in Direct, Group Broadcast, and Group Cleanup INSTEON messages. (Use *INSTEON Broadcast Commands* in Broadcast messages.) Recipients of Direct and Group Cleanup messages will respond to the message originator with an Acknowledge message, but recipients of Group Broadcast messages will not (see *INSTEON Message Summary Table*).

INSTEON Common Command Summary Table

This table lists all current INSTEON Common Commands.

Command Name
Gives the descriptive name of the INSTEON Command. Commands marked * have not been implemented and are reserved for possible future use.

Command 1
Gives the Command Number used in the Command 1 field of the INSTEON message to identify this INSTEON Command.

Command 2
Describes the contents of the Command 2 field of the INSTEON message. In Group Cleanup messages, the Command 2 field contains the Group Number. Set this field to 0x00 if not otherwise specified.

Note
See the item with the same note number in *INSTEON Common Command Details* for more information.

Description
Briefly describes what the command does.

Command Name	Command 1	Command 2	Note	Description
*Reserved	**0x00**		1	
Assign to Group	**0x01**	Group Number	2	Used during *INSTEON Device Linking* session.
Delete from Group	**0x02**	Group Number	2	Used during unlinking session.
*Read Connections	**0x03**	Group Number 0=All	1	Send as Extended message. Returned Extended ACK message will contain the connections.
*Read Device Data	**0x04**		1	*Device Type* and other information specific to INSTEON device returned in Extended Message.
*Factory Reset	**0x05**	Security Code	1	Reset to Factory default conditions.
*Device Reset	**0x06**	Security Code	1	Reset to Factory default conditions but keep connections and links.
*Terminate Download	**0x07**		1	Stops download.
*Reserved	**0x08**		1	
*Enter Master Program Mode	**0x09**	Group Number	1	

Command Name	Command 1	Command 2	Note	Description
*Enter Slave Program Mode	**0x0A**		1	
*Reserved	**0x0B ⇒ 0x0F**		1	
Ping	**0x10**			Receiving device first returns an ACK message, then it sends a *Device Identification Broadcast*.
ON	**0x11**	On Level (0x00 ⇒ 0xFF), or Group number		
*Fast ON	**0x12**	On Level (0x00 ⇒ 0xFF), or Group number	1	
OFF	**0x13**	None, or Group number		
*Fast OFF	**0x14**	None, or Group number	1	
Bright	**0x15**	None, or Group number		Brighten one step. There are 32 steps from off to full brightness. Returned ACK message will contain the *On-Level* in Command 2.
Dim	**0x16**	None, or Group number		Dim one step. There are 32 steps from off to full brightness. Returned ACK message will contain the *On-Level* in Command 2.
Start Manual Change	**0x17**	1 = Up, 0 = Down		Begin changing *On-Level*.
Stop Manual Change	**0x18**	None		Stop changing *On-Level*.
Status Request	**0x19**	None		Returned ACK message will contain the status (often the *On-Level*) in Command 2. Command 1 will contain a *Link Database Delta* number that increments every time there is a change in the addressee's *INSTEON Link Database*.
*Status Report	**0x1A**	New Status	1	Sent to controller when local state changes.
*Read Last Level	**0x1B**	None	1	Returned ACK message will contain the requested data in Command 2.
*Set Last Level	**0x1C**	On-Level	1	
*Read Preset Level	**0x1D**	None	1	Returned ACK message will contain the requested data in Command 2.
*Set Preset Level	**0x1E**	On-Level	1	

Command Name	Command 1	Command 2	Note	Description
*Read Operating Flags	**0x1F**	None	1	Returned ACK message will contain the requested data in Command 2.
*Set Operating Flags	**0x20**	Flags	1	
*Delete Group X10 Address	**0x21**	Group Number	1	Sent to a device to delete the assigned X10 address for a group within the device.
*Load Off	**0x22**		1	Indicates manual load status change.
*Load On	**0x23**		1	Indicates manual load status change.
Do Read EE	**0x24**	None		Read initialization values from EEPROM, so that they will take effect after being poked.
*Level Poke	**0x25**	On-Level	1	Set new *On-Level*.
*Rate Poke	**0x26**	Ramp Rate	1	Set new *Ramp Rate*.
*Current Status	**0x27**	Usually an On-Level		Can be used to update a companion device.
Set Address MSB	**0x28**	High byte of 16-bit address	3	Set Most-significant Byte of EEPROM address for peek or poke.
Poke	**0x29**	Byte to write	3	Poke Data byte into address previously loaded with *Set Address MSB* and *Peek* commands (*Peek* sets LSB).
Poke Extended	**0x2A**	LSB of address to begin writing up to 13 bytes to	3	Poke up to 13 bytes starting at the address whose high byte was previously set using *Set Address MSB*. This command must be sent within an Extended message. Put the number of bytes to poke in the first byte of *User Data*, and the actual bytes to poke in the remaining 13 bytes.
Peek	**0x2B**	LSB of address to peek or poke	3	*Peek* can be sent within a Standard or an Extended message. If Extended, put the number of bytes to peek in the first byte of *User Data*. The returned ACK message will contain the first peeked byte in *Command 2*. The Extended ACK message to an Extended *Peek* will return up to 14 remaining bytes in *User Data*. *Peek* is also used to set the LSB for one-byte pokes.
Peek Internal	**0x2C**	LSB of internal memory address to read from	3	Works like *Peek*, except only used to read from internal memory of a Smarthome ControLinc V2.

Command Name	Command 1	Command 2	Note	Description
Poke Internal	**0x2D**	Byte to write	3	Works like *Poke*, except only used to write one byte into internal EEPROM of a Smarthome ControLinc V2.
*Reserved	**0x2E ⇒ 0x80**		1	
*Assign to Companion Group	**0x81**		1	Allows Slaves of a Master to follow the Master when the Master is controlled by a companion device.
*Reserved	**0x82 ⇒ 0xEF**		1	
*Reserved for RF Devices	**0xF0 ⇒ 0xFB**		1	
Send RF Test Signal	**0xFC**			Causes RF devices to send a 4-second test message for other RF units to hear.
Request RF Test Report	**0xFD**			Causes an RF receiver to transmit back the results of the last RF test.
Reset RF Test Report	**0xFE**			Clears an RF receiver's memory of the results of the previous RF Test.
RF Air Only Test	**0xFF**			Send INSTEON message via RF only (not powerline).

INSTEON Common Command Details

The numbers below refer to the **Note** column in the above *INSTEON Common Command Summary Table*.

1. **Not Implemented**

   These Commands are reserved for possible future use.

2. **Assign to Group (0x01), Delete from Group (0x02)**

   These Commands are used for *INSTEON Device Linking*. See *Example of an INSTEON Linking Session*. For sample INSTEON messages that use the **Assign to Group (0x01)** command.

3. **Peek and Poke (0x29 ⇒ 0x2D)**

   You can use the *Peek* and *Poke* INSTEON Commands (**0x28 ⇒ 0x2D**) to remotely read or write memory in INSTEON devices. For example, you could inspect or alter the *INSTEON Link Database* of an INSTEON device to determine which *INSTEON Groups* it belongs to (i.e. which other INSTEON devices it has links to).

   SALad-enabled INSTEON devices, such as *The Smarthome PowerLinc Controller*, map all memory to one *Flat Memory Map*, but other INSTEON devices, such as Smarthome's ControLinc™ V2, LampLinc™ V2, and SwitchLinc™ V2, do not have a flat address space. For flat devices, use the **Peek (0x2B)**, **Poke (0x29)**, and

Poke Extended (0x2A) to access all memory. For non-flat devices use the **Peek (0x2B)**, **Poke (0x29)**, and **Poke Extended (0x2A)** to access external EEPROM. You only can use **Peek Internal (0x2C)**, and **Poke Internal (0x2D)** to access internal EEPROM of the ControLinc™ V2. Note that you cannot access a device's ROM using these commands.

To peek or poke remote memory data, first use the **Set Address MSB (0x28)** command to set the high byte of the 16-bit address you want to read from or write to. You will set the LSB of the address differently depending on what you will be doing next. If you are going to **Poke (0x29)** or **Poke Internal (0x2D)** one byte of data, you first have to execute a **Peek (0x2B)** command to set the address LSB. In the other cases, **Peek (0x2B)**, **Peek Internal (0x2C)**, and **Poke Extended (0x2A)**, you will set the LSB in the command itself. Note that the address LSB does *not* auto-increment.

To peek one byte of data, use **Peek (0x2B)** or **Peek Internal (0x2C)** in a message of Standard length. The Standard length Acknowledge message that you receive back will contain the peeked byte in the *Command 2* field.

To peek more than one byte of data in a single message, use **Peek (0x2B)** or **Peek Internal (0x2C)** in a message of Extended length. Put the number of bytes you wish to read, 1 through 15 (0x00 ⇒ 0x0F) in the first byte of the *User Data* field. It does not matter what you put in the remaining 13 bytes of the *User Data* field. The Extended length Acknowledge message that you receive back will contain the first peeked byte in the *Command 2* field, and the remaining peeked bytes (up to 14 of them) in the *User Data* field.

To poke one byte of data, use **Poke (0x29)** or **Poke Internal (0x2D)** in a message of Standard length. Remember to set the address you want to poke to by using a **Set Address MSB (0x28)** command followed by a **Peek (0x2B)** command. The *Command 2* field of the Standard length Acknowledge message that you receive back will contain the value of the byte you are poking before it was altered by the poke.

To poke more than one byte of data in a single message, use **Poke Extended (0x2A)** in a message of Extended length. Put the LSB of the starting address you will poke the bytes to in the *Command 1* field. Put the number of bytes you wish to poke, 1 through 13 (0x00 ⇒ 0x0D) in the first byte of the *User Data* field, and the bytes you wish to poke the remaining 13 bytes of the *User Data* field. If you are poking fewer than 13 bytes, it does not matter what the unused bytes of the *User Data* field contain. The Extended length Acknowledge message that you receive back will contain the value of the poked bytes before you altered them. The first byte will be in the *Command 2* field, and the remaining bytes (up to 13 of them) will be in the *User Data* field.

NAK Reason Codes

If the recipient of a Direct or Group Cleanup message responds to the message originator with a NAK, the Acknowledge message will contain the reason for the NAK in the Command 2 field (see *INSTEON Message Summary Table*). These are the NAK Reason codes:

Code	NAK Reason
0x00 ⇒ 0xFD	Reserved
0xFE	No Load
0xFF	Not in Group

INSTEON Broadcast Commands

These INSTEON Commands can only be used in INSTEON Broadcast Messages. (Use *INSTEON Common Commands* in Direct, Group Broadcast, and Group Cleanup messages.)

INSTEON Broadcast messages contain information for all recipients. In Broadcast messages the *To Address* field of the message contains the 16-bit *Device Type* and 8-bit *Firmware Revision* (see *INSTEON Message Summary Table*).

Recipients of Broadcast messages will not respond with an Acknowledge message, but they may engage in a followup conversation if applicable. For an example of how this works, see *Example of an INSTEON Linking Session*.

INSTEON Broadcast Command Summary Table

This table lists all currently INSTEON Broadcast Commands.

Command Name
Gives the descriptive name of the INSTEON Broadcast Command. Commands marked * have not been implemented and are reserved for possible future use.

Command 1
Gives the Command Number used in the Command 1 field of the INSTEON Broadcast message to identify this INSTEON Command.

Command 2
Describes the contents of the Command 2 field of the INSTEON message. This field is not used by INSTEON Broadcast Commands. Set this field to 0x00.

Note
See the item with the same note number in *INSTEON Broadcast Command Details* for more information.

Description
Briefly describes what the command does.

Command Name	Command 1	Command 2	Note	Description
*Announce	**0x00**	None	1	
SET Button Pressed Slave	**0x01**	None	2	Possible Program Mode for a Slave device
*Status Changed	**0x02**	None	1	
SET Button Pressed Master	**0x03**	None	2	Possible Program Mode for a Master device
*Reserved	**0x04 ⇒ 0x48**		1	
Debug Report	**0x49**	None	3	The high and low bytes of the Program Counter (for a SALad program being remotely debugged) are returned in the two low bytes of the *To Address*. See *IBIOS Remote Debugging*.
*Reserved	**0x4A ⇒ 0xFF**		1	

INSTEON Broadcast Command Details

The numbers below refer to the **Note** column in the above *INSTEON Broadcast Command Summary Table*.

1. **Not Implemented**

   These Commands are reserved for possible future use.

2. **SET Button Pressed Slave (0x01), SET Button Pressed Master (0x03)**

   These Commands are used for device linking. For an example of how this works, see *Example of an INSTEON Linking Session*.

3. **Debug Report (0x49)**

   This command returns the 16-bit address contained in the Program Counter of a SALad program being remotely debugged. The high and low bytes are returned in the two low bytes of the *To Address*. See *IBIOS Remote Debugging* for details.

INSTEON Command Examples

To see some typical examples of INSTEON Commands being used in INSTEON messages, look in the sections *Example of an INSTEON Linking Session* and *Example of INSTEON Group Conversation*.

The examples in this section are fairly sophisticated. To use them, you must know what you are doing. In particular, you should fully understand the *INSTEON Link Database*, which is not documented until later in this Developer's Guide.

Some of the examples involve directly poking data into an INSTEON Device's Link Database. If you do this improperly you can corrupt the database and cause the device to malfunction, in which case you will need to perform a 'Factory Reset' on the INSTEON device. The User Guide for the device will explain how to do this.

Common Smarthome INSTEON Device Memory Locations

This table gives some memory locations found in many Smarthome INSTEON devices, such as the LampLinc™, SwitchLinc™, and KeypadLinc™.

Address	Variable Name	Description
0x0301	EESize	MSB of size of EEPROM chip in device
0x0302	EEVersion	Firmware Revision number
0x0320	EEOnLevel	Preset On-Level for lighting control
0x0321	EERampRate	Ramp Rate for lighting control
0x0330	EEX10BaseHouse	Base X10 House Code for device
0x0331	EEX10BaseUnit	Base X10 Unit Code for device

Assumptions for the Following Examples

The numbers in these examples are all hexadecimal.

We assume that you are using *The Smarthome PowerLinc Controller* (PLC) with an INSTEON Address of 01.02.03 to send and receive INSTEON messages.

The INSTEON device you are talking to is a Smarthome LampLinc V2 Dimmer with an INSTEON Address of A1.A2.A3.

All INSTEON messages have a *Max Hops* of 3 and a *Hops Left* of 3, so the low nibble of the *Flags* byte is always 0xF.

These examples involve peeking and poking data directly in the LampLinc's memory using the various INSTEON peek and poke commands. We assume that you have read *INSTEON Common Command Details*, Note 3, which explains how to use these commands.

Poke a New Ramp Rate

PLC Sends		LampLinc Responds
Set Address MSB to 03		**OK**
From Address	**01 02 03** (PLC)	**A1 A2 A3** (LampLinc)
To Address	**A1 A2 A3** (LampLinc)	**01 02 03** (PLC)
Flags	**0F** (Direct Message)	**2F** (Direct ACK Message)
Command 1	**28** (*Set Address MSB*)	**28** (*Set Address MSB*)
Command 2	**03** (MSB of 0x0321)	**03** (MSB of 0x0321)
Peek to Set Address LSB to 21		**OK**
From Address	**01 02 03** (PLC)	**A1 A2 A3** (LampLinc)
To Address	**A1 A2 A3** (LampLinc)	**01 02 03** (PLC)
Flags	**0F** (Direct Message)	**2F** (Direct ACK Message)
Command 1	**2B** (*Peek*)	**2B** (*Peek*)
Command 2	**21** (LSB of 0x0321)	**XX (**Byte at 0x0321)
Poke 80 at Address 0321		**OK**
From Address	**01 02 03** (PLC)	**A1 A2 A3** (LampLinc)
To Address	**A1 A2 A3** (LampLinc)	**01 02 03** (PLC)
Flags	**0F** (Direct Message)	**2F** (Direct ACK Message)
Command 1	**29** (*Poke*)	**29** (*Poke*)
Command 2	**80** (Byte to poke)	**80** (Byte to poke)

Peek Link Database Record at 0x0FF8

Here we assume that the LampLinc's *Non-PLC Link Database* has one record at address 0FF8. The INSTEON Address is AABBCC, and the entire 8-byte record is **A2 01 AABBCC FF 1C 00**.

PLC Sends		LampLinc Responds
Set Address MSB to 0F		**OK**
From Address	**01 02 03** (PLC)	**A1 A2 A3** (LampLinc)
To Address	**A1 A2 A3** (LampLinc)	**01 02 03** (PLC)
Flags	**0F** (Direct Message)	**2F** (Direct ACK Message)
Command 1	**28** (*Set Address MSB*)	**28** (*Set Address MSB*)
Command 2	**0F** (MSB of 0x0FF8)	**0F** (MSB of 0x0FF8)
Peek Extended at Address LSB of F8		**OK**
From Address	**01 02 03** (PLC)	**A1 A2 A3** (LampLinc)
To Address	**A1 A2 A3** (LampLinc)	**01 02 03** (PLC)
Flags	**1F** (Direct Extended Message)	**3F** (Direct ACK Extended Message)
Command 1	**2B** (*Peek*)	**2B** (*Peek*)
Command 2	**F8** (LSB of 0x0FF8)	**A2** (1st Byte at 0x0FF8)
User Data	**00 00 00 00 00 00 00 00 00 00 00 00 00 00**	**01 AA BB CC FF 1C 00 00 00 00 00 00 00 00**

Poke Link Database Record at 0x0FF8

Here we assume that we just used the previous example to read the one record at FFF8 in the LampLinc's *Non-PLC Link Database*, and it contained **A2 01 AABBCC FF 1C 00**. We will change the record to **A2 01 010203 FF 1C 00**, which will enroll the PLC. We will use an Extended Poke to poke only the 3 bytes **01 02 03** at address 0FFA.

PLC Sends		LampLinc Responds
Set Address MSB to 0F		**OK**
From Address	**01 02 03** (PLC)	**A1 A2 A3** (LampLinc)
To Address	**A1 A2 A3** (LampLinc)	**01 02 03** (PLC)
Flags	**0F** (Direct Message)	**2F** (Direct ACK Message)
Command 1	**28** (*Set Address MSB*)	**28** (*Set Address MSB*)
Command 2	**0F** (MSB of 0x0FF8)	**0F** (MSB of 0x0FF8)
Poke Extended at Address 0FFA		**OK**
From Address	**01 02 03** (PLC)	**A1 A2 A3** (LampLinc)
To Address	**A1 A2 A3** (LampLinc)	**01 02 03** (PLC)
Flags	**1F** (Direct Extended Message)	**3F** (Direct ACK Extended Message)
Command 1	**29** (*Poke*)	**29** (*Poke*)
Command 2	**FA** (LSB of 0x0FFA)	**FA** (LSB of 0x0FFA)
User Data	**03 01 02 03 00 00 00 00 00 00 00 00 00 00**	**XX XX XX XX XX XX XX XX XX XX XX XX XX XX**

Note that the first byte of the *User Data* field is the number of bytes to poke.

INSTEON Device Classes

The number of different kinds of devices that can be connected to an INSTEON network is virtually unlimited. Rather than relying on an elaborate scheme for discovery of device capabilities, INSTEON's designers opted for a very simple, yet expandable method for devices to identify themselves—they Broadcast a *SET Button Pressed* message containing device classification information. This information, the *Device Type*, *Device Attributes*, and *Firmware Revision*, appears in fixed-length fields within the Broadcast message.

A 16-bit number identifies an INSTEON Device Type. See the *INSTEON Device Type Summary Table* for a list of currently defined Device Types.

Device Identification Broadcast

INSTEON devices identify themselves to other devices on an INSTEON network by sending a *SET Button Pressed* Broadcast message. This message contains a number of fields that describe the product type and capabilities.

A 16-bit *Device Type* field appears in the most significant 2 bytes of the To Address field, followed by the *Firmware Revision* in the least significant byte. A *Device Attributes* byte appears in the Command 2 field.

<table>
<tr><th colspan="6">INSTEON SET Button Pressed Broadcast Message</th></tr>
<tr><th>From Address</th><th colspan="2">To Address</th><th>Message Flags</th><th>Command 1</th><th>Command 2</th></tr>
<tr><td>3 bytes</td><td colspan="2">3 bytes</td><td>1 byte</td><td>1 byte</td><td>1 byte</td></tr>
<tr><td rowspan="2"></td><td>2 bytes</td><td>1 byte</td><td rowspan="2">1000xxxx</td><td rowspan="2">SET Button Pressed Slave (0x01) or SET Button Pressed Master (0x03)</td><td rowspan="2">Device Attributes</td></tr>
<tr><td>Device Type</td><td>Firmware Revision</td></tr>
</table>

Device Type

Each INSTEON device contains a 2-byte Device Type identifier, which allows for 65,536 different possible Device Types. Smarthome assigns these numbers to device manufacturers. Currently, Types are being assigned sequentially. See the *INSTEON Device Type Summary Table* for a list of currently defined Device Types. Designers who wish to develop INSTEON-enabled products should contact Smarthome at info@insteon.net for more information.

Device Attributes

When an INSTEON device identifies itself by sending out a *SET Button Pressed* Broadcast message, the message includes a Device Attributes byte in the Command 2 field. Although currently unused, this byte can contain individual bit flags that could be interpreted differently for each Device Type.

Firmware Revision

An INSTEON device's firmware revision number appears in the least significant byte of the *To Address* field of a *SET Button Pressed* Broadcast message. The high nibble (4 bits) gives the major revision number and the low nibble gives the minor revision.

INSTEON Device Type Summary Table

These are the currently defined INSTEON Device Types.

Device Type	Product
0x0000	Smarthome PowerLinc™ V2 Controller
0x0001 ⇒ 0x0003	Reserved
0x0004	Smarthome ControLinc™ V2
0x0005	Smarthome RF RemoteLinc™
0x0006 ⇒ 0x00FF	Reserved
0x0100	Smarthome LampLinc™ V2 Dimmer
0x0101	Smarthome SwitchLinc™ V2 Dimmer
0x0102 ⇒ 0x0108	Reserved
0x0109	Smarthome KeypadLinc™ V2
0x010A ⇒ 0x0113	Reserved
0x0114	Smarthome SwitchLinc™ V2 Companion
0x0115 ⇒ 0x0208	Reserved
0x0209	Smarthome ApplianceLinc™ V2
0x020A	Smarthome SwitchLinc™ V2 Relay
0x020B ⇒ 0xFFFF	Reserved

INSTEON Device Linking

When a user adds a new device to an INSTEON network, the newcomer device joins the network automatically, in the sense that it can hear INSTEON messages and will repeat[1] them automatically according to the INSTEON protocol. So, no user intervention is needed to establish an INSTEON network of communicating devices.

However, for one INSTEON device to control other INSTEON devices, the devices must be logically linked together. INSTEON provides two very simple methods for linking devices—manual linking using button pushes, and electronic linking using INSTEON messages.

INSTEON Groups

During linking, users create associations between events that can occur in an INSTEON Controller, such as a button press or a timed event, and the actions of a Group of one or more Responders. This section defines *Groups and Links* and gives *Examples of Groups*.

Groups and Links

A Group is a set of logical Links between INSTEON devices. A Link is an association between a Controller and a Responder or Responders. Controllers originate Groups, and Responders join Groups.

Internally, in an *INSTEON Link Database* maintained by INSTEON devices, a Group ID consists of 4 bytes—the 3-byte address of the Controller, and a 1-byte Group Number. A Controller assigns Group Numbers as needed to the various physical or logical events that it supports. For example, a single press of a certain button could send commands to one Group, and a double press of the same button could send commands to another Group. The Controller determines which commands are sent to which Groups.

A Group can have one or many members, limited only by the memory available for the Link Database.

Examples of Groups

A device configured as a wall switch with a paddle could be designed to support one, two, or three Groups, as shown in the following examples.

One Group		
Controller Event	**Group**	**Action of Group Responders**
Tap Top	1	Turn On
Tap Bottom	1	Turn Off
Hold Top	1	Brighten
Hold Bottom	1	Dim

Two Groups		
Controller Event	**Group**	**Action of Group Responders**
Tap Top	1	Turn On
Tap Top Again	1	Turn Off
Tap Bottom	2	Turn On
Tap Bottom Again	2	Turn Off

Three Groups		
Controller Event	**Group**	**Action of Group Responders**
Tap Top	1	Turn On
Tap Bottom	1	Turn Off
Double Tap Top	2	Turn On
Double Tap Bottom	2	Turn Off
Triple Tap Top	3	Turn On
Triple Tap Bottom	3	Turn Off

Methods for Linking INSTEON Devices

There are two ways to create links among INSTEON devices, *Manual Linking* and *Electronic Linking*. This section also gives an *Example of an INSTEON Linking Session*.

Manual Linking

Easy setup is very important for products sold to a mass market. INSTEON devices can be linked together very simply:

- Push and hold for 10 seconds the button that will control an INSTEON device.
- Push and hold a button on the INSTEON device to be controlled.

This kind of manual linking implements a form of security. Devices cannot be probed by sending messages to discover their addresses—a user must have physical possession of INSTEON devices in order to link them together.

Designers are free to add to this basic linking procedure. For example, when multiple devices are being linked to a single button on a Controller, a *multilink* mode could enable a user to avoid having to press and hold the button for 10 seconds for each new device.

There must also be procedures to unlink devices from a button, and ways to clear links from buttons in case devices linked to them are lost or broken. See the *INSTEON Link Database* section below for more information on this point.

Electronic Linking

As the example below shows (see *Example of an INSTEON Linking Session*), linking is actually accomplished by sending INSTEON messages, so a PC or other device can create links among devices if the device addresses are known and if devices can respond to the necessary commands.

To maintain security, PC-INSTEON interface devices such as Smarthome's PowerLinc™ V2 Controller (PLC) mask the two high bytes of the address fields in INSTEON messages received from unknown devices. Devices are only known if there is a link to the device stored in the Link Database of the PLC, or if the message's To Address matches that of the PLC. Such links must have been previously established by manual button pushing or else by manually typing in the addresses of linked devices (see *Masking Non-linked Network Traffic*, below).

Example of an INSTEON Linking Session

This section outlines the message exchange that occurs when a Controller and Responder set up a link relationship. In this scenario, a Smarthome ControLinc™ V2 is the Controller, and a Smarthome LampLinc™ V2 is the Responder. Numbers are in hexadecimal.

Message 1	**ControLinc: "I'm looking for Group members"**		
`00 00 CC 00 04 0C 8F` **`03`** `00`	ControLinc, with address of 00 00 CC, sends a *SET Button Pressed Master* Broadcast message indicating it is now listening for Responders to be added to Group 1.		
	From Address		00 00 CC (ControLinc)
	To Address	Device Type	00 0A (ControLinc)
		Firmware Version	0C
	Flags		8F (Broadcast Message, 3 Max Hops, 3 Hops Left)
	Command 1		**03** (*SET Button Pressed Master*)
	Command 2	Device Attributes	00 (Not used)

Message 2	**LampLinc: "I'll join your Group"**		
`00 00 AA 01 00 30 8F` **`01`** `00`	LampLinc, with address of 00 00 AA, sends a *SET Button Pressed Slave* Broadcast message. When the ControLinc hears this, it will respond with a message to join Group 1.		
	From Address		00 00 AA (LampLinc)
	To Address	Device Type	00 02 (LampLinc)
		Firmware Version	30
	Flags		8F (Broadcast Message, 3 Max Hops, 3 Hops Left)
	Command 1		**01** (*SET Button Pressed Slave*)
	Command 2	Device Attributes	00 (Not used)

Message 3	**ControLinc: "Okay, join Group 1"**	
`00 00 CC 00 00 AA 0F` **`01 01`**	ControLinc (00 00 CC) sends message to LampLinc (00 00 AA) to join Group 1.	
	From Address	00 00 CC (ControLinc)
	To Address	00 00 AA (LampLinc)
	Flags	0F (Direct Message, 3 Max Hops, 3 Hops Left)
	Command 1	**01** (*Assign to Group*)
	Command 2	**01** (Group 1)

Message 4	**LampLinc: "I joined Group 1"**	
00 00 AA 00 00 CC 2F **01 01**	LampLinc (00 00 31) sends ACK to ControLinc (00 00 10).	
	From Address	00 00 AA (LampLinc)
	To Address	00 00 CC (ControLinc)
	Flags	2F (ACK of Direct Message, 3 Max Hops, 3 Hops Left)
	Command 1	**01** (*Assign to Group*)
	Command 2	**01** (Group 1)

Example of INSTEON Group Conversation

This example illustrates how messages are passed from device to device in a group. In this scenario, a Smarthome ControLinc™ V2 linked to two Smarthome LampLinc™ V2 Dimmers in Group 1 commands them to turn on. Numbers are in hexadecimal.

Note that the Group Broadcast message (which both LampLinc Dimmers should respond to immediately) is followed by an acknowledged Group Cleanup message to each LampLinc Dimmer (in case they didn't get the Broadcast).

Message 1	**ControLinc: "Group 1, turn on"**		
00 00 CC 00 00 **01** CF **11** 00	ControLinc, with address of 00 00 CC, sends a Group Broadcast message to Group 1, with a command of *On*.		
	From Address		00 00 CC (ControLinc)
	To Address	Unused	00 00
		Group Number	**01**
	Flags		CF (Group Broadcast Message, 3 Max Hops, 3 Hops Left)
	Command 1		**11** (*On*)
	Command 2		00 (Unused)

Message 2	**ControLinc: "LampLinc A, turn on"**		
00 00 CC 00 00 AA 4F **11 01**	ControLinc (00 00 CC) sends a Group Cleanup message to LampLinc A (00 00 AA) in Group 1, with a command of *On*.		
	From Address		00 00 CC (ControLinc)
	To Address		00 00 AA (LampLinc A)
	Flags		4F (Group Cleanup Message, 3 Max Hops, 3 Hops Left)
	Command 1		**11** (*On*)
	Command 2	Group Number	**01**

Message 3	**LampLinc A: "I turned on"**		
00 00 AA 00 00 CC 2F **11 01**	LampLinc A (00 00 AA) sends ACK to ControLinc (00 00 CC).		
	From Address		00 00 AA (LampLinc A)
	To Address		00 00 CC (ControLinc)
	Flags		2F (ACK of Direct Message, 3 Max Hops, 3 Hops Left)
	Command 1		**11** (*On*)
	Command 2	Group Number	**01**

Message 4	**ControLinc: "LampLinc B, turn on"**		
00 00 CC 00 00 BB 4F **11 01**	ControLinc (00 00 CC) sends a Group Cleanup message to LampLinc B (00 00 BB) in Group 1, with a command of *On*.		
	From Address		00 00 CC (ControLinc)
	To Address		00 00 BB (LampLinc B)
	Flags		4F (Group Cleanup Message, 3 Max Hops, 3 Hops Left)
	Command 1		**11** (*On*)
	Command 2	Group Number	**01**

Message 5	**LampLinc B: "I turned on"**		
00 00 BB 00 00 CC 2F **11 01**	LampLinc B (00 00 BB) sends ACK to ControLinc (00 00 CC).		
	From Address		00 00 BB (LampLinc B)
	To Address		00 00 CC (ControLinc)
	Flags		2F (ACK of Direct Message, 3 Max Hops, 3 Hops Left)
	Command 1		**11** (*On*)
	Command 2	Group Number	**01**

An INSTEON Controller will send Group Cleanup commands to all Responder devices in a Group, unless other INSTEON traffic interrupts the cleanup, in which case the Group Cleanups will stop.

INSTEON Link Database

Every INSTEON device stores a Link Database in nonvolatile memory, representing Controller/Responder relationships with other INSTEON devices. Controllers know which Responders they are linked to, and Responders know which Controllers they are linked to. Link data is therefore distributed among devices in an INSTEON network.

If a Controller is linked to a Responder, and the Responder is removed from the network without updating the Controller's Link Database, then the Controller will retry messages intended for the missing Responder. The retries, which are guaranteed to fail, will add unnecessary traffic to the network. It is therefore very important for users to unlink INSTEON Responder devices from Controllers when unused Responders are removed. Unlinking is normally accomplished in the same way as linking—press and hold a button on the Controller, then press and hold a button on the Responder.

Because lost or broken Responder devices cannot be unlinked using a manual unlinking procedure, Controllers must also have an independent method for unlinking missing Responders. Providing a 'factory reset' procedure for a single Controller button, or for the entire Controller all at once, is common.

When a Controller is removed from the network, it should likewise be unlinked from all of its Responder devices before removal, or else the Link Databases in the Responders will be cluttered up with obsolete links. A 'factory reset' should be provided for Responder devices for this purpose.

There are two forms of Link Database Record, one for SALad-enabled devices such as Smarthome's PowerLinc™ V2 Controller (PLC), and another for non-PLC devices. This section describes both.

In This Section

PLC Link Database
: Explains the contents of the Link Database for the Smarthome PowerLinc V2 Controller.

Non-PLC Link Database
: Gives the layout of Link Database Records for other INSTEON devices.

PLC Link Database

The PLC Link Database starts at the top of external (serial) EEPROM and grows downward. Because of the way *Flat Memory Addressing* works, top of memory can always be found at 0xFFFF.

Each PLC Link Database Record is 8 bytes long, so the first physical record starts at 0xFFF8, the second physical record starts at 0xFFF0, and so on. The Link Database starts out containing a minimum of 128 physical records, so it occupies the top 1024 bytes of external EEPROM. The Link Database can grow larger than 1024 bytes, until it bumps up against the SALad application that grows upward from the bottom of EEPROM.

In what follows, the 3-byte INSTEON Address contained in a record is called the *Device ID*. The high byte (MSB) of the Device ID is *ID2*, the middle byte is *ID1*, and the low byte (LSB) is *ID0*. *MSb* and *LSb* refer to most and least significant *bits*, respectively. All addresses refer to the *Flat Memory Map*.

The PLC Link Database is organized as a set of 128 threads, where each thread is a (forward) linked list. The address of the first record in a thread is calculated from the 7 MSbs of ID0 of the Device ID, as described in the table.

PLC Link Database Record Format		
Field	**Length (bytes)**	**Description**
Record Control	2	76543210 76543210 Class: XXXXXXXX XXXXXXXX 00 Deleted (ID0 LSb indicates end of DB) Link----++++++++--+++++||| 01 Other (extended class) Class------------------++| 10 Slave (INSTEON Responder) ID0_LSb------------------+ 11 Master (INSTEON Controller) *Link* gives the 13 MSbs of the 16-bit address of the next record in this one of 128 possible database threads. The 3 LSbs of this address are 0. If the 8 MSbs of *Link* are 0, this is the end of the thread. If the *Class* is 00 and *ID0_LSb* is 1, then this is the last physical record in the database. The 16-bit address of the first record in a thread is computed from ID0 by shifting ID0 left 2 (i.e. multiplying ID0 by 4), complementing the 16-bit result, then setting the 3 LSbs to 0. *Class* indicates whether the record is available (Deleted), for an INSTEON Controller (Master), for an INSTEON Responder (Slave), or user-defined (Other). *ID0 LSb* is the LSb of ID0 in the Device ID.
ID1	1	Middle byte of the Device ID
ID2	1	High (MSB) byte of the Device ID
Group	1	Group Number (for *INSTEON Groups*)
Data 1	1	Link-specific data (e.g. *On-Level*)
Data 2	1	Link-specific data (e.g. Ramp Rates, Setpoints, etc.)
Data 3	1	Unused

If you are searching the database for a given Device ID, calculate the starting address for the thread using the given ID's low byte, then follow the links until ID1, ID2, and ID0_LSb match.

If you are searching the database for something else, such as all records with a given Group Number and/or all records in a given Class, then you will have to look at all 128 threads. As you increment through the threads, keep a *Thread_Index* counter running from 0x00 to 0x7F. To recover ID0 of the Device ID in a given record, multiply *Thread_Index* by 2 and add in the ID0_LSb bit.

The above information should be sufficient for you to write routines that can manipulate the PLC Link Database directly using *IBIOS Serial Commands* or *INSTEON Commands*. Be sure that the database doesn't grow too large as you add records, and you might want to 'de-frag' the database to recover space as you delete records.

There are utility routines in the *SALad coreApp Program* that you can use for dealing with the PLC Link Database. Using coreApp routines is far easier than writing your own.

During SALad code development, you can directly read and write records to a PLC Link Database if you are using the SALad Integrated Development Environment (IDE). See the *PLC Database* section of the *SALad Integrated Development Environment User's Guide*.

Non-PLC Link Database

Non-PLC devices, such as the ControLinc™ V2, SwitchLinc™ V2, LampLinc™ V2, or ApplianceLinc™ V2, contain a Link Database whose records are stored sequentially, as opposed to linked-list storage in the PLC. Nevertheless, the data contained in a given record is similar.

The Non-PLC Link Database starts at the top of external (serial) EEPROM and grows downward. In most Non-PLC INSTEON devices, top of memory is 0x0FFF. Each Non-PLC Link Database Record is 8 bytes long, so the first record starts at 0x0FF8, the second record starts at 0x0FF0, and so on.

In the table, the 3-byte INSTEON Address contained in a record is called the *Device ID*.

Non-PLC Link Database Record Format		
Field	**Length (bytes)**	**Description**
Flags	1	Link Control Flags: Bit 7: 1 = Record is in use, 0 = Record is available Bit 6: 1 = Controller (Master) of Device ID, 0 = Responder to (Slave of) Device ID Bit 5: 1 = ACK Required, 0 = No ACK Required (currently always set to 1) Bit 4: Reserved Bit 3: Reserved Bit 2: Reserved Bit 1: 1 = Record has been used before ('high-water' mark) Bit 0: Unused
Group	1	Group Number
ID	3	Device ID
Data 1	1	Link-specific data (e.g. *On-Level*)
Data 2	1	Link-specific data (e.g. Ramp Rates, Setpoints, etc.)
Data 3	1	Unused

INSTEON Extended Messages

Designers are free to devise all kinds of meanings for the User Data that can be exchanged among devices using INSTEON Extended messages. For example, many INSTEON devices include an interpreter for an application language, called SALad, which is compiled into token strings. SALad token string programs can be downloaded into devices (and they can also be debugged) using Extended messages (see *SALad Applications*, below).

For applications that must be secure, such as door locks and security systems, Extended messages can contain encrypted data. See *Encryption within Extended Messages*, below for more information.

If an application needs to transport more than 14 bytes of User Data, then it can use multiple Extended messages. Each Extended message can act as a packet, with the complete User Data reassembled after all packets are received. Bear in mind, however, that INSTEON was not designed for unrestricted data communications—instead, it is optimized for simplicity and reliability. Designers are encouraged to minimize the use of INSTEON to handle large amounts of data.

INSTEON Security

INSTEON network security is maintained at two levels. *Linking Control* ensures that users cannot create links that would allow them to control their neighbors' INSTEON devices, even though those devices may be repeating each other's messages. *Encryption within Extended Messages* permits completely secure communications for applications that require it.

Linking Control

INSTEON enforces Linking Control by requiring that users have *Physical Possession of Devices* in order to create links, and by *Masking Non-linked Network Traffic* when messages are relayed outside the INSTEON network itself.

Physical Possession of Devices

Firmware in INSTEON devices prohibits them from identifying themselves to other devices unless a user physically presses a button on the device. That is why the Command in the network identification Broadcast message is called *SET Button Pressed*. As shown above in the section *Example of an INSTEON Linking Session*, a user has to push buttons on both the Controller device and the Responder device in order to establish a link between them. A Responder will not act on commands from an unlinked Controller.

Linking by sending INSTEON messages requires knowledge of the 3-byte addresses of INSTEON devices. These addresses, unique for each device, are assigned at the factory and displayed on printed labels attached to the device. Users who have physical possession of a device can read the device address from the label and manually enter it when prompted by a computer program.

Masking Non-linked Network Traffic

As described in the section *Interfacing to an INSTEON Network*, above, there can be many kinds of INSTEON devices, called Bridges, that connect an INSTEON network to the outside world. But since an INSTEON Bridge is itself just another INSTEON device, it must be linked to other devices on the INSTEON network in order to exchange messages with them. A user must establish these links in the same way as for any other INSTEON device—by pushing buttons or by typing in addresses.

Smarthome's PowerLinc™ Controller (PLC) is an example of an INSTEON-certified Bridge device that monitors INSTEON traffic and relays it to a computer via a serial link. For security, the PLC's firmware masks the all but the two low-bytes of the *From Address* and *To Address* fields of INSTEON messages unless the traffic is from an INSTEON device already linked to the PLC, or the traffic is from a device that already knows the address of the PLC. In this way, software can take into account the existence of INSTEON traffic without users being able to discover the addresses of devices that they never had physical access to.

To avoid 'spoofing,' where an attacker poses as someone else (by causing the PLC to send messages with bogus From Addresses), the PLC's firmware always inserts the true PLC ID number in the From Address field of messages that it sends.

Encryption within Extended Messages

For applications such as door locks and security systems, INSTEON Extended messages can contain encrypted payloads. Possible encryption methods include rolling-code, managed-key, and public-key algorithms. In keeping with INSTEON's hallmark of simplicity, rolling-code encryption, as used by garage door openers and radio keyfobs for cars, is the method preferred by Smarthome. The encryption method that will be certified as the INSTEON standard is currently under development.

INSTEON BIOS (IBIOS)

The INSTEON Basic Input/Output System (IBIOS) implements the basic functionality of INSTEON devices like Smarthome's PowerLinc™ V2 Controller (PLC). Built on a flat 16-bit address space, this firmware includes an INSTEON Engine that handles low-level INSTEON messaging, PLC event generation, USB or RS232 serial communications, IBIOS Serial Command processing, a software realtime clock, a SALad language interpreter, and various other functions.

The SALad language interpreter runs SALad applications that can stand alone or interface serially with other computing devices. *The Smarthome PowerLinc Controller* comes from the factory with a pre-installed *SALad coreApp Program* that provides onboard event handling, INSTEON device linking, and many other useful functions not supported by IBIOS. But even without a SALad program like coreApp running, IBIOS offers a sophisticated serial interface to the outside world.

This section explains what the INSTEON Engine does, what IBIOS Events occur, how serial communications works, how to use IBIOS Serial Commands directly, and other features of IBIOS. See the *SALad Language Documentation* section for complete information on writing and running SALad programs.

In This Section

IBIOS Flat Memory Model
: Describes how physical memory is mapped into a single 16-bit address space and gives a table of all important memory locations.

IBIOS Events
: Lists the Events that IBIOS generates and explains how they work.

IBIOS Serial Communication Protocol and Settings
: Describes the serial communication protocol, the port settings for an RS232 link, and how to use a USB link.

IBIOS Serial Commands
: Lists the IBIOS Serial Commands, describes what they do, and gives examples of how to use them.

IBIOS INSTEON Engine
: Discusses the functionality of the INSTEON Engine.

IBIOS Software Realtime Clock/Calendar
: Describes how to use the software realtime clock/calendar.

IBIOS X10 Signaling
: Explains usage of the X10 powerline interface.

IBIOS Input/Output
: Gives details on the Pushbutton input and LED flasher.

IBIOS Remote Debugging
: Describes how SALad programs can be remotely debugged using IBIOS.

IBIOS Watchdog Timer
: Describes how the watchdog timer works.

IBIOS Flat Memory Model

All IBIOS memory, no matter what its physical address, is accessed using a flat 16-bit address space. Microcontroller RAM, microcontroller internal (high-speed) EEPROM, external I^2C serial (low-speed) EEPROM, and any other I^2C chips are all mapped into this single 16-bit space.

In This Section

Flat Memory Addressing

The 16-bit flat address space is broken up into two regions. The bottom 32K (0x0000 through 0x7FFF) is a fixed area that always maps to the same physical memory. The top 32K (0x8000 through 0xFFFF) is a variable area that maps to the physical memory of up to four different I^2C chips provided in hardware, under control of the register I2C_Addr.

The bottom 32K fixed area always contains the microcontroller RAM registers in addresses 0x0000 through 0x01FF, and microcontroller internal EEPROM in addresses 0x200 through 0x02FF. The remainder of the bottom 32K, from 0x0300 through 0x7FFF, always maps to the physical memory of the primary external I^2C serial EEPROM chip from 0x0000 through 0x7CFF. (The *primary* external I^2C serial chip has all of its chip-select pins tied low in hardware.)

To choose which of the four possible I^2C chips to map to the top 32K of the flat address space, set the top 7 bits of the I2C_Addr register. The top 4 bits (bits 7 – 4) will match the I^2C address burned into the I^2C chip by the manufacturer. Bits 3 and 2 will match the way the chip-select pins of the I^2C chip are wired in hardware. Set the bit to 0 for a pin that is tied low, or to 1 for a pin that is tied high. The third chip-select pin must always be tied low in hardware so that bit 1 of the I2C_Addr register can be used to indicate whether the I^2C chip uses 8-bit or 16-bit addressing. (A chip with no more than 256 memory locations, such as a realtime clock chip, will normally use 8-bit addressing, but chips with more than 256 memory locations, such as serial EEPROMs, *must* use 16-bit addressing.)

While top 7 bits of the I2C_Addr register uniquely identify which I^2C chip to map into the top 32K of flat memory, the least-significant bit (the LSb, bit 0), selects whether the bottom 32K or the top 32K of this chip's memory will appear in that space. If the I^2C chip in question does not contain more than 32K of memory, it does not actually matter how the I2C_Addr register's LSb is set. But if the I^2C chip does contain more than 32K of memory, then setting the LSb to 0 will access the bottom 32K, and setting the LSb to 1 will access any memory above 32K.

Note that I^2C chips that contain fewer than 32K addresses in power-of-two increments (e.g. 1K, 2K, 4K, 8K, or 16K chips) will have their available addresses repeated multiple times in the top 32K of flat memory. For example, an 8K chip will appear four times in the 32K space, and a 16K chip will appear twice. (Mathematically, if the chip contains 2^N addresses, and N is 15 or less, then the chip's contents will be repeated 2^(15 – N) times in the top 32K of flat memory.)

Overall Flat Memory Map

Address Range	Contents
0x0000⇒0x01FF	Microcontroller RAM Registers
0x0200⇒0x02FF	Microcontroller Hi speed EEPROM
0x0300⇒0x7FFF	Serial I^2C EEPROM physical addresses 0x0000 through 0x7CFF
0x8000⇒0xFFFF	Varies depending on contents of register I2C_Addr I2C_Addr: xxxxxxx0 = I^2C physical addresses 0x0000⇒0x7FFF I2C_Addr: xxxxxxx1 = I^2C physical addresses 0x8000⇒0xFFFF

I2C_Addr Register

I2C_Addr Register	
Bit	**Meaning**
7 (MSb)	Top nibble of I^2C Address burned into I^2C chip by manufacturer.
6	
5	
4	
3	Top 2 of 3 chip-select pins tied high or low in hardware design. There can be a total of 4 I^2C chips.
2	
1	3rd of 3 chip-select pins must be tied low in hardware design, so that IBIOS can use this bit as follows: 0 = I^2C chip uses 8-bit addressing (e.g. realtime clock chip). 1 = I^2C chip uses 16-bit addressing (e.g. serial EEPROM).
0 (LSb)	Defined by I^2C spec as Read/Write bit, but used by IBIOS as follows: 0 = bottom 32K of I^2C chip is mapped to top 32K of flat memory. 1 = top 32K of I^2C chip is mapped to top 32K of flat memory.

Flat Memory Map

Address	Register and Bits		Description
0x0024	NTL_CNT		Count for SALad block mode operations
0x0026	RD_H		Remote Debugging breakpoint address MSB
0x0027	RD_L		Remote Debugging breakpoint address LSB
0x0028	PC_H		SALad Program Counter MSB
0x0029	PC_L		SALad Program Counter LSB
0x002A	DB_H		Database Pointer MSB
0x002B	DB_L		Database Pointer LSB
0x002C	NTL_SP_H		Return Stack Pointer MSB
0x002D	NTL_SP_L		Return Stack Pointer LSB
0x0033	NTL_BUFFER		Pointer to end of Timer Buffer, which begins at 0x0046. This 8-bit pointer defaults to 0x4D to allow room for 4 timers which are 2 bytes each.
0x0034	NTL_RND		Random Number Register
0x0035	NTL_REG_H		High byte of Pointer to R0
0x0036	NTL_REG_L		Low byte of Pointer to R0
0x0037	NTL_EVENT		Event used to invoke SALad
0x0038 ⇒ 0x003F	NTL_EVNT0- NTL_EVNT7		Static Event Queue
0x0040	NTL_TIME_H		Time-of-day alarm (minutes since midnight MSB)
0x0041	NTL_TIME_L		Time-of-day alarm (minutes since midnight LSB)
0x0042	NTL_TICK		Zero Crossing count down tick timer
0x0046 ⇒ NTL_BUFFER	NTL_TIMERS		Timer Buffer; Starts at 0x0046
NTL_BUFFER ⇒ NTL_SP	NTL_REGS		User Register Space
NTL_SP ⇒ 0x006F	NTL_STACK		SALad Return Stack
0x0074	TOKEN		Currently executing SALad instruction token
0x0075	NTL_STAT		SALad Status Register
	_DB_END	4	1=Enrollment database search reached end of database
	_DB_PASS	3	1=Enrollment database search successful
	_NTL_DZ	2	1=Divide by Zero
	_NTL_BO	1	1=Buffer Overrun

Address	Register and Bits		Description		
	_NTL_CY	0	1=Carry from Math and Test operations		
0x0076	NTL_CONTROL		SALad debugging control flags		
	_RD_STEP _RD_HALT	7 6	_RD_HALT	_RD_STEP	
			0	0	Normal execution
			0	1	Animation (Trace)
			1	0	Execution halted
			1	1	Single step requested
	_RD_BREAK	5	0=Range Check Mode, 1=Breakpoint Mode		
0x0142	I_Control		INSTEON result flags		
	_I_DebugRpt	6	1=Enable INSTEON Debug Report		
	_I_SendDirect	5	0=Send INSTEON from working buffer, 1=Send INSTEON direct		
	_I_Transmit	4	1=Request To Send INSTEON		
	_Repeat_On	1	1=Hops enabled		
0x0154	Control		General system control flags		
	_Reset	7	1=request system reset		
	_Watchdog	6	1=request watchdog reset		
	_PDI	2	1=Daughter card interrupt occurred and has been serviced		
	_NoEventRpt	1	1=Inhibit static event report		
	_TAP_LAST	0	last state of push button		
0x0156	TAP_CNT		Counts multiple *SET Button* taps		
0x0157	Tick		Incremented from 0⇒120 every second		
0x0158	RTC_TIME_H		Time since midnight in minutes (MSB, 0-1439)		
0x0159	RTC_TIME_L		Time since midnight in minutes (LSB, 0-1439)		
0x015A	RTC_YEAR		Year (0-99)		
0x015B	RTC_MON		Month (1-12)		
0x015C	RTC_DAY		Day (1-31, month specific)		
0x015D	RTC_DOW		Day-of-Week bitmap (0SSFTWTM)		
0x015E	RTC_HOUR		Hour (0-23)		
0x015F	RTC_MIN		Minute (0-59)		
0x0160	RTC_SEC		Second (0-59)		
0x0164	X10_RX		X10 Receive Buffer		

Address	Register and Bits		Description
0x0165	X10_TX		X10 Transmit Buffer
0x0166	X10_FLAGS		X10 Flags
	_X10_RTS	7	1=Request To Send
	_X10_TXEX	6	1=Start extended transmit after current command (for internal use)
	_X10_EXTENDED	5	1=Extended transfer in progress (Tx or Rx)
	_X10_COMBUF	4	1=Command, 0=Address (for internal use)
	_X10_TXCOMMAND	3	1=Command, 0=Address for transmit
	_X10_RXCOMMAND	2	1=Command, 0=Address for receive
	_X10_VALID	1	1=X10 receive valid
	_X10_ENABLED	0	1=X10 active (for internal use)
0x0168	LED_MODE		Bitmap defines flashing pattern for LED 1=On, 0=Off
0x0169	LED_TMR		Duration of LED flashing in seconds
0x016A	LED_DLY		Period between each flash. Defaults to 5, which is 1/8 second per bit in LED_MODE.
0x016B	RS_CONTROL		Control flags for serial command interpreter
	_RS_ComReset	7	These are used for serial command time limit. This limits how long a command remains active in SALad. This prevents the serial engine from locking you out if SALad receives a corrupt command (non-native serial command). Default time limit is two seconds. To disable, clear bits 5⇒7.
	_RS_ComLimit2	6	
	_RS_ComLimit1	5	
	_RS_AppLock	3	1=Enable overwriting of SALad code from 0x0200 to end of SALad App given in 0x0216 and 0x0217
	_RS_ComDisable	2	1=core command processing disabled
	_RS_ComActive	1	1=command active for SALad processing (Non-native serial command)
	_RS_02	0	1=02 received for command start
0x016F	EventMask		Mask to enable or disable events
	_EM_BtnTap	7	1=enabled
	_EM_BtnHold	6	1=enabled
	_EM_BtnRel	5	1=enabled
	_EM_TickTimer	4	1=enabled
	_EM_Alarm	3	1=enabled

Address	Register and Bits		Description
	_EM_Midnight	2	1=enabled
	_EM_2AM	1	1=enabled
	_EM_RX	0	1=enabled
0x017D	I2C_ADDR		Address of I^2C device; bit 0 controls 0x8000 ⇒ 0xFFFF of flat model, 1=hi region, 0=low region
0x01A0	DB_FLAGS		Database search mode bitmap
0x01A1	DB_0		Database ID_H; (INSTEON construction buffer) From Address_H; ignored for INSTEON message construction
0x01A2	DB_1		Database ID_M; (INSTEON construction buffer) From Address_M; ignored for INSTEON message construction
0x01A3	DB_2		Database ID_L; (INSTEON construction buffer) From Address_L; ignored for INSTEON message construction
0x01A4	DB_3		Database Command 1; (INSTEON construction buffer) To Address_H
0x01A5	DB_4		Database Command 2; (INSTEON construction buffer) To Address_M
0x01A6	DB_5		Database Group Number; (INSTEON construction buffer) To Address_L
0x01A7	DB_6		Database State; (INSTEON construction buffer) Message Flags
0x01A8	DB_7		Message Command 1; (INSTEON construction buffer) Command 1
0x01A9	DB_8		Message Command 2; (INSTEON construction buffer) Command 2
0x01AA	DB_9		
0x01AB	DB_A		
0x01AC	RxFrom0		Receive "From" address high byte
0x01AD	RxFrom1		Receive "From" address middle byte
0x01AE	RxFrom2		Receive "From" address low byte
0x01AF	RxTo0		Receive "To" address high byte
0x01B0	RxTo1		Receive "To" address middle byte
0x01B1	RxTo2		Receive "To" address low byte
0x01B2	RxExtRpt		Receive Control Flags
	_RxBroadcastBit	7	Broadcast Message
	_RxGroup	6	Group Message
	_RxAckBit	5	Acknowledge Message
	_RxExtMsgBit	4	Extended Message

Address	Register and Bits		Description
0x01B3	RxCmd1		Command byte 1
0x01B4	RxCmd2		Command byte 2
0x01B5	RxExtDataD		Short message CRC or extended message Data D
0x01B6	RxExtDataC		Extended message Data C
0x01B7	RxExtDataB		Extended message Data B
0x01B8	RxExtDataA		Extended message Data A
0x01B9	RxExtData9		Extended message Data 9
0x01BA	RxExtData8		Extended message Data 8
0x01BB	RxExtData7		Extended message Data 7
0x01BC	RxExtData6		Extended message Data 6
0x01BD	RxExtData5		Extended message Data 5
0x01BE	RxExtData4		Extended message Data 4
0x01BF	RxExtData3		Extended message Data 3
0x01C0	RxExtData2		Extended message Data 2
0x01C1	RxExtData1		Extended message Data 1
0x01C2	RxExtData0		Extended message Data 0
0x01C3	RxExtCrc		Extended message CRC
0x01C4	TxFrom0		Transmit "From" address high byte
0x01C5	TxFrom1		Transmit "From" address middle byte
0x01C6	TxFrom2		Transmit "From" address low byte
0x01C7	TxTo0		Transmit "To" address high byte
0x01C8	TxTo1		Transmit "To" address middle byte
0x01C9	TxTo2		Transmit "To" address low byte
0x01CA	TxExtRpt		Transmit Control Flags
	_TxBroadcastBit	7	Broadcast
	_TxGroup	6	Group
	_TxAckBit	5	Acknowledge
	_TxExtMsgBit	4	Extended
0x01CB	TxCmd1		Command byte 1
0x01CC	TxCmd2		Command byte 2
0x01CD	TxExtDataD		Short message CRC or extended message Data D
0x01CE	TxExtDataC		Extended message Data C

Address	Register and Bits		Description
0x01CF	TxExtDataB		Extended message Data B
0x01D0	TxExtDataA		Extended message Data A
0x01D1	TxExtData9		Extended message Data 9
0x01D2	TxExtData8		Extended message Data 8
0x01D3	TxExtData7		Extended message Data 7
0x01D4	TxExtData6		Extended message Data 6
0x01D5	TxExtData5		Extended message Data 5
0x01D6	TxExtData4		Extended message Data 4
0x01D7	TxExtData3		Extended message Data 3
0x01D8	TxExtData2		Extended message Data 2
0x01D9	TxExtData1		Extended message Data 1
0x01DA	TxExtData0		Extended message Data 0
0x01DB	TxExtCrc		Extended message CRC
0x01DD	RS_ERROR		RS232 Error register
	_RX_Empty	7	Receive buffer empty
	_Rx_Full	6	Receive buffer full
	_RX_OF	5	Receive buffer overflow
	_RX_Busy	4	Receive busy
	_TX_Empty	3	Transmit buffer empty
	_TX_Full	2	Transmit buffer full
	_TX_OF	1	Transmit buffer overflow
	_TX_Busy	0	Transmit busy
0x01E8 ⇒ 0x01EF	RX_Buffer		RS232 receive buffer
0x01DF	RX_PTR		Points to next open slot in serial receive buffer, if it contains 0xE8, the buffer is empty
0x0200	VALID		= 'P' if EEPROM is valid; 0x0200 is beginning of microcontroller internal EEPROM
0x0201	ID_H		High byte if ID
0x0202	ID_M		Middle byte of ID
0x0203	ID_L		Low byte of ID
0x0204	DEV_TYPE		Device Type
0x0205	SUB_TYPE		Device Sub Type
0x0206	REV		Firmware Revision (MSN=Release, LSN=Ver)

Address	Register and Bits	Description
0x0207	MEM_SIZE	Mask for installed external memory: 00000000=none 00001111=4K 01111111=32K 11111111=64K
0x0210 ⇒ 0x0211	APP_ADDR_TEST	Address of range of application for verification test
0x0212 ⇒ 0x0213	APP_LEN_TEST	Length of range of application for verification test
0x0214 ⇒ 0x0215	APP_CHECK_TEST	Two's complement checksum of range of application for verification test
0x0216 ⇒ 0x0217	APP_END	Top of currently loaded SALad application. EEPROM is write-protected from 0x0200 to the address contained here. Set 0x16B bit 7 to enable over-writing.
0x0230	TIMER_EVENT	Entry point of SALad application for Timer Events
0x0237	STATIC_EVENT	Entry point of SALad application for Static Events
0x0230 ⇒ 0x02FF	SALad application code for fast execution; Microcontroller Internal EEPROM	
0x0300 ⇒ 0x7FFF	Slow SALad application code; External serial EEPROM	
0x8000 ⇒ 0xFFFF	If I2C_Addr bit 0 = **0**, then 0x8000⇒0xFFFF is **low** region (0x0000⇒0x7FFF) of memory in I2C device specified by upper 7 bits of I2C_Addr. If I2C_Addr bit 0 = **1**, then 0x8000⇒0xFFFF is **high** region (0x8000⇒0xFFFF) of memory in I2C device specified by upper 7 bits of I2C_Addr.	

IBIOS Events

IBIOS events are all *Static Events*. If a SALad application is present, all of these Static Events are sent to a SALad event handler, like the one in the *SALad coreApp Program*, which comes pre-installed in *The Smarthome PowerLinc Controller*. In fact, some of these events, such as receiving INSTEON or X10 messages, *require* SALad handling in order to guarantee realtime processing. *Timer Events* also occur, but these must be handled by a SALad application, so they are documented elsewhere (see *SALad Timers*).

Whenever an IBIOS Event occurs (assuming you have not disabled event reporting), IBIOS will notify your PC by sending an *Event Report (0x45)* IBIOS Serial Command. See the *IBIOS Serial Command Summary Table* for more information about event reporting.

Eight events (**0x0A** through **0x11**) can be prevented from occurring by clearing individual bits in the *EventMask* register at 0x016f (see *Flat Memory Map*). Initialization code sets *EventMask* to 0xFF at power up, so all events are enabled by default. Bit 7 (the MSb) of *EventMask* controls event **0x0A**, down through bit 0 (the LSb), which corresponds to event **0x11**.

IBIOS Event Summary Table

This table lists all currently defined Static Event handles.

Handle
Gives the number used by IBIOS to report the Event.

Name
Gives the name of the Event as used in software.

Note
See the item with the same number in *IBIOS Event Details* for more information.

Description
Briefly describes what happened to fire the event.

SALad Static Events			
Handle	**Name**	**Note**	**Description**
0x00	EVNT_INIT	1	SALad initialization code started (automatic at power-up).
0x01	EVNT_IRX_MYMSG	2	Received the first message in a hop sequence addressed to me.
0x02	EVNT_IRX_MSG	2	Received the first message in a hop sequence not addressed to me.
0x03	EVNT_IRX_PKT	2	Received a duplicate message in a hop sequence whether or not addressed to me (*may* occur after an **0x01** or **0x02** event).
0x04	EVNT_ITX_ACK	2	Received expected Acknowledge message after Direct message sent.
0x05	EVNT_ITX_NACK	2	Did not receive expected Acknowledge message after Direct message sent using 5 retries.
0x06	EVNT_IRX_BADID	3	Received a message with an unknown *To Address*. The message was censored by replacing its contents with 0xFFs, except for

SALad Static Events			
Handle	**Name**	**Note**	**Description**
			the *From Address* and *To Address* LSBs).
0x07	EVNT_IRX_ENROLL	4	Received an INSTEON message containing an enrollment-specific INSTEON command. The INSTEON message was received *after* a SALad *Enroll* instruction was executed while the *SET Button* was being held down, and *before* a four-minute timeout expired. The received message's contents are in plaintext (i.e. not censored by masking with 0xFFs).
0x08	EVNT_XRX_MSG	6	Received an X10 byte.
0x09	EVNT_XRX_XMSG	6	Received an X10 Extended Message byte.
0x0A	EVNT_BTN_TAP	7	The *SET Button* was tapped for the first time.
0x0B	EVNT_BTN_HOLD	7	The *SET Button* is being held down.
0x0C	EVNT_BTN_REL	7	The *SET Button* is no longer being tapped or held down. The number of taps is in the *TAP_CNT* register at 0x0156.
0x0D	EVNT_TICK	8	Tick counter has expired (NTL_TICK)
0x0E	EVNT_ALARM	8	Hours and minutes equals current time
0x0F	EVNT_MIDNIGHT	8	Event occurs every midnight
0x10	EVNT_2AM	8	Event occurs every 2:00 am
0x11	EVNT_RX	9	Received a serial byte for SALad processing
0x12	EVNT_SRX_COM	9	Received an unknown IBIOS Serial Command
0x13	EVNT_DAUGHTER	10	Received interrupt from daughter card
0x14	EVNT_LOAD_ON	11	Load turned on
0x15	EVNT_LOAD_OFF	11	Load turned off

IBIOS Event Details

The numbers below refer to the **Note** column in the above *IBIOS Event Summary Table*.

1. **EVNT_INIT (0x00)**

   When power is first applied, IBIOS runs its initialization code, then it checks for the existence of a valid SALad application program. If there is one, IBIOS fires this event and then starts SALad with this event in the event queue.

   You can force a power-on reset in software by setting bit 7 (*_Reset*) of the *Control* register at 0x0154 to one (see *Flat Memory Map*).

2. **EVNT_IRX_MYMSG (0x01), EVNT_IRX_MSG (0x02), EVNT_IRX_PKT (0x03)**

When IBIOS first receives a new INSTEON message that has not been seen before, the message may be arriving on its first hop, or it may be arriving on a subsequent hop, depending on the INSTEON environment. If the *To Address* of this new message matches the 3-byte IBIOS ID burned in at the factory, then the new message is *to me*, and an **EVNT_IRX_MYMSG (0x01)** event will fire. If the new message is *not to me*, then an **EVNT_IRX_MSG (0x02)** event will fire instead. If the new message was received before its last hop, then the same message may be received again on subsequent hops. Whenever this happens an **EVNT_IRX_PKT (0x03)** event will fire whether the duplicate message is *to me* or *not to me*.

After any of these events fire, IBIOS will use a SALad application like the *SALad coreApp Program* to send an *INSTEON Received (0x4F)* IBIOS Serial Command (see *IBIOS Serial Command Summary Table*).

3. **EVNT_IRX_ACK (0x04), EVNT_IRX_NACK (0x05)**

   After IBIOS sends a Direct (Point-to-Point) INSTEON message, it expects the addressee to respond with an INSTEON Acknowledge message. If IBIOS receives the Acknowledge message as expected, it fires an **EVNT_IRX_ACK (0x04)** event. On the other hand, if the expected Acknowledge message is not received, IBIOS will automatically retry sending the Direct message again. If after five retries the addressee still does not respond with an Acknowledgement message, IBIOS will fire an **EVNT_IRX_NACK (0x05)** event. One or the other (but not both) of these events is guaranteed to fire after sending a Direct message, although the **EVNT_IRX_NACK (0x05)** event may not fire for a long time due to the retries.

   After either of these events fires, IBIOS will use a SALad application like the *SALad coreApp Program* to send an *INSTEON Received (0x4F)* IBIOS Serial Command (see *IBIOS Serial Command Summary Table*).

4. **EVNT_IRX_BADID (0x06)**

   If the *To Address* of an incoming INSTEON message does not match the 3-byte IBIOS ID burned in at the factory (i.e. the message is *not to me*), and the *To Address* does not match any of the IDs in IBIOS's *INSTEON Link Database*, then for security reasons, the message is censored. A censored INSTEON message will have all of its data bytes replaced with 0xFFs, except for the low bytes of the *From Address* and the *To Address*. See *Masking Non-linked Network Traffic* for more information on *INSTEON Security*.

   After this event fires, IBIOS will use a SALad application like the *SALad coreApp Program* to send an *INSTEON Received (0x4F)* IBIOS Serial Command (see *IBIOS Serial Command Summary Table*).

5. **EVNT_IRX_ENROLL (0x07)**

   This event supports *INSTEON Device Linking*, also known as *enrollment*, that is being handled by a suitable SALad application. The event may only fire **after** a SALad *Enroll* instruction (see SALad Instruction Summary) executes during the time that the *SET Button* is held down, and **before** a four-minute timer has expired. If during that time IBIOS sends or receives an INSTEON Broadcast message containing an INSTEON *SET Button Pressed Slave (0x01)* or *SET Button*

Pressed Master (0x03) command or an INSTEON Direct message containing an INSTEON *Assign to Group (0x01)* or *Delete from Group (0x02)* command (see *INSTEON Command Tables*), the event will fire. The received message that triggered the event will *not* be censored by replacing its contents with 0xFFs, so that the SALad program can enroll the INSTEON device that sent the message in the *INSTEON Link Database*.

Requiring that the *SET Button* be pushed enforces *INSTEON Security* by requiring *Physical Possession of Devices*.

After this event fires, IBIOS will send an *INSTEON Received (0x4F)* IBIOS Serial Command (see *IBIOS Serial Command Summary Table*).

6. **EVNT_XRX_MSG (0x08), EVNT_XRX_XMSG (0x09)**

   These events occur when IBIOS receives an X10 byte over the powerline. When IBIOS receives a new X10 byte, it fires an **EVNT_XRX_MSG (0x08)** event. If IBIOS determines that subsequent received X10 bytes are part of an Extended X10 message, then it will fire an **EVNT_XRX_XMSG (0x09)** event for those bytes. After IBIOS detects the end of the Extended X10 message (three 0x00 bytes in succession), or else if a timeout expires, IBIOS will revert to firing an **EVNT_XRX_MSG (0x08)** event for the next X10 byte it receives.

   After either of these events occurs, IBIOS will use a SALad application like the *SALad coreApp Program* to send an *X10 Received (0x4A)* IBIOS Serial Command (see *IBIOS Serial Command Summary Table*).

7. **EVNT_BTN_TAP (0x0A), EVNT_BTN_HOLD (0x0B), EVNT_BTN_REL (0x0C)**

   These events fire when the *SET Button* is pushed in various ways. A *Button Tap* occurs when the user pushes the *SET Button* and then lets up less than 350 milliseconds (350 ms) later. A *Button Hold* occurs when the user pushes the *SET Button* and then lets up more than 350 ms later.

   The first time IBIOS detects a Button Tap, it fires an **EVNT_BTN_TAP (0x0A)** event. If there are more Button Taps following the first one, with less than 350 ms between each one, then IBIOS does not fire an event, but it does count the number of Button Taps. When more than 350 ms elapses after a Button Tap, IBIOS fires an **EVNT_BTN_REL (0x0C)** event to indicate the *SET Button* has been released. At that time, you can inspect the *TAP_CNT* register at 0x0156 (see *Flat Memory Map*) to see how many Button Taps occurred before the release.

   Whenever IBIOS detects a Button Hold it fires an **EVNT_BTN_HOLD (0x0B)** event, then when it detects that the button has been released it fires an **EVNT_BTN_REL (0x0C)** event.

   Note that a Button Hold can follow some number of Button Taps, in which case events **EVNT_BTN_TAP (0x0A)**, **EVNT_BTN_HOLD (0x0B)**, and **EVNT_BTN_REL (0x0C)** would occur in that order. Inspect *TAP_CNT* after the **EVNT_BTN_REL (0x0C)** event to see how many Button Taps there were.

8. **EVNT_TICK (0x0D), EVNT_ALARM (0x0E), EVNT_MIDNIGHT (0x0F), EVNT_2AM (0x10)**

   These events depend on the *IBIOS Software Realtime Clock* (RTC), which counts powerline zero crossings (120 per second) for timing. The Software RTC must be set by a SALad program or by IBIOS Serial Commands upon power up for **EVNT_ALARM (0x0E), EVNT_MIDNIGHT (0x0F), EVNT_2AM (0x10)** to work properly.

   You can use **EVNT_TICK (0x0D)** to measure short time intervals, ranging from 8.333 milliseconds (ms) up to 2.125 seconds. Each tick is one powerline zero crossing, which is 1/120 second, or 8.333 ms. To cause this event to fire, load a value from 1 to 255 into the *NTL_TICK* register at 0x0042 (see *Flat Memory Map*). After that number of ticks, IBIOS will fire this event one time only. You can disable this event at any time by loading 0x00 into *NTL_TICK*.

   Registers *RTC _TIME_H* and *RTC _TIME_L* at 0x0158 and 0x0159 contain a 16-bit value ranging from 0 to 1439 that counts the number of minutes that has elapsed since midnight. You can cause an **EVNT_ALARM (0x0E)** event to fire by loading a valid minutes-from-midnight value into the *NTL_TIME_H* and *NTL _TIME_H* registers at 0x0040 and 0x0041. When the value in *RTC_TIME_H,L* matches your value in *NTL_TIME_H,L*, IBIOS will fire **EVNT_ALARM (0x0E)**. The value you loaded into *NTL_TIME_H,L* is not altered when the event fires, so the event will fire every day. However, if you load an invalid value (greater than 1439) into *NTL_TIME_H,L*, then the event will never fire.

   The **EVNT_MIDNIGHT (0x0F)** and **EVNT_2AM (0x10)** events fire at midnight and at 2:00 am, as you would expect.

9. **EVNT_RX (0x11), EVNT_SRX_COM (0x12)**

   These events allow the number of *IBIOS Serial Commands* to be extended. All IBIOS Serial Commands begin with 0x02, followed by the number of the command. If IBIOS receives a Serial Command number outside the range 0x40 through 0x48 (see *IBIOS Serial Command Summary Table*), it will fire the **EVNT_SRX_COM (0x12)** event, then start a two-second timer. Thereafter, every time IBIOS receives a serial byte, it will fire the **EVNT_RX_COM (0x11)** event and restart the two-second timer. If the two-second timer expires, IBIOS will send a serial NAK (ASCII 0x15) and stop firing **EVNT_RX_COM (0x11)** events.

   When you are finished parsing the incoming IBIOS Serial Command, clear the two-second timer yourself by clearing bits 5 and 6, *_RSComLimit1* and *_RSComLimit*, in the *RS_CONTROL* register at 0x016B (see *Flat Memory Map*). If you want more time, you can get another two seconds by setting the same two bits to one.

10. **EVNT_DAUGHTER (0x13)**

    IBIOS fires this event when it detects that port pin RB6 on the microprocessor went low. Normally this is caused by an attached daughter card requesting an interrupt.

11. **EVNT_LOAD_ON (0x14), EVNT_LOAD_OFF (0x15)**

    These events are for monitoring electrical loads connected to the INSTEON device. They will fire in response to the state of current-sensing hardware, and so are implementation-specific. Contact the INSTEON Developer's Forum or email sdk@insteon.net for more information.

IBIOS Serial Communication Protocol and Settings

In This Section

IBIOS Serial Communication Protocol

All IBIOS Serial Commands start with ASCII 0x02 (STX, Start-of-Text) followed by the Serial Command number (see *IBIOS Serial Commands*). What data follows the command depends on the command syntax (see *IBIOS Serial Command Summary Table*).

IBIOS will respond with an echo of the command number followed by any data that the command returns.

If IBIOS is responding to a Serial Command that it received, it will terminate its response with ASCII 0x06 (ACK, Acknowledge), or sometimes ASCII 0x15 (NAK, Negative Acknowledge).

(**S:** and **R:** denote serial data you **Send to** or **Receive from** IBIOS, respectively.)

S:	0x02 <command number> <parameters>
R:	0x02 <command number> <any returned data> 0x06

If IBIOS is not ready, it will return ASCII 0x15 (NAK).

S:	0x02 <command number> <parameters>
R:	0x15 (NAK)

If you receive 0x15 (NAK), resend your Serial Command.

IBIOS RS232 Port Settings

To communicate to an RS232 PLC, set your PC's COM port as follows:

Setting	Value
Baud	4800
Data Bits	8
Parity	N
Stop Bits	1
Hardware Flow Control	None
Software Flow Control	None

IBIOS USB Serial Interface

The interface to a USB PLC is a simple USB wrapper around the *IBIOS Serial Communication Protocol*, implemented using the Human Interface Device (HID) interface. See, for example, http://www.lvr.com/hidpage.htm for more information on using HID to implement a USB interface.

Smarthome's VendorID and ProductID are listed at http://www.linux-usb.org/usb.ids. The ProductID for the USB PowerLinc™ V2 Controller is 0x0004.

When communicating to a USB PLC, send the same commands as given in *IBIOS Serial Commands*, except use 8-byte USB packets.

The PLC will set the most significant bit of the *Count* byte to indicate *Clear-to-Send*, i.e. that the PLC is ready to receive more data.

For example, to send the IBIOS Serial Command 0x48 (Get PLC Version), send the following 8-byte packet (data bytes are **bold**):

HID USB 8-byte Packet							
Count	**Data**						
0x02	**0x02**	**0x48**	0x00	0x00	0x00	0x00	0x00

The PLC will reply with data similar to the following (data bytes are **bold**; 0x80 in the count indicates that the PLC is ready to receive more data):

HID USB 8-byte Packet							
Count	**Data**						
0x80	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x01	**0x02**	0x00	0x00	0x00	0x00	0x00	0x00
0x03	**0x48**	**0xff**	**0xff**	0x00	0x00	0x00	0x00
0x03	**0xff**	**0x04**	**0x00**	0x00	0x00	0x00	0x00
0x02	**0x23**	**0x06**	0x00	0x00	0x00	0x00	0x00

IBIOS Serial Commands

The IBIOS Serial Command set is the basic interface between a computing device such as a PC or dedicated home controller and a serially connected INSTEON Bridge device such as *The Smarthome PowerLinc Controller* (PLC). For example, a PC connected to a PLC could use IBIOS Serial Commands to send and receive INSTEON or X10 messages directly, or it could download and debug a SALad program that runs on the PLC.

IBIOS Serial Commands also allow indirect access, via INSTEON messages, to other INSTEON devices on the network. For instance, you could upgrade the capabilities of SALad-enabled INSTEON devices by remotely installing and debugging new SALad applications in them.

Some of the IBIOS Serial Commands require a SALad application such as the *SALad coreApp Program* to be running in order to ensure realtime execution.

In This Section

IBIOS Serial Command Table
: Describes all of the IBIOS Serial Commands in an *IBIOS Serial Command Summary Table* and gives *IBIOS Serial Command Details*.

IBIOS Serial Command Examples
: Gives examples of how to use the IBIOS Serial Commands.

IBIOS Serial Command Table

IBIOS Serial Command Parameters

This is what the common parameters shown in the *Format* column of the *IBIOS Serial Command Summary Table* mean. Parameters not listed here should be understood from the context.

MSB means Most-Significant Byte, *LSB* means Least-Significant Byte, *MSb* means Most-Significant bit.

Parameter	Description
Address High (Low)	MSB (LSB) of a 16-bit address
Length High (Low)	MSB (LSB) of a 16-bit length of a data block
Checksum High (Low)	MSB (LSB) of a 16-bit value calculated by summing all the bytes in a data block specified in an IBIOS Serial Command, then taking the two's complement
Data	Block of data bytes
Event Handle	8-bit number indicating the event that fired (see *IBIOS Events*)
Timer Value	8-bit time value: 1 to 127 seconds, if the MSb (bit 7) is 0 (clear) 1 to 127 minutes, if the MSb (bit 7) is 1 (set)

IBIOS Serial Command Summary Table

This table lists all of the IBIOS Serial Commands supported by the PLC.

Code
Gives the hexadecimal number of the IBIOS Serial Command. **SALad** means that the command is only applicable if a suitable SALad application, such as the *SALad coreApp Program*, is running.

Command
Gives the name of the IBIOS Serial Command.

Note
See the item with the same number in *IBIOS Serial Command Details* for more information.

Format
Gives the syntax of the IBIOS Serial Command, including any parameters.

S: and **R:** denote serial data you **Send to** or **Receive from** IBIOS, respectively. See *IBIOS Serial Communication Protocol* for more information.

All IBIOS Serial Commands start with ASCII 0x02 (STX, Start-of-Text) followed by the Serial Command number. See *IBIOS Serial Command Parameters*, above, for the meaning of the command parameters.

All fields in this table contain only one byte, except for those with '…' (e.g. <Data…>, or <Message…>), which contain a variable number of bytes.

IBIOS Serial Commands			
Code	**Command**	**Note**	**Format**
0x40	**Download** (IBIOS receives data from you)	1	**S:** 0x02 **0x40** <Address High> <Address Low> <Length High> <Length Low> <Checksum High> <Checksum Low> <Data…> **R:** 0x02 **0x40** <Address High> <Address Low> <Length High> <Length Low> <0x06 (ACK) | 0x15 (NAK)>
0x41 **SALad**	**Fixed-length Message**	2	**R:** 0x02 **0x41** <Length> <Message…> **NOTE:** <Message…> can contain unrestricted data, such as embedded INSTEON or X10 messages.
0x42	**Upload** (IBIOS sends data to you)	1	**S:** 0x02 **0x42** <Address High> <Address Low> <Length High> <Length Low> **R:** 0x02 **0x42** <Address High> <Address Low> <Length High> <Length Low> <Data…> <Checksum High> <Checksum Low> 0x06
0x43 **SALad**	**Variable-length Text Message**	2	**R:** 0x02 **0x43** <Message…> 0x03 (ASCII ETX, End-of-Text) **NOTE:** <Message…> must not contain 0x03 (ETX, End-of-Text).
0x44	**Get Checksum**	3	**S:** 0x02 **0x44** <Address High> <Address Low> <Length High> <Length Low> **R:** 0x02 **0x44** <Address High> <Address Low> <Length High> <Length Low> <Checksum High> <Checksum Low> 0x06
0x45	**Event Report**	4	**R:** 0x02 **0x45** <Event Handle> can be disabled

IBIOS Serial Commands			
Code	**Command**	**Note**	**Format**
0x46	**Mask**	5	**S:** 0x02 **0x46** <Address High> <Address Low> <OR mask> <AND mask> **R:** 0x02 **0x46** <Address High> <Address Low> <OR mask> <AND mask> 0x06
0x47	**Simulated Event**	6	**S:** 0x02 **0x47** <Event Handle> <Timer Value> **R:** 0x02 **0x47** <Event Handle> <Timer Value> 0x06
0x48	**Get Version**	7	**S:** 0x02 **0x48** **R:** 0x02 **0x48** <INSTEON Address High> <INSTEON Address Middle> <INSTEON Address Low> <Device Type High> <Device Type Low> <Firmware Revision> 0x06
0x49 **SALad**	**Debug Report**	8	**R:** 0x02 **0x49** <Next SALad instruction to execute Address High> <Next SALad instruction to execute Address Low>
0x4A **SALad**	**X10 Byte Received**	9	**R:** 0x02 **0x4A** <0x00 (X10 Address) | 0x01 (X10 Command)> <X10 Byte>
0x4F **SALad**	**INSTEON Message Received**	10	**R:** 0x02 **0x4F** <Event Handle (0x01-0x07)> <INSTEON message (9 or 23 bytes)>

IBIOS Serial Command Details

The numbers below refer to the **Note** column in the above *IBIOS Serial Command Summary Table*.

1. **Download (0x40), Upload (0x42)**

   Use these commands to write **Download (0x40)** or read **Upload (0x42)** IBIOS's memory. The *Flat Memory Map* lists all of the memory locations that you can access, and defines what the contents are.

   You can neither read nor write the firmware in IBIOS's EPROM. The microprocessor's program counter (appearing at locations 0x0002, 0x0082, 0x0102, and 0x0182) is write-protected, as is the Enrollment Timer at 0x016D. No harm will occur if you attempt to read or write address locations where there is no memory, but the results will be indeterminate.

   See item 3, **Get Checksum (0x44)**, for IBIOS's method of calculating checksums (two's complement of a 16-bit sum).

   After downloading, even if you receive 0x06 (ACK), you should immediately issue a **Get Checksum (0x44)** Serial Command to verify that IBIOS wrote the data to memory correctly.

   If you are setting or clearing individual flag bits in a register, use the **Mask (0x46)** command to avoid affecting flags you are not changing.

2. **Fixed-length Message (0x41), Variable-length Text Message (0x43)**

   These Serial Commands require a SALad application such as the *SALad coreApp Program* to be running. A SALad program can send up to 255 bytes of unrestricted (ASCII or binary) data to the host using a **Fixed-length Message (0x41)** Serial Command containing a length byte.

   To send simple text messages without a length restriction, a SALad program can use the **Variable-length Text Message (0x43)** Serial Command. This command does not use a length byte. Instead, it uses an ASCII 0x03 (ETX, End-of-Text) byte to delimit the end of the ASCII text message, so the text message must not contain an embedded ETX character before the actual end of the message.

3. **Get Checksum (0x44)**

   IBIOS calculates checksums by summing up all of the bytes in the given range into a 16-bit register, then taking a two's complement of the 16-bit sum. You can take a two's complement by inverting all 16 bits and then incrementing by one, or else you can just subtract the 16-bit value from 0x0000.

4. **Event Report (0x45)**

   IBIOS sends this Serial Command whenever one of the *IBIOS Events* given in the *IBIOS Event Summary Table* fires.

   You can prevent IBIOS from sending Event Reports by setting bit 1, *_NoEventRpt*, in the *Control* register at 0x0154 to one (see *Flat Memory Map*). IBIOS clears this flag to zero at power up, so Event Reports are enabled by default.

5. **Mask (0x46)**

   Use this Serial Command to set or clear individual flag bits in a register without affecting flags you are not changing.

   To **set** one or more bits in a register to **one**, set the corresponding bits in the OR Mask to one. Bits set to zero in the OR Mask will not change the corresponding bits in the register.

   To **clear** one or more bits in a register to **zero**, set the corresponding bits in the AND Mask to zero. Bits set to one in the AND Mask will not change the corresponding bits in the register.

6. **Simulated Event (0x47)**

   You can force IBIOS to fire one of the Static Events in the *IBIOS Event Summary Table* with this Serial Command by sending the desired <Event Handle> number with a <Timer Value> of zero. You cannot use this Serial Command to fire an **EVNT_INIT (0x00)** initialization event, but there is an alternate method to force a power-on reset (see *IBIOS Event Details*, Note 1).

   IBIOS will fire Timer Events, but unless you are running a SALad program with Timer Event handling code, nothing will happen. See *SALad Timers* for an explanation of SALad Timer events. To simulate a Timer Event, set <Event

Handle> to the Timer Index number of the SALad handler for the timer, and set <Timer Value> to a non-zero value denoting how much time should elapse before the Timer Event fires. If the high bit of <Timer Value> is 0, then the time will be 1 to 127 seconds; if the high bit is 1, then the time will be 1 to 127 minutes.

7. **Get Version (0x48)**

   This Serial Command retrieves the 3-byte INSTEON ID number (see *Device Addresses*), 2-byte *Device Type*, and 1-byte *Firmware Revision* burned in at the factory. This information is read-only.

8. **Debug Report (0x49)**

   You can remotely debug a SALad program with this Serial Command. This is the underlying mechanism used by the IDE debugger described in the *SALad Integrated Development Environment User's Guide*.

   See the *IBIOS Remote Debugging* section for details on how IBIOS remote debugging works. When remote debugging is enabled, IBIOS will use this Serial Command to report the location of the next SALad instruction to be executed.

9. **X10 Byte Received (0x4A)**

   This Serial Command requires a SALad application such as the *SALad coreApp Program* to be running. After IBIOS fires an **EVNT_XRX_MSG (0x08)** or **EVNT_XRX_XMSG (0x09)** (see *IBIOS Event Details*), The SALad app will report the received X10 byte by sending this Serial Command.

   The byte following the **0x4A** command number tells whether the received X10 byte is an X10 Address (the byte is 0x00) or X10 Command (the byte is 0x01). If you are receiving an X10 Extended Message, then this byte is irrelevant.

   See *IBIOS X10 Signaling* for more information.

10. **INSTEON Message Received (0x4F)**

    This Serial Command requires a SALad application such as the *SALad coreApp Program* to be running. After IBIOS fires any of the events **0x01** through **0x07** (see *IBIOS Event Details*), The SALad app will report the INSTEON message received by sending this Serial Command.

    To determine if the INSTEON message's length is 9 bytes (Standard) or 23 bytes (Extended), inspect the message's *Extended Message Flag*.

IBIOS Serial Command Examples

This section contains examples showing how to use various IBIOS Serial Commands described in the *IBIOS Serial Command Table* above. The examples assume you are serially connected to *The Smarthome PowerLinc Controller* (PLC) running the default *SALad coreApp Program*.

In This Section

Get Version

Summary

In this example we will use the **Get Version (0x48)** IBIOS Serial Command to get the PLC's 3-byte INSTEON ID number (see *Device Addresses*), 2-byte *Device Type*, and 1-byte *Firmware Revision* burned in at the factory.

Procedure

1. Send the **Get Version (0x48)** IBIOS Serial Command from the *IBIOS Serial Command Summary Table*:

   `0x02` **`0x48`**.

2. The response should be:

   `0x02` **`0x48`** `<INSTEON Address (3 bytes)> <Device Type (2 bytes> <Firmware Revision (1 byte)> 0x06`.

Read and Write Memory

Summary

In this example we will use the **Upload (0x42)** and **Download (0x40)** IBIOS Serial Commands to alter data stored in the PLC's serial EEPROM chip. See the *Flat Memory Map* for memory address locations. First we will read 4 bytes of data starting at the beginning of external EEPROM at address 0x0300, then we will write **0x55 0xAA 0x55 0xAA** to those locations, then read the data back out to observe our changes.

Procedure

1. Use the **Upload (0x42)** IBIOS Serial Command from the *IBIOS Serial Command Summary Table* to read 0x0004 bytes starting at 0x0300:

   0x02 **0x42** 0x03 0x00 0x00 0x04.

   You can check the response to see what the four bytes are. See the *IBIOS Serial Command Summary Table* for the response syntax.

2. Now use **Download (0x40)** to write **0x55 0xAA 0x55 0xAA** into those locations:

   0x02 **0x40** 0x03 0x00 0x00 0x04 0xFD 0xFB **0x55 0xAA 0x55 0xAA**.

   It is not mandatory that you include a valid checksum (here 0xFDFB), but it is strongly recommended, because you can then use a **Get Checksum (0x44)** IBIOS Serial Command to be sure the data was properly written, without having to read all of the data back.

   The response should be:

   0x02 **0x40** 0x03 0x00 0x00 0x04 0x06.

   Note that the response merely echoes the first 6 bytes of the received command, followed by an ASCII 0x06 (ACK) indicating that the command was properly received. The ACK does *not* indicate that the command executed properly.

3. Now read out the four bytes at 0x0300 again as in Step 1:

   0x02 **0x42** 0x03 0x00 0x00 0x04.

   Check that the response contains the **0x55 0xAA 0x55 0xAA** data. For further validation, you can also verify that the checksum in the response matches a checksum that you compute over the received data.

Get Checksum on Region of Memory

Summary

This example will demonstrate getting the checksum over the first 10 bytes of SALad application code. The checksum is computed as the two's complement of a 16-bit sum of the bytes. You can take a two's complement by inverting all 16 bits in the sum and then incrementing by one, or else you can just subtract the 16-bit value from 0x0000.

For this example the beginning of the SALad application starts at 0x0230 and is 0x0a bytes long.

Procedure

1. Use the **Get Checksum (0x44)** IBIOS Serial Command from the *IBIOS Serial Command Summary Table* to get the checksum on the region starting at 0x0230 and extending 0x000a bytes:

   0x02 **0x44** 0x02 0x30 0x00 0xa0.

2. Retrieve the checksum from the return message, which should be:

   0x02 **0x44** 0x02 0x30 0x00 0xA0 <checksum MSB> <checksum LSB> 0x06.

Send INSTEON

Summary

Before sending INSTEON messages you should familiarize yourself with *INSTEON Messages* and *INSTEON Commands*.

In this example we will send the INSTEON **ON** command with **Level 0xFF** (full on) from a PLC to a LampLinc™ V2 Dimmer that has an INSTEON Address of 0x0002AC. We will first copy the INSTEON message to an area of PLC memory used as a message construction buffer. Then we will set the Request-to-Send INSTEON flag, causing the PLC to send the INSTEON message in its construction buffer to the LampLinc.

Note that for security reasons, IBIOS will always insert its own address (the 3-byte IBIOS ID burned in at the factory) in the *From Address* field no matter what you write to the construction buffer. See *Masking Non-linked Network Traffic* for more information on *INSTEON Security*. Therefore, we do not need to put the *From Address* in the INSTEON message.

Procedure

1. Use the **Download (0x40)** IBIOS Serial Command from the *IBIOS Serial Command Summary Table* to load this 6-byte INSTEON message starting with the *To Address* field

   0x00 0x02 0xAC 0x0F 0x11 0xFF

   into the PLC's C*onstruction Buffer* starting at the *To Address* field at location 0x1A4 (see *Flat Memory Map*).

   The fully formed **Download (0x40)** IBIOS Serial Command, with the embedded INSTEON message in **bold**, is:

   0x02 0x40 0x01 0xA4 0x00 0x06 0xFD 0x88 **0x00 0x02 0xAC 0x0F 0x11 0xFF**.

   The returned serial message should be an echo of the first 6 bytes of the Serial Command followed by an ASCII 0x06 (ACK), like this:

   0x02 0x40 0x01 0xA4 0x00 0x06 0x06.

2. Now use the **Mask (0x46)** IBIOS Serial Command to set the *_I_Transmit* Request-to-Send INSTEON flag (bit 4) in the *I_Control* register at 0x142 (see *Flat Memory Map*), by sending:

   0x02 **0x46** 0x01 0x42 0x10 0xFF.

   The returned serial message should be:

   0x02 **0x46** 0x01 0x42 0x10 0xFF 0x06.

Receive INSTEON

Summary

Before receiving INSTEON messages you should familiarize yourself with *INSTEON Messages* and *INSTEON Commands*.

Here we assume you are using a *The Smarthome PowerLinc Controller* (PLC) running the default *SALad coreApp Program*.

When the PLC receives an INSTEON message, IBIOS fires an IBIOS Event that a SALad program handles. PLCs come from the factory with an open-source *SALad coreApp Program* installed, and you can create your own custom applications by modifying coreApp.

When an INSTEON message arrives, coreApp's event handlers send an **INSTEON Received (0x4F)** IBIOS Serial Command containing the IBIOS Event number and the INSTEON message to your PC. Your PC can then deal with the **INSTEON Received (0x4F)** IBIOS Serial Command whenever it shows up in its serial buffer.

Procedure

1. When an INSTEON message is received, coreApp (and all applications built upon it) will send the message data to your PC using the following format:

   ```
   0x02 0x4F <Event Handle> <INSTEON message>
   ```

2. The *Event Handle* byte tells which of the *IBIOS Events* (0x01 through 0x07) IBIOS fired to trigger the SALad coreApp program. See the *IBIOS Event Summary Table* and *IBIOS Event Details* for more information about what kinds of INSTEON message fire which events.
3. To determine if the length of the *INSTEON message* is 9 bytes (Standard) or 23 bytes (Extended), inspect the message's *Extended Message Flag* (bit 4 of the message's 7th byte). The *INSTEON Message Summary Table* shows all possible INSTEON message types.
4. To determine the meaning of the INSTEON message, look at the *INSTEON Command 1 and 2* fields in the message. The *INSTEON Command Tables* enumerate all of the possible *INSTEON Commands*.

Send X10

Summary

In this example we will send X10 **A1/AON** over the powerline using a PLC and IBIOS Serial Commands. First we will download the **A1** X10 address into the X10 transmit buffer. Then we will set the Request-to-Send X10 bit and clear the Command/Address bit of the X10 Flags register with a **Mask (0x46)** IBIOS Serial Command. Then we will download the **AON** X10 command into the X10 transmit buffer, followed by setting both the Request-to-Send X10 and the Command/Address bits with another **Mask (0x46)** command.

See *IBIOS X10 Signaling* for more information.

Procedure

1. Use the **Download (0x40)** IBIOS Serial Command from the *IBIOS Serial Command Summary Table* to load an X10 **A1** address (**0x66**) into the X10 transmit buffer *X10_TX* at 0x0165 (see *Flat Memory Map*) by sending:

   `0x02 **0x40** 0x01 0x65 0x00 0x01 0xFF 0x33 **0x66**,`

   then check for the ASCII 0x06 (ACK) at the end of the echoed response:

   `0x02 **0x40** 0x01 0x65 0x00 0x01 0x06.`

2. Use the **Mask (0x46)** IBIOS Serial Command to **set** the *_X10_RTS* flag (bit 7) and **clear** the *_X10_TXCOMMAND* flag (bit 3) in the *X10_FLAGS* register at 0x0166 (see *Flat Memory Map*) via:

   `0x02 **0x46** 0x01 0x66 0x80 0xF7,`

   then check for the ASCII 0x06 (ACK) at the end of the echoed response:

   `0x02 **0x46** 0x01 0x66 **0x80 0xF7** 0x06.`

3. The PLC will now send an **A1** X10 address over the powerline.

4. As in Step 1, load an X10 **AON** command (**0x62**) into the X10 transmit buffer by sending:

   `0x02 **0x40** 0x01 0x65 0x00 0x01 0xff 0x37 **0x62**,`

   and check for an appropriate response:

   `0x02 0x40 0x01 0x65 0x00 0x01 0x06.`

5. As in Step 2, use the Mask command to **set** the _X10_RTS bit and **set** the _X10_TXCOMMAND bit via:

   `0x02 0x46 0x01 0x66 **0x88 0xff**`

   then check for an appropriate response:

   `0x02 0x46 0x01 0x66 0x88 0xFF 0x06.`

6. The PLC will now send an **AON** X10 command over the powerline.

Simulated Event

Summary

You can use the **Simulated Event (0x47)** IBIOS Serial Command from the *IBIOS Serial Command Summary Table* to cause the PLC to run its SALad application with the specified event or timer handle in the event queue. See *SALad Event Handling* for more information on the event processing system that SALad programs use.

This example lists a demo SALad application that you can install in the PLC using the tools documented in the *SALad Integrated Development Environment User's Guide*.

Whenever when the PLC's *SET Button* is tapped, the **EVNT_BTN_TAP (0x0A)** IBIOS Event fires (see *IBIOS Event Summary Table*). The demo application's event handler uses a **Variable-length Text Message (0x43)** IBIOS Serial Command to send a 'Button tap detected' ASCII message over the serial connection when this event fires.

You can use this same method for testing other event processing code in your SALad applications.

To demonstrate the **Simulated Event (0x47)** IBIOS Serial Command we will send a simulated **EVNT_BTN_TAP (0x0A)** and observe the 'Button tap detected' message.

Procedure

1. Using the SALad IDE, download the following SALad application into the PowerLinc V2 Controller (iPLC_Map.sal has the definitions and equates for the *Flat Memory Map*, and Event.sal has equates for the *IBIOS Events*):

```
INCLUDE "iPLC_Map.sal"
INCLUDE "Event.sal"

; API Macro Definitions
DEFINE API DATA 0x04
DEFINE SendString API 0x86   ; send a null terminated string
DEFINE SendByte API 0x44

; application header
ORG 0x210
  DATA 0x02 0x10    ; start at 0x0210
  DATA 0x00 0x01    ; length 1
  DATA 0x00 0x00    ; checksum 0 (no verification)
; entry point for timers
ORG 0x230
  END   ; just exit if timer

; entry point for static events
ORG 0x237
  COMP= #EVNT_INIT, NTL_EVENT, ButtonEvent  ; if initialization
  MOVE$ #0x00, NTL_TIMERS, 0x2D             ; clear out NTL_TIMERS
ButtonEvent
  COMP= #EVNT_BTN_TAP, NTL_EVENT, Exit      ; check for button tap
  SendString strButtonMessage
Exit
  END
strButtonMessage
  DATA "Button tap detected", 0x0d, 0x0a, 0x00
```

2. Tap the *SET Button* on the PLC and you should see the message 'Button tap detected' displayed in the IDE's *Comm Window – ASCII Window*.

3. Now send the **Simulated Event** serial command from the *IBIOS Serial Command Summary Table* to fire the **EVNT_BTN_TAP (0x0A)** IBIOS Event:

`0x02 0x47` **`0x0A`** `0x00`.

You should see the same 'Button tap detected' message displayed in the IDE's ASCII Window.

IBIOS INSTEON Engine

The IBIOS INSTEON Engine handles INSTEON message transport. The format and meaning of INSTEON messages are described in detail in the section *INSTEON Messages*. How INSTEON messages propagate in an INSTEON network is explained in the section *INSTEON Signaling Details*.

You can use the method given in the *Send INSTEON* example in the *IBIOS Serial Commands* section to send an INSTEON message with the INSTEON Engine. However, receiving INSTEON messages is time-critical, and although it is technically possible to wait for one of the 'INSTEON message received' events and then poll the INSTEON receive buffer, the buffer can easily be overwritten by new INSTEON messages if it is not read quickly enough. Therefore, the safest way to receive INSTEON messages is to use the *SALad coreApp Program* pre-installed in *The Smarthome PowerLinc Controller*, or else to write your own SALad application that employs the same method as coreApp. See the *Receive INSTEON* example in the *IBIOS Serial Commands* section for more details.

The INSTEON Engine automatically handles *INSTEON Message Hopping* and *INSTEON Message Retrying*. It also deals with the *Message Integrity Byte* so you don't have to. Whenever you send or receive a Direct (Point-to-Point) INSTEON message, the INSTEON Engine knows about the expected Acknowledgement message and handles it for you, then fires one of the EVNT_ITX_ACK or EVNT_ITX_NACK *IBIOS Events* to notify you of the outcome.

The current INSTEON Engine does not handle *INSTEON Groups* and Group Cleanup messages, nor anything involving the *INSTEON Link Database*, although this functionality is planned for a later version. The SALad coreApp program does handle these functions at a higher level, however, so you do not have to worry about coding them yourself.

IBIOS Software Realtime Clock/Calendar

IBIOS keeps time using a software realtime clock (RTC) that ticks once per second. Devices that also have a hardware RTC can use it to set the software RTC. The *SALad coreApp Program* uses the hardware RTC in *The Smarthome PowerLinc Controller* to set the software RTC at power up and also every midnight.

You can set the software RTC manually using the registers shown below, excerpted from the *Flat Memory Map*.

Address	Register and Bits	Description
0x0158	RTC_TIME_H	Time since midnight in minutes (MSB, 0-1439)
0x0159	RTC_TIME_L	Time since midnight in minutes (LSB, 0-1439)
0x015A	RTC_YEAR	Year (0-99)
0x015B	RTC_MON	Month (1-12)
0x015C	RTC_DAY	Day (1-31, month specific)
0x015D	RTC_DOW	Day-of-Week bitmap (0SSFTWTM)
0x015E	RTC_HOUR	Hour (0-23)
0x015F	RTC_MIN	Minute (0-59)
0x0160	RTC_SEC	Second (0-59)

When you set the software RTC, you should also set the *RTC_TIME_H,L* minutes-from-midnight value, because IBIOS will only set it by zeroing it at the next midnight.

The software RTC handles leap year, but it does not handle daylight-savings time. CoreApp, however, does handle daylight savings.

IBIOS X10 Signaling

When IBIOS receives an X10 byte over the powerline, it fires an **EVNT_XRX_MSG (0x08)** or **EVNT_XRX_XMSG (0x09)** IBIOS Event, as explained in *IBIOS Event Details*, Note 6. If the *SALad coreApp Program* or another SALad application with an appropriate event handler is running, SALad will send an **X10 Byte Received (0x4A)** IBIOS Serial Command, as explained in *IBIOS Serial Command Details*, Note 9.

The manual method for transmitting an X10 Address followed by an X10 Command is explained in the *Send X10* IBIOS Serial Command example.

The following excerpt from the *Flat Memory Map* shows the registers and flags that IBIOS uses for sending and receiving X10 bytes.

Address	Register and Bits		Description
0x0164	X10_RX		X10 Receive Buffer
0x0165	X10_TX		X10 Transmit Buffer
0x0166	X10_FLAGS		X10 Flags
	_X10_RTS	7	1=Request To Send
	_X10_EXTENDED	5	1=Extended transfer in progress (Tx or Rx)
	_X10_TXCOMMAND	3	1=Command, 0=Address for transmit
	_X10_RXCOMMAND	2	1=Command, 0=Address for receive
	_X10_VALID	1	1=X10 receive valid

To send an X10 byte, place it in the *X10_TX* buffer, set or clear *_X10_TXCOMMAND* to show whether it is an X10 Command or X10 Address, then set the *_X10_RTS* flag.

To see if there is a new received X10 byte in the *X10_TX* buffer, inspect the *_X10_VALID* flag, or just wait for an **EVNT_XRX_MSG (0x08)** or **EVNT_XRX_XMSG (0x09)** IBIOS Event. Look at *_X10_RXCOMMAND* to see if the received byte is an X10 Command or X10 Address, and look at *_X10_EXTENDED* to see if it is part of an X10 Extended message.

IBIOS Input/Output

IBIOS I/O drivers are limited to an *IBIOS LED Flasher* and an *IBIOS SET Button Handler*.

IBIOS LED Flasher

You can control LED flashing using the following registers excerpted from the *Flat Memory Map*.

Address	Register and Bits	Description
0x0168	LED_MODE	Bitmap defines flashing pattern for LED 1=On, 0=Off
0x0169	LED_TMR	Duration of LED flashing in seconds
0x016A	LED_DLY	Period between each flash. Defaults to 5, which is 1/8 second per bit in LED_MODE.

LED_MODE is a bitmap that defines 8 on or off periods for the LED, and *LED_DLY* defines how fast the bits are shifted to flash the LED. The default *LED_DLY* value is 5, which is 1/8 second per bit. Larger values will slow down the flashing.

To flash the LED, load the number of seconds that you want it to flash into *LED_TMR*.

For example, to flash the LED on and off at half-second intervals for three seconds, load 0xF0 into *LED_MODE* and then load 0x03 into *LED_TMR*.

IBIOS SET Button Handler

Pushing the *SET Button* generates **EVNT_BTN_TAP (0x0A)**, **EVNT_BTN_HOLD (0x0B)**, and **EVNT_BTN_REL (0x0C)** IBIOS Events, as explained in *IBIOS Event Details*, Note 7. The *TAP_CNT* register in the *Flat Memory Map*. Lets you see how many times the *SET Button* was tapped.

Address	Register and Bits	Description
0x0156	TAP_CNT	Counts multiple *SET Button* taps

IBIOS Remote Debugging

You can remotely debug a SALad program with the **Debug Report (0x49)** IBIOS Serial Command (see *IBIOS Serial Command Details*). This is the underlying mechanism used by the IDE debugger described in the *SALad Integrated Development Environment User's Guide*.

Three flags in the *NTL_CONTROL* register at 0x0076 (see *Flat Memory Map*) control IBIOS remote debugging. These flags are bit 7, (*_RD_STEP*), bit 6 (*_RD_HALT*), and bit 5 (*_RD_BREAK*). A 16-bit address for breakpoints or range checking can be set in the *RD_H* and *RD_L* registers at 0x0026.

Address	Register and Bits		Description		
0x0026	RD_H		Remote Debugging breakpoint address MSB		
0x0027	RD_L		Remote Debugging breakpoint address LSB		
0x0076	NTL_CONTROL		SALad debugging control flags		
	_RD_STEP _RD_HALT	7 6	_RD_HALT	_RD_STEP	
			0	0	Normal execution
			0	1	Animation (Trace)
			1	0	Execution halted
			1	1	Single step requested
	_RD_BREAK	5	0=Range Check Mode, 1=Breakpoint Mode		

To run a SALad program normally, clear both *_RD_HALT* and *_RD_STEP*. This is the default at power up.

To halt a SALad program as soon as possible, set *_RD_HALT* and clear *_RD_STEP*. Execution will stop after the current instruction executes and a **Debug Report (0x49)** IBIOS Serial Command will report the address of the next instruction to be executed.

To send a **Debug Report** before *every* instruction executes, clear *_RD_HALT* and set *_RD_STEP*.

To single-step through a SALad program, set both *_RD_HALT* and *_RD_STEP*. This will cause the next instruction to execute followed by an immediate halt. The halt will cause a **Debug Report** to be sent.

To do range checking or to set a breakpoint, load a comparison address into the *RD_H* and *RD_L* registers. If *RH_H* contains 0x00 (the default power-up condition), range checking and breakpoints are disabled.

Clear the *_RD_BREAK* flag to use the comparison address for range checking or else set *_RD_BREAK* to use the comparison address as a breakpoint.

If you are range checking and program execution is attempted at a location greater than the comparison address, execution will be halted, a **Debug Report** will be sent, and the *IBIOS Watchdog Timer* will cause a power-on reset.

If you are using the comparison register for a breakpoint, execution will halt only if the beginning of the next instruction exactly matches the comparison address.

You can also do remote debugging over the INSTEON network alone by setting the flag *_I_DebugRpt* (bit 6) in the *I_Control* register at 0x0142. This flag is cleared at power up. When the *_I_DebugRpt* flag is set, every time a **Debug Report (0x49)** IBIOS Serial Command would be sent over a serial connection, a **Debug Report (0x49)** INSTEON Command from the *INSTEON Broadcast Command Summary Table* will be sent in an INSTEON Broadcast message. You can use INSTEON peek and poke commands from the *INSTEON Common Command Summary Table* to set the comparison address and the debugging control flags in the remote INSTEON device.

IBIOS Watchdog Timer

The watchdog timer in IBIOS is automatic. IBIOS sets appropriate timeout values for itself and resets the watchdog whenever it returns from a task before the timeout. If a timeout does occur, the watchdog code performs a power-on reset, which puts the device in the same state as cycling power does.

There are two ways to force the watchdog timer to cause a reset. One will occur if you are using IBIOS debugging to do program counter range checking (see *IBIOS Remote Debugging*) and the range is exceeded. The other way is to set the *_Reset* flag (bit 7) in the *Control* register at 0x0154 to one (see *Flat Memory Map*). Both conditions cause IBIOS to execute an endless loop, which will eventually time out the watchdog.

You can manually reset the watchdog (buying more time) by setting the *_Watchdog* flag (bit 6) in the *Control* register at 0x0154 to one.

SALad Language Documentation

The INSTEON PowerLinc™ V2 Controller (PLC) and other planned INSTEON devices contain an embedded language interpreter, called SALad, that allows programming of complex behavior into SALad-enabled devices. See *SALad Applications* in the *INSTEON Application Development Overview* section for a description of the application development process using SALad.

The SALad language is designed to make execution of INSTEON Internal Applications fast, while keeping the size of the programs small.

SALad is event driven. Examples of events that can occur in a PLC include reception of an INSTEON message or an X10 command, expiration of a timer, or pushing the *SET Button*. As events occur, the PLC firmware posts event handles to an event queue. The firmware then starts the SALad program with the current event handle located in a specific memory location called NTL_EVENT. The SALad program inspects NTL_EVENT in order to determine what action to take based on the event that occurred. Most SALad programming is just a matter of writing event-handling routines, or modifying the routines found in sample applications.

The SALad Integrated Development Environment (IDE) makes writing and debugging SALad programs fast and easy. Besides a built-in SALad editor, compiler, and debugger, the IDE contains an integrated set of INSTEON-specific tools that give the programmer access to every aspect of the INSTEON environment.

In This Section

SALad Programming Guide
: Shows the structure of a SALad program, lists sample 'Hello World' programs, and gives tips for developing SALad applications.

SALad Language Reference
: Lists the register locations critical to SALad, and describes the SALad instruction set and addressing modes.

SALad Integrated Development Environment User's Guide
: Describes a comprehensive software tool used for writing and debugging embedded SALad applications.

SALad Programming Guide

In This Section

Structure of a SALad Program

Application Header

All SALad applications require a program header for program verification. This header can have many pieces of information in it, but it must contain the application verification data.

Starting at address 0x0210 (see *Flat Memory Map*), there are 8 bytes of data that define a region of code that will not be altered during execution. This is used to test the application for possible corruption.

Address	Register and Bits	Description
0x0210 ⇒ 0x0211	APP_ADDR_TEST	Address of range of application for verification test
0x0212 ⇒ 0x0213	APP_LEN_TEST	Length of range of application for verification test
0x0214 ⇒ 0x0215	APP_CHECK_TEST	Two's complement checksum of range of application for verification test
0x0216 ⇒ 0x0217	APP_END	Top of currently loaded SALad application. EEPROM is write-protected from 0x0200 to the address contained here. Set 0x16B bit 7 to enable over-writing.

APP_ADDR_TEST is the address of the beginning of the application code, normally 0x0230. *APP_LEN_TEST* is the length of the region to be tested on device initialization, normally the length of the application. *APP_CHECK_TEST* is a two's complement checksum of that region. If you are using the SALad IDE, it will fill in this number for you. *APP_END* is the address of the last byte-plus-one in the currently loaded application, and is used to write-protect the EEPROM code segment (see *Overwriting a SALad Application*).

When the PLC is reset, IBIOS uses the checksum to verify the SALad program before running it. If it is corrupt, IBIOS will flash the PLC's LED at about 2 Hz. If the application is valid, the LED will be lit continuously.

An Application Header structure using literal values might look like this:

```
ORG 0x210
  DATA 0x0230 ; address of beginning of application
  DATA 0x000a ; length of application
  DATA 0x0041 ; checksum of verification region
  DATA 0x023b ; end of application
```

If you are using the SALad IDE, you can use labels and skip filling in the checksum, like this:

```
ORG 0x0210
;=======Application Header================================
       DATA    Start          ;Beginning of application
       DATA    App_End-Start  ;Length of application
       DATA    0x00, 0x00     ;Two's complement checksum
       DATA    App_End        ;End of application
```

Program Body

The general structure of a SALad program is similar to that of other low-level assembly programming languages, with varying details depending on the application. As a stand-alone language, SALad has no specific structural requirements. However,

most INSTEON applications are event-driven, requiring that SALad event handlers follow a definite structural organization.

For a simple direct SALad application, start the program at 0x0237, which is the standard entry vector for Static *IBIOS Events*. When the PLC is reset (either by power cycling or a reset command), the SALad application will be started with an initialization **EVNT_INIT (0x00)** IBIOS Event posted to the event queue.

It is possible to write a SALad application without event handlers, but you must then poll all hardware for a change of state, and *SALad Timers* will not work without event processing.

By far the easiest way to write SALad applications is use the IDE as explained in the *SALad Integrated Development Environment User's Guide* to modify the event handlers in the open-source *SALad coreApp Program*.

The SALad Version of Hello World

The SALad language contains general commands that are useful for most SALad-enabled INSTEON products. Any commands that are unique to a subset of the product line are provided through the API (Application Program Interface) feature. This allows custom firmware and commands to be added to the SALad language without altering the core SALad engine.

In this example, the RS232 port is utilized to send a `Hello World!` ASCII message. Because the RS232 port is not found in all the Smarthome products, these commands are provided as API calls. In this case, we will use the `API_RS_SendStr` command which has a command code of 0x82. The format of this command is the API code (0x0A) followed by the command code (0x82) followed by the 2-byte address of the data containing the message to send.

When this example is executed, the output will be to the RS232 Port at 4800 Baud, 8 bits, 1 stop bit, and no parity. The application looks like this:

```
ORG 0x0210
;=======Application Header================================
        DATA    Start           ;Beginning of application
        DATA    App_End-Start   ;Length of application
        DATA    0x00, 0x00      ;Two's complement checksum
        DATA    App_End         ;End of application

ORG 0x0237
;=======Application Code==================================
Start
        API     0x82, Message   ;Send Message to RS232 port
        JUMP    Start           ;Loop non-stop
Message
        DATA    "Hello "
        DATA    "World!"
        DATA    0x0D,0x0A,0x00 ;Zero terminates string
App_End
```

SALad Event Handling

As events occur, IBIOS posts messages to a message queue that the SALad application can respond to for processing. When SALad is idle, the event processor continually looks for new events to process. When one appears, IBIOS puts the Event Handle in the *NTL_EVENT* register and then calls SALad at one of two possible vectors:

0x0230 – for Timer Events (see *SALad Timers*)

0x0237 – for Static Events (see *IBIOS Events*)

Upon entry, the SALad application must process the event handle in *NTL_EVENT* in order to run the correct event handler. In the following code, taken from the *SALad coreApp Program*, the event handle is used as an index into a table of addresses that point to the beginning of the event handling routines.

```
;=======Register Definitions=============================
APP_TMP_H       EQU     0x006E  ;These stay alive only until
APP_TMP_L       EQU     0x006F  ;the first CALL.  They are
                                ;used for the Event handler.

ORG 0x0210
;=======Application Header===============================
        DATA    Start           ;Beginning of application
        DATA    App_End-Start   ;Length of application
        DATA    0x00, 0x00      ;Two's complement checksum

ORG 0x0230
;=======Event Process Code===============================
Start
StartTimer
        JUMP    ProcessTimer            ;jump to Timer process

StartEvent
        MOVE    NTL_EVENT, APP_TMP_L    ;setup event offset
        MUL     #0x0002, APP_TMP_H      ;multiply by 2 for
                                        ;word offset (16bit)
        ADD     #EventJmpTbl, APP_TMP_H ;add table address 0243:
                                        ;to offset (16bit)
        JUMP    ProcessEvent            ;process table entry

ProcessTimer
        MOVE    NTL_EVENT, APP_TMP_L    ;setup event offset
        MUL     #0x0002, APP_TMP_H      ;multiply by 2 for
                                        ;word offset (16bit)
        ADD     #EventJmpTbl,APP_TMP_H ;add table address 0256:
                                        ;to offset (16bit)
ProcessEvent
         MOVE$  @APP_TMP_H,APP_TMP_H,2 ;get indirect pointer
                                        ;from table
        JUMP    @APP_TMP_H              ;execute code at table
                                        ;entry
EventJmpTbl
        DATA    Event00                 ;EVNT_INIT
        DATA    Event01                 ;EVNT_IRX_MSG
        DATA    Event02                 ;EVNT_IRX_NACK
        DATA    Event03                 ;EVNT_XRX_MSG
        DATA    Event04                 ;EVNT_XRX_XMSG
        DATA    Event05                 ;EVNT_BTN_TAP
        DATA    Event06                 ;EVNT_BTN_HOLD
        DATA    Event07                 ;EVNT_BTN_REL
        DATA    Event08                 ;EVNT_ALARM
        DATA    Event09                 ;EVNT_MIDNIGHT
        DATA    Event0A                 ;EVNT_2AM
```

```
        DATA    Event0B                 ;EVNT_SRX_COM

TimerJmpTbl
        DATA 0x00, 0x00                 ;No Timer Events

;=======Static Events=============================
;-------EVNT_INIT
Event00
        END

;-------EVNT_IRX_MSG
Event01
        END

;-------EVNT_IRX_NACK
Event02
        END

;-------EVNT_XRX_MSG
Event03
        END

;-------EVNT_XRX_XMSG
Event04
        END

;-------EVNT_BTN_TAP
Event05
        END

;-------EVNT_BTN_HOLD
Event06
        END

;-------EVNT_BTN_REL
Event07
        END

;-------EVNT_ALARM
Event08
        END

;-------EVNT_MIDNIGHT
Event09
        END

;-------EVNT_2AM
Event0A
        END

;-------EVNT_SRX_COM  - Serial Received a command
Event0B
        END

;======Timer Events================================
;       No Timer Events
App_End
```

Hello World 2 – Event Driven Version

The following example shows the "Hello World" program implemented as an event driven process. This version prints "Hello World!" out the RS232 port each time the *SET Button* is pressed. This demonstrates the processing of the **EVNT_BTN_TAP (0x0A)** and **EVNT_BTN_HOLD (0x0B)** *IBIOS Events* that are generated when the button is tapped or held. If the button is pressed for more than 350 ms, the program will send "Good-Bye World!" instead.

When this example is executed, the output will be to the RS232 Port at 4800 Baud, 8 bits, 1 stop bit, and no parity. The program is a simple modification of the *SALad coreApp Program* event processor shown in *SALad Event Handling*.

```
;=======Register Definitions=============================
APP_TMP_H       EQU     0x006E  ;These stay alive only until
APP_TMP_L       EQU     0x006F  ;the first CALL.  They are
                                ;used for the Event handler.


ORG 0x0210
;=======Application Header================================
        DATA    Start           ;Beginning of application
       DATA App_End-Start       ;Length of application
        DATA    0x00, 0x00      ;Two's complement checksum


ORG 0x0230

;=======Event Process Code================================
Start
StartTimer
        JUMP    ProcessTimer            ;jump to Timer process

StartEvent
        MOVE    NTL_EVENT, APP_TMP_L    ;setup event offset
         MUL    #0x0002, APP_TMP_H      ;multiply by 2 for
                                        ;word offset (16bit)
        ADD     #EventJmpTbl, APP_TMP_H ;add table address 0243:
                                        ;to offset (16bit)
        JUMP    ProcessEvent            ;process table entry
ProcessTimer
        MOVE    NTL_EVENT, APP_TMP_L    ;setup event offset
       MUL      #0x0002, APP_TMP_H      ;multiply by 2 for
                                        ;word offset (16bit)
        ADD     #EventJmpTbl,APP_TMP_H ;add table address 0256:
                                        ;to offset (16bit)
ProcessEvent
MOVE$   @APP_TMP_H,APP_TMP_H,2 ;get indirect pointer
                                        ;from table
        JUMP    @APP_TMP_H              ;execute code at table
                                        ;entry
EventJmpTbl
        DATA    Event00                 ;EVNT_INIT
        DATA    Event01                 ;EVNT_IRX_MSG
        DATA    Event02                 ;EVNT_IRX_NACK
        DATA    Event03                 ;EVNT_XRX_MSG
        DATA    Event04                 ;EVNT_XRX_XMSG
        DATA    Event05                 ;EVNT_BTN_TAP
        DATA    Event06                 ;EVNT_BTN_HOLD
        DATA    Event07                 ;EVNT_BTN_REL
        DATA    Event08                 ;EVNT_ALARM
        DATA    Event09                 ;EVNT_MIDNIGHT
        DATA    Event0A                 ;EVNT_2AM
        DATA    Event0B                 ;EVNT_SRX_COM
TimerJmpTbl
```

```
        DATA 0x00, 0x00              ;No Timer Events

;=======Static Events=============================
;-------EVNT_INIT
Event00
        END
;-------EVNT_IRX_MSG
Event01
        END
;-------EVNT_IRX_NACK
Event02
        END
;-------EVNT_XRX_MSG
Event03
        END
;-------EVNT_XRX_XMSG
Event04
        END
;-------EVNT_BTN_TAP
Event05
        API     0x82, Message1 ;Send Message to RS232 port
        END
;-------EVNT_BTN_HOLD
Event06
        API     0x82, Message2 ;Send Message to RS232 port
        END
;-------EVNT_BTN_REL
Event07
        END
;-------EVNT_ALARM
Event08
        END
;-------EVNT_MIDNIGHT
Event09
        END
;-------EVNT_2AM
Event0A
        END

;-------EVNT_SRX_COM  - Serial Received a command
Event0B
        END

;======Timer Events================================
;       No Timer Events
Message1
        DATA    "Hello "
        DATA    "World!"
        DATA    0x0D,0x0A,0x00 ;Zero terminates string
Message2
        DATA    "GoodBye "
        DATA    "World!"
        DATA    0x0D,0x0A,0x00 ;Zero terminates string
App_End
```

SALad coreApp Program

As shipped by Smarthome, the PowerLinc™ V2 Controller (PLC) contains a 1200-byte SALad program called coreApp that performs a number of useful functions:

- When coreApp receives messages from INSTEON devices, it sends them to the computing device via its serial port, and when it receives INSTEON-formatted messages from the computing device via the serial port, it sends them out over the INSTEON network.
- CoreApp handles linking to other INSTEON devices and maintains a *PLC Link Database*.
- CoreApp is event-driven, meaning that it can send messages to the computing device based on *IBIOS Events* and *SALad Timers*.
- CoreApp can send and receive X10 commands.
- CoreApp sets the software realtime clock using the hardware realtime clock and handles daylight savings time.

Several of the IBIOS Events listed in the *IBIOS Event Summary Table* and IBIOS Serial Commands listed in the *IBIOS Serial Command Summary Table* *require* SALad event handlers like those in coreApp in order to ensure realtime execution.

Source code for coreApp is available to developers to modify for their own purposes. Using the tools described in the *SALad Integrated Development Environment User's Guide* to modify coreApp is by far the easiest way to develop your own SALad applications. Once programmed with an appropriately modified SALad App, the PLC can operate on its own without being connected to a computing device.

SALad Timers

You can set up a SALad Timer Event by loading 2 bytes into a timer buffer. The first byte, called the *Timer Index*, is the number of the timer handler routine that you want to execute when the Timer Event fires. The second byte, called the *Timer Time*, designates how much time you want to elapse from the time you set up the Timer Event until the Timer Event fires. If the high bit of *Timer Time* is 0, the other 7 bits designate 1 to 127 seconds; if the high bit is 1, the other 7 bits designate 1 to 127 minutes. A *Timer Time* of 0x00 designates that the timer is disabled.

The default number of co-pending Timer Events that you can set up is four. If you need more Timer Events, increase the pointer value stored in *NTL_BUFFER* at 0x0033 (see *Flat Memory Map*) by two for each additional Timer Event you want to support. *NTL_BUFFER* points to the *end* of the timer buffer, which *begins* at 0x0046.

You need to write a SALad timer handler routine for each *Timer Index* that you will be using. *Timer Index 1* will fire the first handler, *Timer Index 2* will fire the second handler, and so on.

There are four SALad instructions for setting up and removing Timer Events (see *Two-byte SALad Instructions*):

ONESHOT (*Timer Index*, *Timer Time*)
Set up a new Timer Event if there is no pre-existing Timer Event with the same *Timer Index*. If there is such a Timer Event, replace its *Timer Time* value. After this Timer Event fires, remove it by setting its *Timer Index* to 0x00.

TIMER (*Timer Index*, *Timer Time*)
Set up a new Timer Event if there is no pre-existing Timer Event with the same *Timer Index*. If there is such a Timer Event, replace its *Timer Time* value. After this Timer Event fires, disable it by setting its *Timer Time* to 0x00, but do not remove it.

TIMERS (*Timer Index*, *Timer Time*)
Set up a new Timer Event unconditionally. After this Timer Event fires, disable it by setting its *Timer Time* to 0x00, but do not remove it.

KILL (*Timer Index*)
If there is a pre-existing Timer Event with the same *Timer Index*, remove it by setting its *Timer Index* to 0x00.

If you try to set up a Timer Event but there is no room in the timer buffer, the Timer Event will not be set up. If this happened, the *_NTL_BO* buffer overrun flag (bit 1) of *NTL_STAT* at 0x0075 will be 1.

SALad Remote Debugging

It is possible to debug SALad applications remotely using either Serial (RS232 or USB) or INSTEON communications. IBIOS provides the necessary support. See *IBIOS Remote Debugging* for a full explanation of how this works at the low level.

The comprehensive debugging features built into the SALad IDE (see the *SALad Integrated Development Environment User's Guide*) are built on this low level IBIOS mechanism. Therefore, the easiest way to take advantage of remote SALad debugging is to use the SALad IDE.

Overwriting a SALad Application

SALad code in EEPROM is write-protected from 0x0200 to the top of the SALad application pointed to by *APP_END*, as shown in the *Flat Memory Map* excerpt below. You must set the *_RS_AppLock* flag before you attempt to write to this area.

Address	Register and Bits		Description
0x016B	RS_CONTROL		Control flags for serial command interpreter
	_RS_AppLock	3	1=Enable overwriting of SALad code from 0x0200 to end of SALad App given in 0x0216 and 0x0217
0x0216 ⇒ 0x0217	APP_END		Top of currently loaded SALad application. EEPROM is write-protected from 0x0200 to the address contained here. Set 0x16B bit 7 to enable over-writing.

This mechanism is not implemented in PLC firmware versions earlier than 2.10K.

Preventing a SALad Application from Running

While developing a SALad application that runs on the PLC, you may need to manually prevent the SALad code from executing, because of faulty SALad code that prevents serial communication.

To prevent SALad execution while allowing IBIOS to run normally, remove and then re-apply power to the PLC while holding down the *SET Button* for 5 seconds. IBIOS will then write the *complement* of the SALad program's checksum to the SALad checksum verification register. Since the SALad program's checksum will not match the complemented checksum, the SALad program will not run.

To restore the SALad checksum verification register to its correct value, just complement it again by repeating the 'apply power while holding the *SET Button* for 5 seconds' procedure.

SALad Language Reference

In This Section

SALad Memory Addresses
Describes SALad's usage of memory.

SALad Instruction Set
Documents all SALad instructions and addressing modes.

SALad Memory Addresses

SALad uses the same *Flat Memory Addressing* as IBIOS. See the *Flat Memory Map* for a table of important memory locations.

See *Structure of a SALad Program* for the location of the SALad program itself and the structure of its header.

SALad program flags appear in the register *NTL_STAT* as follows:

Address	Register and Bits		Description
0x0075	NTL_STAT		SALad Status Register
	_DB_END	4	1=Enrollment Link Database search reached end of database
	_DB_PASS	3	1=Enrollment Link Database search successful
	_NTL_DZ	2	1=Divide by Zero
	_NTL_BO	1	1=Buffer Overrun
	_NTL_CY	0	1=Carry from Math and Test operations

SALad Instruction Set

The SALad instructions are designed to support all the processing needs of a basic programming language. Additional device-specific functions can be defined using Application Programming Interface (API) calls to firmware.

SALad universally supports access to or from RAM or EEPROM directly or indirectly. Literal data can be provided for immediate operations. The *Compare* and *Move* instructions have a block mode to perform string compares or block moves.

Indirect addressing modes support address pointers and jump tables.

In This Section

SALad Universal Addressing Module (UAM)

The UAM is a mechanism for encoding addressing mode and parameter information in the instructions. The commands have UAM-specific information in the low nibble of the instruction. These bits define what kind of memory is being accessed and the number of parameter bytes.

SALad instructions use Full-UAM, Half-UAM, or No-UAM encodings. Full-UAM instructions take two parameters, Half-UAM instructions take one parameter, and No-UAM instructions do not use the UAM at all and have no parameters.

These tables show how SALad instructions are UAM-encoded in a byte:

Full-UAM SALad Instruction							
Command				**Source Parameter**	**Destination Parameter**	**Mode**	
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
4-bit command identifier				0=8 bit Parameter 1=16 bit Parameter	0=8 bit Parameter 1=16 bit Parameter	00 = Direct to Direct 01 = Direct to Indirect 10 = Indirect to Direct 11 = Literal to Direct	

Half-UAM SALad Instruction							
Command							**Mode**
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
7-bit command identifier							0=Direct 1=Indirect

No-UAM SALad Instruction							
Command							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
8-bit command identifier							

SALad Parameter Encoding

Parameters are of types:

- Register
- Direct
- Indirect
- 8-bit Literal Value
- 16-bit Literal Value

A Register parameter is an 8-bit location in memory that is referenced by an offset that is added to a base value stored in NTL_EVENT.

A Direct parameter is a parameter that is referenced by a 16-bit address in the *Flat Memory Map*.

An Indirect parameter is a pointer to a location in memory.

Literal values are constant values that are coded directly into the program.

In This Section

Parameter Reference Tables
Contains reference tables for encoding SALad instructions.

Parameter Reference Tables

Parameter Types

This table shows the types of parameters described above, whether they are 8 or 16-bit, whether they refer to addresses relative to the Program Counter or relative to the Absolute address in the *Flat Memory Map*, and whether they are Indirect or Direct.

Ref	Description	8-bit or 16-bit	Relative or Absolute	Indirect or Direct
Rn	Direct Register Mode (n + NTL_REG)	8-bit	Absolute	Direct
@Rn	Indirect Register Mode (n + NTL_REG)	8-bit	Absolute	Indirect
D	Direct Mode (PC + D)	16-bit	Relative	Direct
@D	Indirect Mode (PC + D)	16-bit	Relative	Indirect
#8	8-bit Literal	8-bit	Absolute	Direct
#16	16-bit Literal	16-bit	Absolute	Direct

Full-UAM Instruction Encoding

This table shows how to encode the low nibble for full UAM instructions (i.e. instructions that take both source and destination parameters). Parameter references are from the table above.

Command and Parameters	Instruction Code	Command and Parameters	Instruction Code
Command Rn, Rn	0x?0	Command D, Rn	0x?8
Command Rn, @Rn	0x?1	Command D, @Rn	0x?9
Command @Rn, Rn	0x?2	Command @D, Rn	0x?A
Command #8, Rn	0x?3	Command #16, Rn	0x?B
Command Rn, D	0x?4	Command D, D	0x?C
Command Rn, @D	0x?5	Command D, @D	0x?D
Command @Rn, D	0x?6	Command @D, D	0x?E
Command #8, D	0x?7	Command #16, D	0x?F

Half-UAM Instruction Encoding

This table shows how to encode half UAM instructions (i.e. instructions that take only a destination parameter). Parameter references are from the table above. Instruction Code is shown as 8-bit pattern.

Command and Parameters	Instruction Code	Command and Parameters	Instruction Code
Command D	xxxxxxx0	Command @D	xxxxxxx1

SALad Instruction Summary Table

One-byte SALad Instructions

Command	Code	UAM	Parameters	Description
NOP	0x00	None		No Operation
RET	0x01	None		Return from call
END	0x02	None		Return to System
Function	0x03	None		Enable extended function mode
API	0x04	None		Execute external firmware routines
8BIT	0x05	None		Select 8 bit processing
16BIT	0x06	None		Select 16 bit processing
HALT	0x07	None		Halt SALad execution
RL	0x08⇒0x09	Half-UAM	<dest>	Rotate Left value stored in dest by 1-bit
RR	0x0A⇒0x0B	Half-UAM	<dest>	Rotate Right value stored in dest by 1-bit
JUMP	0x0C⇒0x0D	Half-UAM	<dest>	Jump to specified destination relative to PC
CALL	0x0E⇒0x0F	Half-UAM	<dest>	Call routine at specified destination relative to PC
TEST	0x10⇒0x1F	Half-UAM	<dest><jump>	Test bit in byte at destination, jump if false
CLR	0x20⇒0x2F	Half-UAM	<dest>	Clear bit in byte at destination
SET	0x30⇒0x3F	Half-UAM	<dest>	Set bit in byte at destination
ADD	0x40⇒0x4F	Full-UAM	<source><dest>	Add source to destination, store result in destination
OR	0x50⇒0x5F	Full-UAM	<source><dest>	OR source to destination, store result in destination
AND	0x60⇒0x6F	Full-UAM	<source><dest>	AND source to destination, store result in destination
XOR	0x70⇒0x7F	Full-UAM	<source><dest>	XOR source to destination, store result in destination
COMP>	0x80⇒0x8F	Full-UAM	<source><dest><jump>	Compare source greater to destination, jump if false
COMP<	0x90⇒0x9F	Full-UAM	<source><dest><jump>	Compare source less than destination, jump if false

Command	Code	UAM	Parameters	Description
LOOP-	0xA0⇒0xAF	Full-UAM	<source><dest><jump>	Decrement destination, branch while greater than source
LOOP+	0xB0⇒0xBF	Full-UAM	<source><dest><jump>	Increment destination, branch while less than source
MOVE	0xC0⇒0xCF	Full-UAM	<source><dest>	Move source to destination
COMP=	0xD0⇒0xDF	Full-UAM	<source><dest><jump>	Compare source equal to destination, jump if false
SUB	0xE0⇒0xEF	Full-UAM	<source><dest>	Subtract source from destination, store result in destination

Two-byte SALad Instructions

These are Extended Commands, preceded by 0x03.

Command	Code	UAM	Parameters	Description
ENROLL	0x03 0x00	None		Enable Enrollment for 2 minutes Starts 2-minute enrollment timer and enables Enrollment command pass-through. a. Button must be pushed when this command is executed b. Timer is restarted when valid enrollment command is received c. Enrollment command generates a unique event: EVNT_IRX_ENROLL or 0x07
NEXT	0x03 0x01	None		Find next group in Link Database
SEND	0x03 0x02	None		Send INSTEON
ENDPROC	0x03 0x03	None		Skip parameter on stack and return
KILL	0x03 0x04	None	<index>	Delete pending timer event specified by index from event queue
PAUSE	0x03 0x05	None	<time>	Pause for time in 25ms increments

Command	Code	UAM	Parameters	Description
FIND	0x03 0x06	None	<flags>	Find record in database. These bits define the field(s) of the record that you are searching for in the Enrollment database. Flags is defined as: DB_FLAGS 76543210 XXXXXXXX ID0-------------+||||||| ID1--------------+|||||| ID2---------------+||||| Group---------------+|||| Linear Search-------+||| Mode-----------------++| INSTEON Master/Slave---+ Search Mode (Mode bits): 00 Deleted 01 Other 10 INSTEON Slave 11 INSTEON Master DB_Flags configuration examples: EMPTY EQU 0x00 ; look for empty slot ; index is in DB_H SLAVE EQU 0xE4 ; match ID, look for ; SLAVE, ID is in ; RxFrom0-RxFrom2 MASTER EQU 0xE6 ; match ID, look for ; MASTER, ID is in ; RxFrom0-RxFrom2 INSTEON EQU 0xE5; match ID, look for ; MASTER or SLAVE, ; ID is in RxFrom0- ; RxFrom2 MEMBER EQU 0x1C ; match group, look ; for SLAVE, index ; is in DB_0 GROUP EQU 0xF6 ; match ID and group, ; look for MASTER, ID ; is in RxFrom0- ; RxFrom2, group is ; in RxTo2 To Find a member of a local group, set group number in DB_3 and execute: FIND MEMBER To Find either a Master or Slave INSTEON record that matches the received message, execute: FIND INSTEON
X10	0x03 0x08	None	<HC/UC><HC/com>	Send X10 message

Command	Code	UAM	Parameters	Description
LED	0x03 0x09	None	<pattern><time>	Flash LED; Pattern defines the blinking pattern, and time specifies the duration of the blinking from 0 to 255 seconds. For example: LED 0x55 0x0A would make the LED blink at 8Hz for 10 seconds; The delay between blinks can be controlled by writing to LED_DLY
RANDOM	0x03 0x0A	None	<limit><register>	Generates random number between 0 and *limit* and stores in *register*
RANDOMIZE	0x03 0x0B	None	<dest>	Get next random number from generator using 16 bit absolute address to an 8-bit seed value and stores in NTL_RND
ONESHOT	0x03 0x0C	None	<index><time>	Set One-Shot timer: Index is the event number that the firmware stores in NTL_EVENT when the timer expires; Time is 0⇒127 seconds, or 2 minutes 8 seconds ⇒ 130 minutes 8 seconds if MSb is set.
TIMER	0x03 0x0D	None	<index><time>	Set or Reset timer
TIMERS	0x03 0x0E	None	<index><time>	Set multiple timers
Undefined	0x03 0x0F	None		
X10EXT	0x03 0x10⇒ 0x03 0x11	Half-UAM	<dest>	Additional data for extended X10 message
SEND$	0x03 0x20 ⇒ 0x03 0x21	Half-UAM	<dest>	Send 9 byte INSTEON message located at destination (from ID is inserted automatically)
SENDEXT$	0x03 0x30 ⇒ 0x03 0x31	Half-UAM	<dest>	Send 23 byte INSTEON message located at destination (from ID is inserted automatically)
MUL	0x03 0x50⇒ 0x03 0x5F	Full-UAM	<source><dest>	Multiply source to destination, store result in destination and destination +1
DIV	0x03 0x60⇒ 0x03 0x6F	Full-UAM	<source><dest>	Divide source into destination, store result in destination
MOD	0x03 0x70⇒ 0x03 0x7F	Full-UAM	<source><dest>	Divide source into destination, store remainder in destination
PROC	0x03 0x80⇒ 0x03 0x8F	Full-UAM	<source><dest><jump>	Place source on stack and call destination, jump if _NTL_CY=0 upon return
MASK	0x03 0x90⇒ 0x03 0x9F	Full-UAM	<source><dest><jump>	AND source to destination, jump if zero
COMP$>	0x03 0xA0⇒ 0x03 0xAF	Full-UAM	<source><dest><len> <jump>	Compare source string greater to destination string, jump if false
COMP$<	0x03 0xB0⇒ 0x03 0xBF	Full-UAM	<source><dest><len> <jump>	Compare source string less than destination string, jump if false
TJUMP	0x03 0xC0⇒ 0x03 0xCF	Full-UAM	<source><dest><limit> source: <index> dest: <base address> limit: <end of table>	Increment destination, branch while less than source

Command	Code	UAM	Parameters	Description
TCALL	0x03 0xD0⇒ 0x03 0xDF	Full-UAM	<source><dest><limit> source: <index> dest: <base address> limit: <end of table>	Decrement destination, branch while greater than source

SALad Integrated Development Environment User's Guide

The SALad Integrated Development Environment (IDE) is a powerful tool for developing applications to run on SALad-enabled INSTEON devices. For information about the SALad Language, refer to the *SALad Programming* and *SALad Language Reference* sections below.

The SALad IDE's source code editor handles multiple files, using color to indicate code context. In debug mode, programmers can use single-stepping, breakpoints, tracing, and watches to find and fix coding errors quickly.

At the heart of the IDE is The SALad Compiler, which reads SALad language source files and creates SALad object code, along with an error listing and symbol map. Compiled object code can either be serially downloaded to a *real* Smarthome PowerLinc™ V2 Controller (PLC) plugged into the powerline, or else it can be run on a *virtual* PLC simulated in software. Besides the virtual PLC, the IDE can also simulate a virtual powerline environment with any number of virtual LampLinc™ devices plugged into it, so that developers can create and test complex SALad applications on a standalone PC before validating them in a real INSTEON environment.

Using an integrated set of INSTEON-specific tools, programmers can compose and monitor INSTEON, X10, ASCII, or raw data messages, simulate PLC or realtime clock/calendar events, and directly manipulate the PLC's Link Database, all without ever leaving the IDE.

In This Section

SALad IDE Quickstart
: Explains how to get started using the SALad IDE right away.

IDE Main Window
: Describes the IDE's main menus and toolbar.

IDE Editor
: Gives the features of the IDE's Editor.

IDE Watch Panel
: Explains how to use Watches during debugging.

IDE Options Dialog Box
: Discusses the IDE's setup options.

IDE Windows and Inspectors
: Explains the many additional tools available in the IDE.

IDE Virtual Devices
: Describes how to use the software PLC Simulator, Virtual Powerline, and Virtual LampLinc.

IDE Keyboard Shortcuts
: Shows how to use the keyboard to perform common actions in the IDE.

SALad IDE Quickstart

Follow these steps to get familiar with the IDE quickly. The IDE works with a Smarthome PowerLinc™ V2 Controller, called a PLC for short. You can either use a real PLC plugged into the powerline and connected to your PC via a USB or RS232 serial port, or else the IDE can simulate a virtual PLC in software. This quick tutorial will get you connected to the PLC, then guide you through compiling, downloading, testing and debugging a sample 'Hello World' SALad program

1. **Connect the PowerLinc Controller.**

   If you are using a real PLC, connect a USB or RS232 serial cable from it to your Windows PC, and then plug the PLC into an electrical outlet.

   If you wish to use the PLC Simulator instead, simply turn it on by selecting *PLC Simulator* from the IDE's *Mode* menu.

2. **Run the SALad IDE.**

   After installing the SALad IDE on your Windows PC, go to *Start->Programs->SALad IDE->SALad IDE* to launch the program.

3. **Test the serial connection to the PLC.**

   When you run the IDE for the first time, a Startup Wizard will automatically help you test the serial connection to the PLC. You can also launch the Startup Wizard from the *View* menu.

   If you would rather test the serial connection to the PLC manually, follow these steps:

   a. Click the *Edit->Application Options* menu item. An *Options* dialog box will appear.

   b. Under the *Communications* tab, select either *USB* or the *COM* (RS232) port you are using to connect to the PLC. If you are using the PLC Simulator, it does not matter what you select here.

   c. Click the *Connect Now* button, which will establish the serial connection and then automatically perform the same test as the *Test Connection* button. After a short delay, a `Successfully connected` message should appear next to the *Test Connection* button.

   d. You do not have to press the *Download Core Application* button because you will be downloading a different SALad application below.

   e. Click *OK* to close the *Options* window.

4. **Load the sample 'Hello World' SALad program.**

   In the IDE, press the *Open* toolbutton (or else click the *File->Open…* menu item), then find and open *HelloWorld1.sal* in the *Samples* directory. A collection of files will appear in the editor window, with each file under a different tab. The tab labeled *HelloWorld1* should automatically be selected after the files finish loading into the editor.

5. **Mark *HelloWorld1* as the Main Code Module.**

   Right-click on the *HelloWorld1* tab and select *Mark as Main Code Module*. This tells the compiler which file contains the beginning of the SALad application program.

6. **View the ASCII Communications Window.**

   Select the *Comm Window* tab at the bottom of the IDE in the *Windows and Inspectors* section.

   Under the *Comm Window* tab, select the *ASCII Window* tab to see text messages sent from the PLC to the PC.

7. **Compile and Download *HelloWorld1.sal*.**

   In the IDE, press the *Compile/Download* toolbutton, or else click the *Project->Compile/Download* menu item.

   The IDE's LED will turn yellow and a progress bar will indicate that *HelloWorld1.sal* is being compiled and downloaded to the PLC. The IDE's LED will turn green when the download is completed.

   After the PLC receives the code, it will automatically reset and run the downloaded *HelloWorld1.sal* program from the beginning.

   The IDE's *ASCII Window* will display the text:

   ```
   Database Initialized
   Core App Running
   Core App Running
   ```

   NOTE: If the PLC's LED continues to flash on and off once per second after the download, the SALad program failed its checksum test. Recheck the serial connection and try the *Compile/Download* again.

8. **Test the *HelloWorld1.sal* program.**

   Tap the PLC's *SET Button* (located above the LED) to fire the *Button Tap* SALad event. The *HelloWorld1* SALad program will respond to the event by sending an ASCII message to the PC.

   The IDE's *ASCII Window* will display the text:

   ```
   Core App Running-Button Pressed
   ```

9. **Debug the *HelloWorld1.sal* program.**

   To the right of the toolbar, click the *Debug Mode* checkbox to begin a debugging session. A watch window will appear to the left of the editor pane. In debug mode, the PLC will report the address of its program counter to the IDE and optionally stop on each line, allowing you to debug the code.

   To try out the debugger, press the *Fast Step* toolbutton, and then tap the PLC's *SET Button* again. This time, the IDE will step rapidly through the program in the

editor and highlight each line of code that it executes.

You can cause the debugger to execute one line of code at a time by single stepping. Try pressing the *Single Step* toolbutton, and then tap the PLC's *SETt Button*. The editor will highlight the first line of SALad code to be executed. Thereafter, each time you press the *Single Step* toolbutton, the highlighted line of code will execute, then the editor will jump to and highlight the next line to be executed after that one.

You can set (soft) breakpoints in the code by clicking on the green dots in the margin at the left of the code. Active breakpoints are indicated by a red dot. To clear a breakpoint, just click it again.

Try using a breakpoint by single-stepping a few lines into the code and setting a breakpoint there. Press the *Fast Step* toolbutton to let the code finish executing, and then press the PLC's *SET Button* to fire a new event. The debugger will stop at the breakpoint you set.

To run the debugger as fast as it will go, press the *Run* toolbutton. Soft breakpoints will be ignored but code highlighting will still work. If you don't want the editor to jump around in the code as it executes, uncheck the *Show CP* check box (CP means Code Point).

To exit the debugging session and allow the PLC to run normally, simply uncheck *Debug Mode*.

10. **Congratulations, you have compiled, downloaded, tested and debugged a sample SALad program running on a PowerLinc Controller.**

    HelloWorld1.sal is built on a *SALad coreApp Program* called c*oreApp.sal* that provides basic INSTEON and X10 communication along with other essential features such as initialization, serial communication, and timers. You can find other sample SALad programs and templates in the *SALad IDE\Samples* and *SALad IDE\Code Templates* directories.

11. **Further steps.**

    You can become more familiar with INSTEON and X10 SALad programming by reading this Developer's Guide, trying out other sample SALad programs, and using the debugger. Since SALad programs mostly consist of event handlers, the easiest and safest way to write SALad code is to modify existing code, such as *coreUser.sal* that already contains the necessary infrastructure.

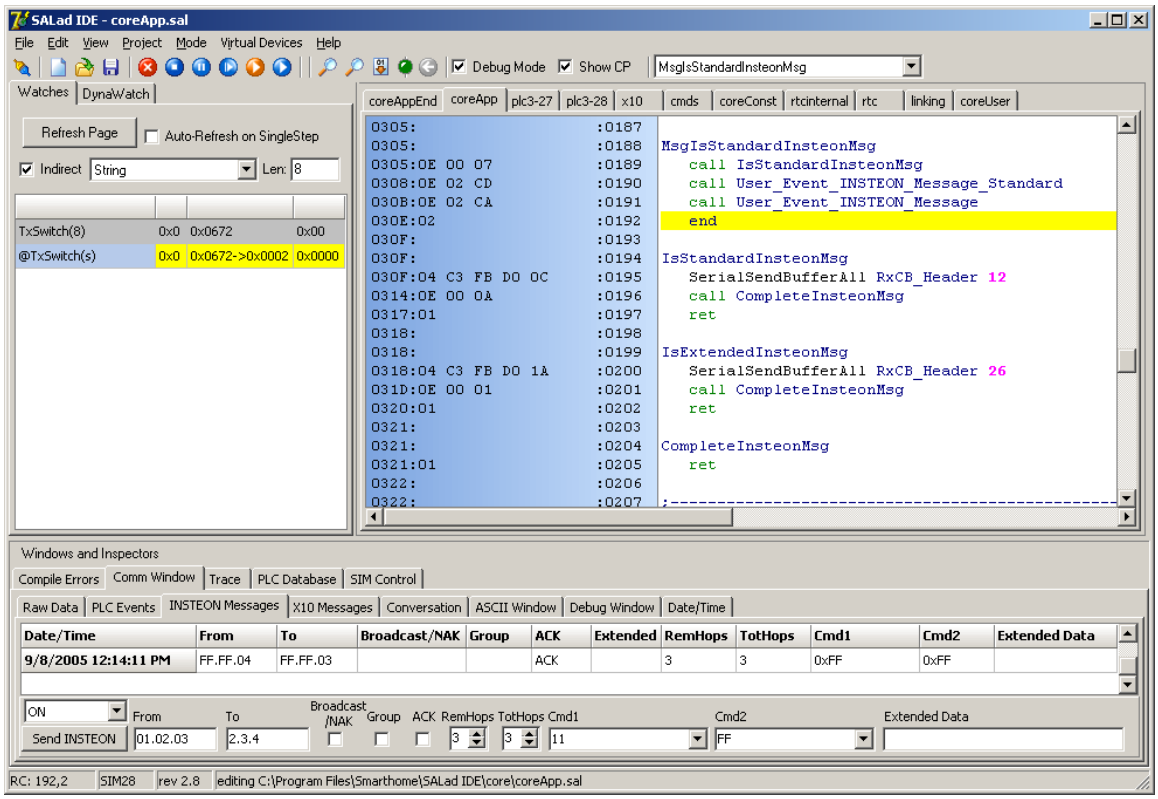
    Smarthome and other leading developers are continuously creating new sample and real-world SALad applications. Be sure to check the INSTEON Forum frequently at http://www.insteon.net/sdk/forum/ for the latest information.

IDE Main Window

This is what the SALad IDE's main window looks like. This section explains the *IDE Menus* and *IDE Toolbar* at the top.

The *IDE Editor* is below the toolbar on the right. During debugging, the *IDE Watch Panel* appears to the left of the Editor. You can access the various *IDE Windows and Inspectors* at the bottom by clicking on the tabs.

You can drag the borders at the left and bottom of the Editor to resize the panes, and you can instantly collapse or expand the Windows and Inspectors pane by clicking at the top of it.



IDE Menus

These are the main menus in the SALad IDE.

In This Section

Menu – File
Opens and saves source and output files.

Menu – Edit
Helps you navigate within your source code, find text, replace found text, and modify IDE options.

Menu – View
Changes the appearance of the IDE by displaying and hiding various parts of it.

Menu – Project
Controls compilation options and loading of new firmware into the PLC.

Menu – Mode
Switches between using a real PLC, a simulated PLC, or no PLC.

Menu – Virtual Devices
Launches the virtual powerline and virtual LampLinc devices for use with the PLC Simulator.

Menu – Help
Launches this Developer's Guide in compiled help form, gives version information about the IDE, lets you check for IDE updates, and links you to the INSTEON Developer's Forum.

Menu – File

The *File* menu is for opening and saving source and output files.

or **Toolbutton**  Creates a new source code file in the editor. A new blank file will appear under a new tab in the editor, labeled *unnamedx*, where *x* will increment as necessary to provide a unique filename. To save the new file under the name you really want it to have, use *Save As…*.

or **Toolbutton**  Opens an existing source code file in the editor.

This is the same as *Open…* except the submenu lets you choose from a list of the last files you opened.

or **Toolbutton**  **(Keyboard Shortcut: Ctrl-S)** Saves the file currently displayed in the editor using the same name as shown in the active tab (with an extension of *sal*), in the same directory from which it was opened.

Saves the file currently displayed in the editor, using whatever name you choose, in whatever directory you choose.

Saves all of the files under all tabs in the editor in the same way that an individual file would be saved.

This compiles your SALad program and saves it as a binary file (with an extension of *slb*) in whatever directory and with whatever name you specify.

This compiles your SALad program and saves it as a hexadecimal file (with an extension of *hex*) in whatever directory and with whatever name you specify.

This is a very convenient option for saving your work and optionally sending it to someone else (or yourself). It will create a Zip file containing your source code, then launch your emailer with the Zip file already attached to a default message. If you don't want to email the Zip file, just close the emailer.

(Keyboard Shortcut: Ctrl-F4) Closes the file currently displayed in the editor. This will not delete the file saved on disk (if there is one).

Closes all of the files currently open in the editor. This will not delete any of the files saved on disk (if they exist). After all of the files are closed, a new blank file will appear under a new tab in the editor, labeled *unnamedx*, where *x* will increment as necessary to provide a unique filename.

Launches the *Options* dialog box, which lets you change various settings for the IDE. See IDE Options Dialog Box for details.

This compiles your SALad program and saves it as a binary file (with an extension of *slb*) in whatever directory and with whatever name you specify.

This saves the currently displayed source file in the editor as a listing file (with the extension *txt*). The listing includes the information in the 'gutter' at the left of the editor, which includes hex addresses, bytecode and line numbers.

This ends your IDE session and closes the program. Be sure to save your work if you did not specify the *Save all files on close* option in the *Options – Saving* dialog box.

Menu – Edit

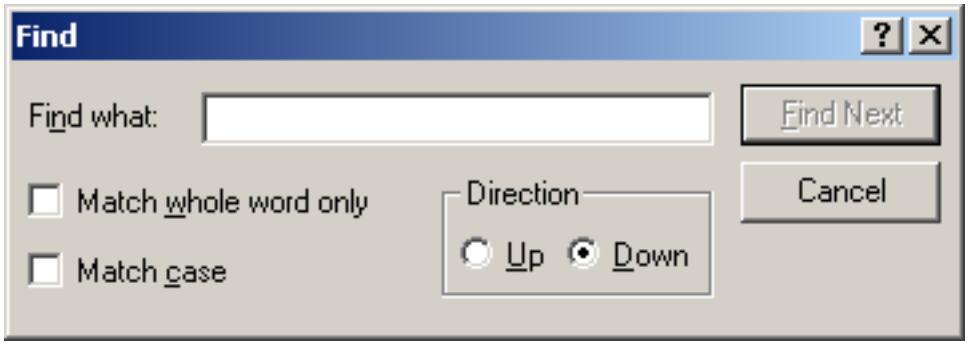
The *Edit* menu helps you navigate within your source code, find text, replace found text, and modify IDE options.

(Keyboard Shortcut: Alt-G) This lets you jump to a specified line number in the source file currently displayed in the editor. The editor will display the line highlighted for a few seconds after it jumps to it. If you specify a line number beyond the end of the file, the editor will jump to the last line in the file.

If this option is checked, you can edit source files. To disable editing, uncheck this option.

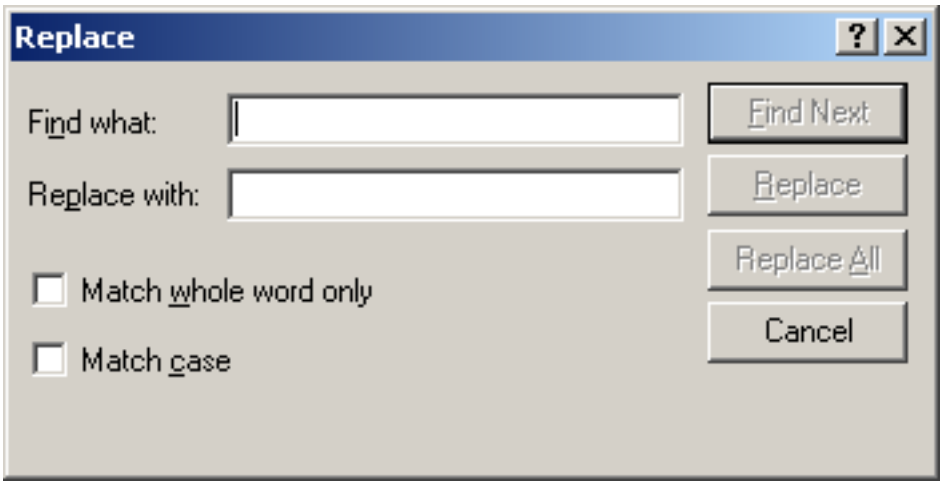
Launches the *Options* dialog box, which lets you change various settings for the IDE. See IDE Options Dialog Box for details.

or **Toolbutton**  **(Keyboard Shortcut: Alt-F)** Launches the *Find* dialog box, which looks like this:



You cse **Ctrl-Down** to find the next occurrence of the string you are finding, and **Ctrl-Up** to find the previous occurrence.

or **Toolbutton**  **(Keyboard Shortcut: Alt-R)** Launches the *Find and Replace* dialog box, which looks like this:



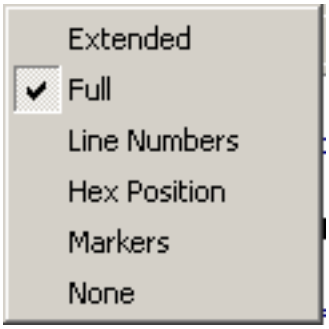
You can use **Ctrl-Down** to find the next occurrence of the string you are finding, and **Ctrl-Up** to find the previous occurrence.

Menu – View

The *View* menu changes the appearance of the IDE by displaying and hiding various parts of it.

Check or uncheck this option to display or hide the *Windows and Inspectors* tabs at the bottom of the IDE window. You can achieve the same result by clicking on the *Windows and Inspectors* label bar.

This gives a submenu that changes the appearance of the information to the left of the source code in the editor. The submenu looks like this:



You can check only one of the options. *Full*, the default, displays (from left to right) hex addresses, hex bytecode, source code line numbers, and breakpoint markers, as shown in the section *IDE Editor*, below. If you check *Extended*, the gutter pane is widened so you can see more of the bytecode. If you check any of the other options, so will only what you selected. During debugging, you will not be able to set breakpoints unless the *Markers* column is displayed.

This launches the *Startup Wizard* that guides you through setting up communications with your PLC. You can achieve the same results manually by going to the *Options – Communications* menu.

Menu – Project

The *Project* menu controls compilation and loading of new firmware into the PLC.

(Keyboard Shortcut: F9 while not debugging)
This compiles your SALad program and downloads the compiled bytecode to your PLC (which can be a real PLC or the PLC Simulator). The Compile/Download toolbar button does the same thing.

This lets you download a selected version of firmware code to the PLC. Firmware object code has the file extension *ump*.

Menu – Mode

The *Mode* menu switches between using a real PLC, a simulated PLC, or no PLC. You can select only one of the options.

Select this option if you are connected to a real PLC. This option is the default.

Select this option if you are not connected to a PLC. You will not be able to download or debug code while you are offline.

Select this option if you wish to use the software PLC Simulator. You can download code to the PLC Simulator and debug it just as you would for a real PLC. See the

PLC Simulator section below for more information. While you are using the PLC Simulator you can also simulate a *Virtual Powerline* environment along with any number of *Virtual LampLinc* devices plugged into it.

Menu – Virtual Devices

The *Virtual Devices* menu launches the virtual powerline and virtual LampLinc devices for use with the PLC Simulator.

This will launch a virtual LampLinc device simulated in software. The virtual LampLinc will appear in a separate window. See *Virtual LampLinc* for details.

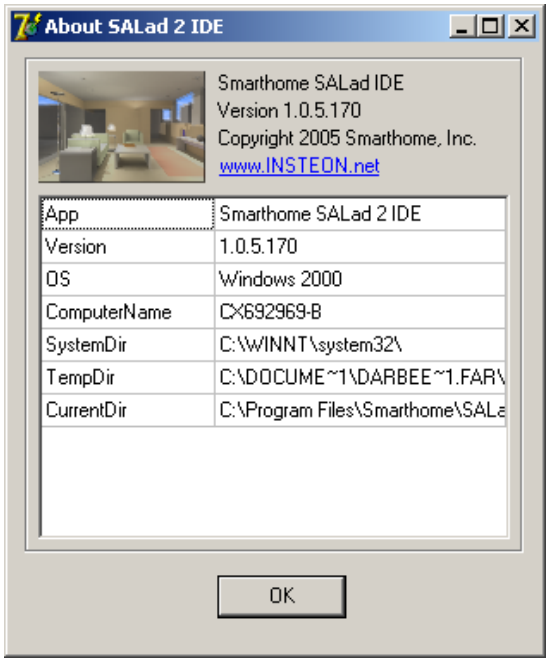
This will launch a virtual powerline environment simulated in software. The virtual powerline will appear in a separate window. If you are going to use the virtual LampLinc, launch a virtual powerline first. See Virtual Powerline for details.

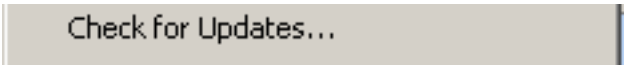
Menu – Help

The *Help* menu launches this Developer's Guide in compiled help form, gives version information about the IDE, lets you check for IDE updates, and links you to the INSTEON Developer's Forum.

This launches this INSTEON Developer's Guide in compiled help format.

This displays a screen like this one:



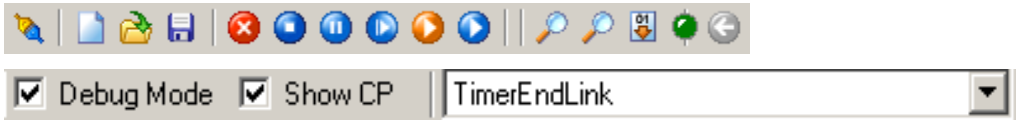


This will automatically check for newer versions of the IDE and update it if appropriate.

This will open your web browser and take you to www.insteon.net.

IDE Toolbar

This is what the main Toolbar looks like:



This is what the individual toolbuttons do:

Connect or Reconnect serially to the PLC.

New File. Create a new SALad file (*.sal).

Open File. Open a SALad file (*.sal) or SALad Template (*.salt).

Save File. Save the current SALad file (*.sal) or SALad Template (*.salt).

Reset. Reset the PLC by forcing an **EVNT_INIT (0x00)** IBIOS Event (see *IBIOS Event Details*, Note 1). This will start the SALad program with **EVNT_INIT (0x00)** in its event queue.

Stop. Stop execution of the SALad program.

Pause. During debugging only, pause the SALad program at the next line of code.

Single Step. During debugging only, execute the next line of code in the SALad program. This line is normally highlighted in the IDE.

Fast Step to Next Breakpoint. During debugging only, execute one line of code at a time, reporting each line executed, and stop at the next soft or hard breakpoint.

Run in Animate Mode. During debugging, run at top debugging speed, reporting each line executed, and stop at the next soft or hard breakpoint or at the end of the program. During normal execution, run at full speed and stop at the next hard breakpoint or at the end of the program.

Find. Find text in the currently displayed SALad program.

Find and Replace. Find and replace text in the currently displayed SALad program.

Compile and Download. Compile the SALad program and download the bytecode into the PLC over the serial port. If you right-click this button, the following options appear:

1. 

Display differences between the code to download and the code read back from the PLC.

2. 

Download all files on the next download instead of just incrementally downloading files that have changed.
3. 

Always download all files. This is the default setting.
4. 

Read back the code image from the PLC after the next code download.

 **LED**. This is an indicator only, not a button. After pressing the  *Connect/Reconnect* button, green means you are successfully connected to the PLC and red means you are not connected. After pressing the  *Compile/Download* button, green means the program downloaded to the PLC successfully, red means the program did not download successfully, and yellow means the program is currently downloading.

 **Go Back**. Return to the previous SALad program location in the editor. Use this after hot-jumping to another SALad program by right-clicking on a label.

There are two checkboxes, used for debugging:

Enable/Disable Debugging Mode. When checked, you will be in debugging mode and the Watch window will appear to the left of the edit window. You can change the size of the Watch window by dragging the separator bar.

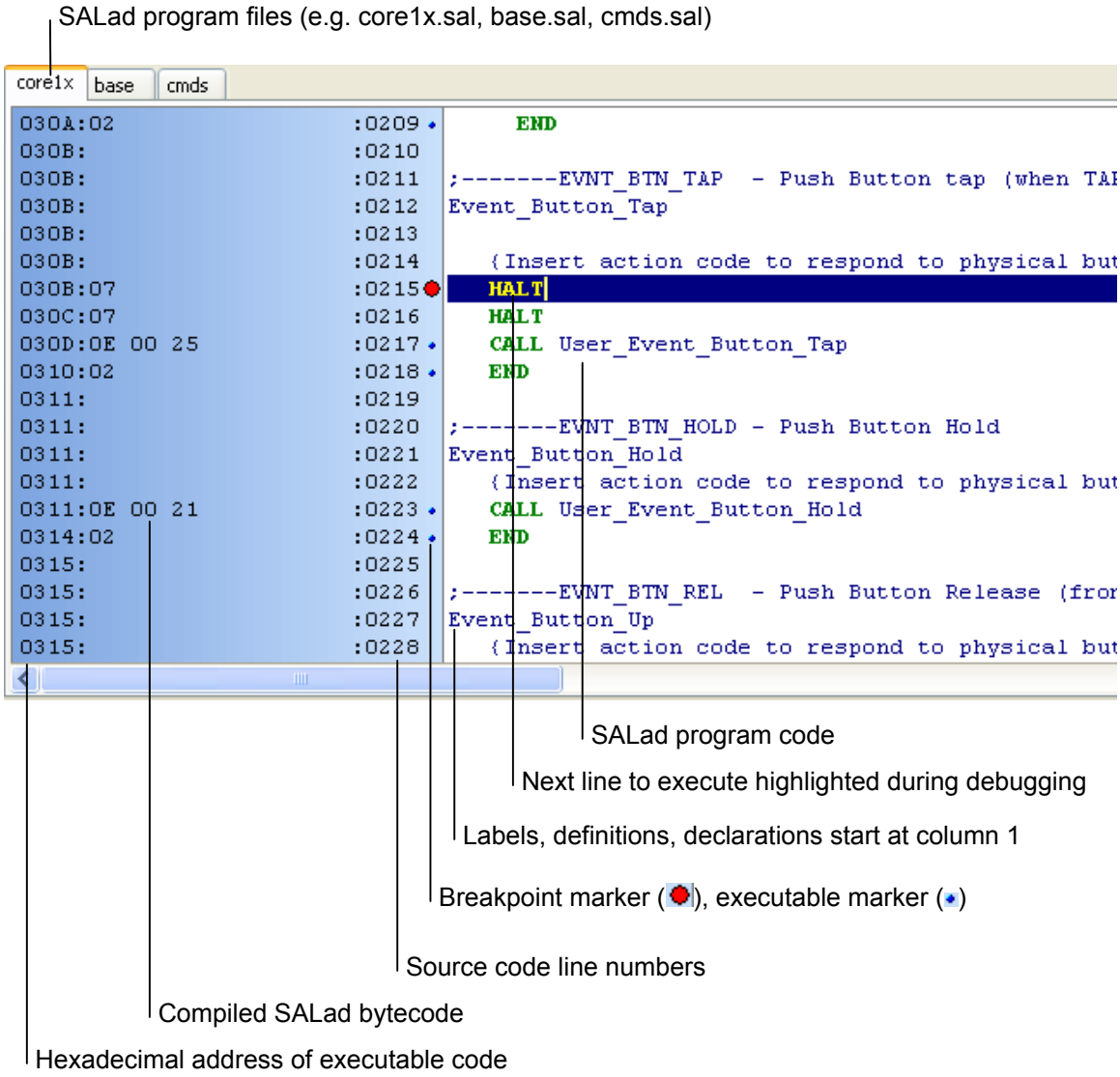
Show Code Point. When you are debugging and *Show CP* is checked, the editor will jump to whatever code line is executing and highlight that line, even if the line is not on the currently displayed page. When *Show CP* is unchecked, the currently displayed page will remain visible during execution.

The pulldown box works with the editor:

Jump To Label. This pulldown box contains all labels used in the SALad program. When you select a label from the pulldown box, the editor will jump to the program location where the label first appears. During editing and debugging, the label displayed in the pulldown box will be the one that fits the current context.

IDE Editor

The IDE editor handles multiple source files and displays SALad source code in context-sensitive color. The screenshot below shows how an editing session would typically appear.



You can change the appearance of the editor by right-clicking anywhere in the editor pane and selecting *Editor Properties…*. This displays the *Editor* tab in the *Options* dialog box. See the *Options – Editor* section for details.

You can change which columns of information are displayed in the 'gutter' using the *View->Gutter* menu item (see *Menu – View* above).

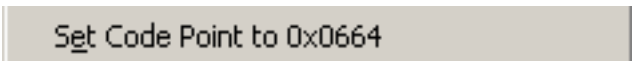
Right-clicking anywhere in the editor displays the following menu options:

(Keyboard Shortcut: Ctrl-C) This copies selected text to the clipboard.

(Keyboard Shortcut: Ctrl-X) This cuts selected text to the clipboard.

(Keyboard Shortcut: Ctrl-V) This Pastes text in the clipboard at the cursor location.

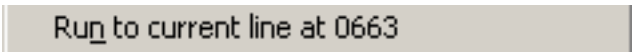
(Keyboard Shortcut: Ctrl-A) This selects all of the text in the current file.



This sets the program counter so that the next line to be executed will be where the cursor is, i.e. where you right-clicked to get this popup menu.

This sets the program counter so that the next line to be executed will be where the caret is. The caret (a vertical bar) is where you last left-clicked clicked with the cursor.

Run the program from the Code Point (the current address of the program counter) to the line where the cursor is, i.e. where you right-clicked to get this popup menu.



Run the program from the Code Point (the current address of the program counter) to the line where the caret is. The caret (a vertical bar) is where you last left-clicked clicked with the cursor.

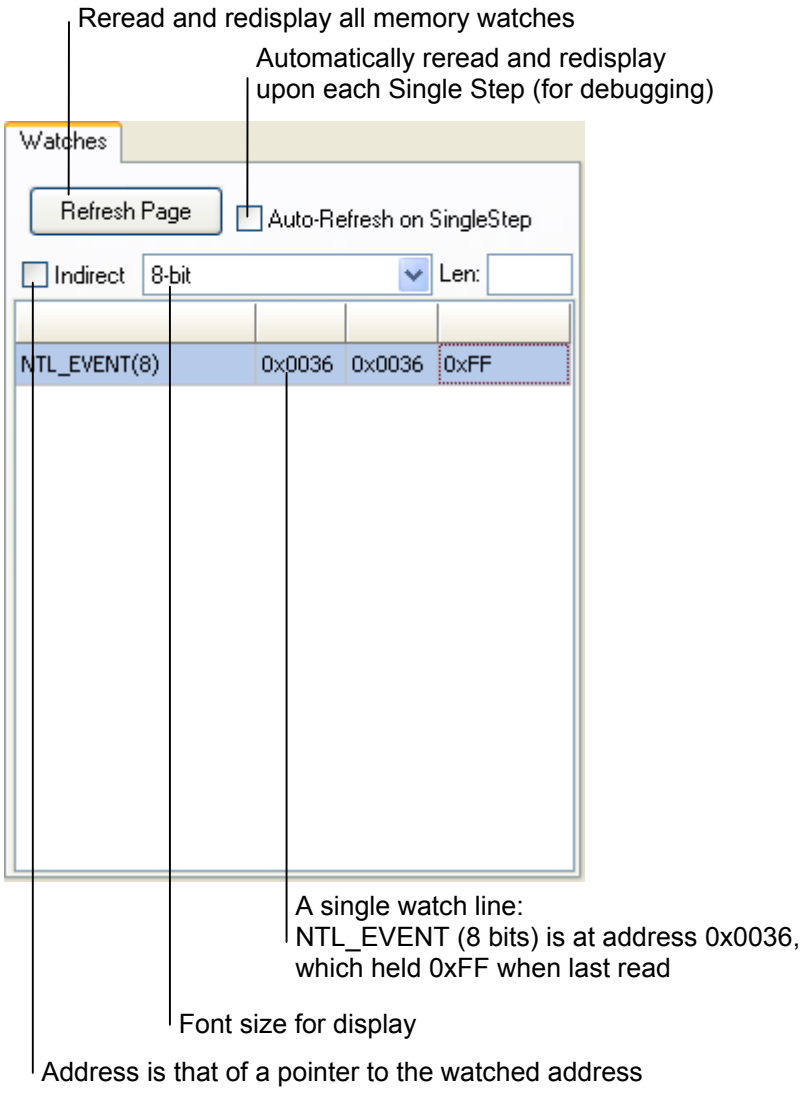
Add the variable under the caret to the Watch Window. The caret (a vertical bar) is where you last left-clicked clicked with the cursor. See *IDE Watch Panel* for details.

If there are any undefined labels in your SALad program, this will insert them at the cursor position so you can define them.

This displays the *Editor* tab in the *Options* dialog box, so you can change the way the editor looks and behaves. See *Options – Editor* for details.

IDE Watch Panel

The watch panel appears to the left of the editor during debugging sessions. You can change its size by dragging the border between it and the editor.



You can add variables that you want to watch by clicking on a variable in the editor, then right-clicking and choosing *Add Watch* in the popup menu.

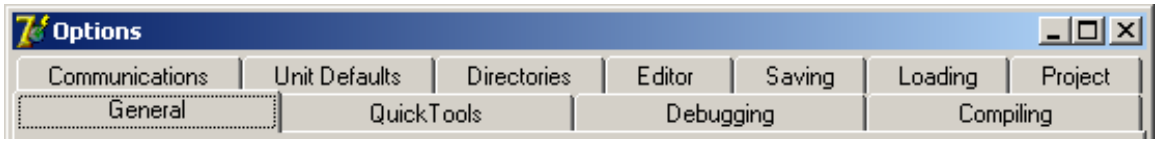
Click the *Refresh Page* button to update all watches. If the *Auto-Refresh on SingleStep* check box is checked, then single stepping during debugging (keyboard shortcut F8) will update the watches after each step.

Check *Indirect* if the variable that you are watching contains a pointer to another variable. The Watch Window will display then also display the contents of the variable being pointed to.

The pulldown box lets you choose whether the watched address is the beginning of an 8-bit variable, a 16-bit variable, an ASCII string, or a hex string. If it is an ASCII or hex string, set the length of the string to display in the *Len:* box.

IDE Options Dialog Box

The Options Dialog Box has a number of tabs that look like this. The panels that the various tabs display are explained below. You can launch the *Options* Dialog Box by choosing the *Edit->Applications Options…* menu item.



The *OK* and *Cancel* buttons appear at the bottom of each panel, but they are not shown in the figures below. Settings take effect when you push *OK*. If you push *Cancel* the previous settings will remain in effect.

In This Section

Options – General
Controls overall IDE behavior.

Options – Quick Tools
A set of tools for working with the PLC's firmware and Link Database.

Options – Debugging
Contains controls for debugging sessions.

Options – Compiling
Contains controls for the compiler.

Options – Communications
Has setup options and tools to control communication with the PLC.

Options – Unit Defaults
Controls settings for the PLC firmware.

Options – Directories
Sets file search paths.

Options – Editor
Controls how the editor looks and behaves.

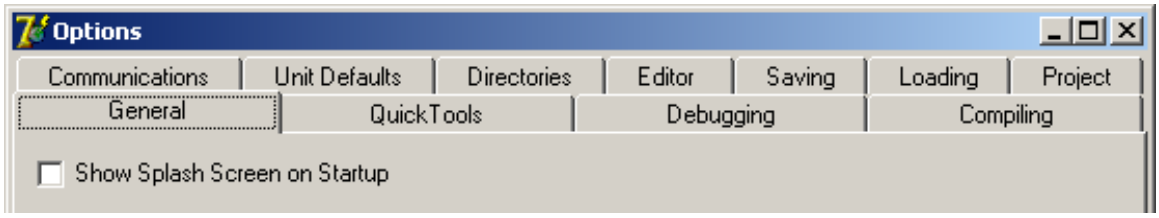
Options – Saving
Sets file saving and backup options.

Options – Loading
Sets file loading options.

Options – Project
Designates the file containing the beginning of your SALad program.

Options – General

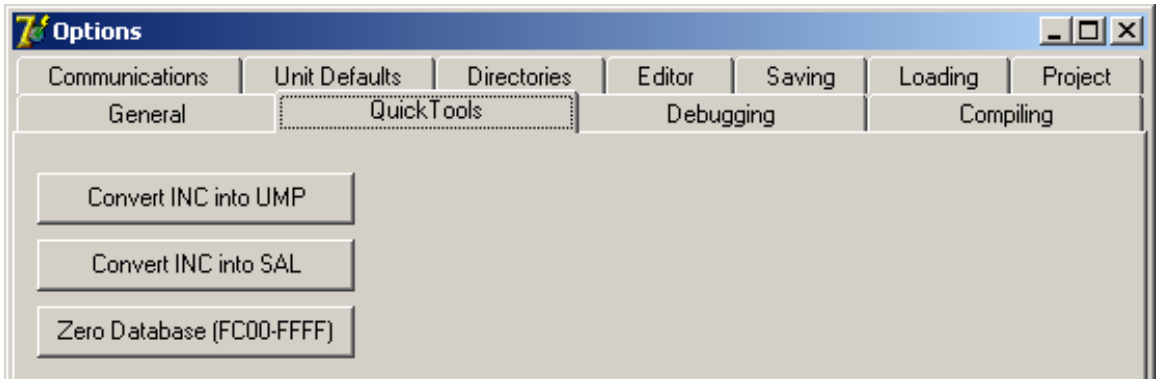
The *General* tab controls overall IDE behavior.



Currently the only option is to show the Splash Screen when the program starts. This defaults to off after the first time the program is run.

Options – Quick Tools

Quick Tools are for working with the PLC's firmware and Link Database. These are here for convenience and others may be added in the future.



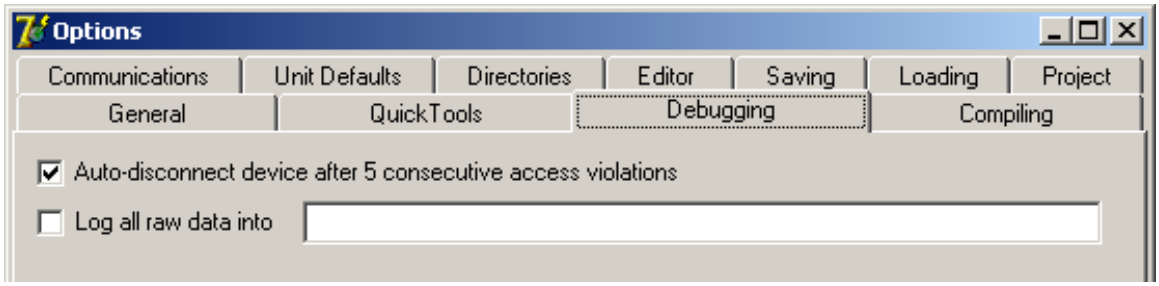
Convert INC into UMP and *Convert INC into SAL* are utilities for converting assembler include (INC) files into Unit Map files (UMP) or SALad source code files (SAL).

Zero Database (FC00-FFFF) clears the PLC Link Database from hex address 0xFC00 to 0xFFFF by writing zeros into it. You can also zero the Link Database from the

PLC Database tab under *Windows and Inspectors*.

Options – Debugging

The *Debugging* tab contains controls for debugging sessions.



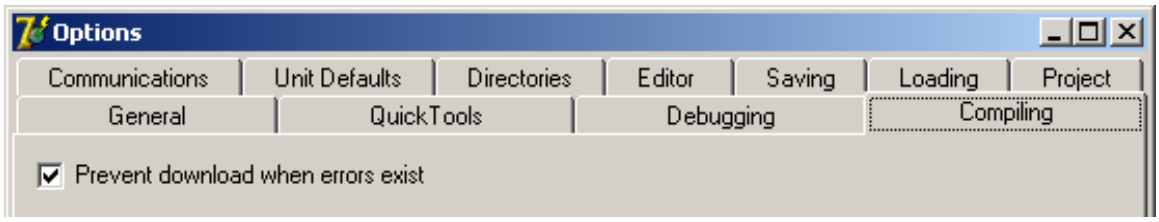
Auto-disconnect device after 5 consecutive access violations stops serial data flow if faulty SALad code is generating an access violation storm that prevents you from seeing where the code went awry.

Log all raw data into <specified log file> lets you designate a text file that will log the data that appears in the

Comm Window – Raw Data panel.

Options – Compiling

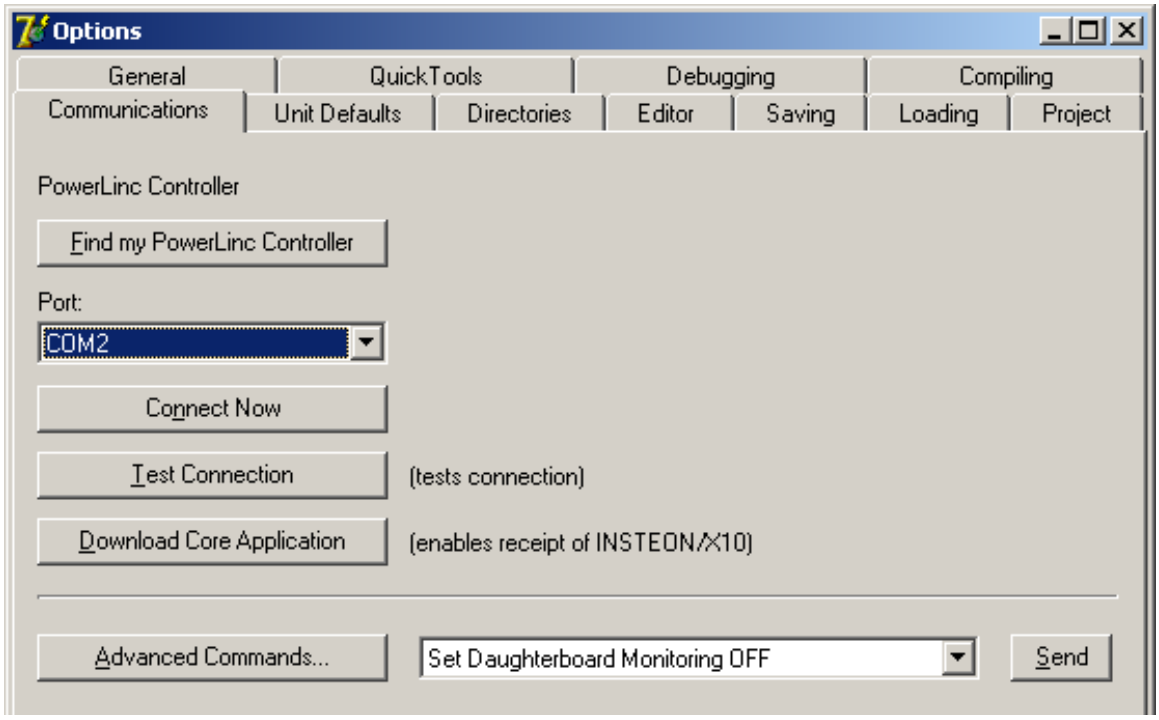
The *Compiling* tab contains controls for the compiler.



Prevent download when errors exist, when checked, will not download bytecode to the PLC if the compiler generates errors. This option is checked by default.

Options – Communications

The *Communications* tab has setup options and tools to control communication with the PLC.



Press *Find my PowerLinc Controller* to check your serial ports for an attached PLC. If a PLC is found, the port to which it is attached will appear in the *Port:* pulldown box.

If you want to manually designate the serial port that your PLC is attached to, you can use the pulldown box directly. The pulldown box contains options for your available COM (RS232) and USB ports. If you are using the PLC Simulator it does not matter which option you choose.

Press the *Connect Now* button to establish a connection with your PLC over the port designated in the *Port:* pulldown box. Pushing the *Connect Now* button will automatically "push" the *Test Connection* button.

If you are already connected to your PLC, you can test the connection at any time by pressing the *Test Connection* button.

Press *Download Core Application* to download a precompiled version of the *coreApp.sal* *SALad coreApp Program* to your PLC. This file to be downloaded, named *coreApp.slb*, is the one in the *Program Files\Smarthome\Salad IDE\core* directory.

Press *Advanced Commands...* to toggle the appearance of a pulldown box containing PLC command options that only apply to particular versions of the PLC. The command shown, *Set Daughterboard Monitoring OFF* is for a Hardware Development Kit. Press the *Send* button to actually send the chosen command to the PLC

Options – Unit Defaults

The *Unit Defaults* tab controls settings for the PLC firmware.

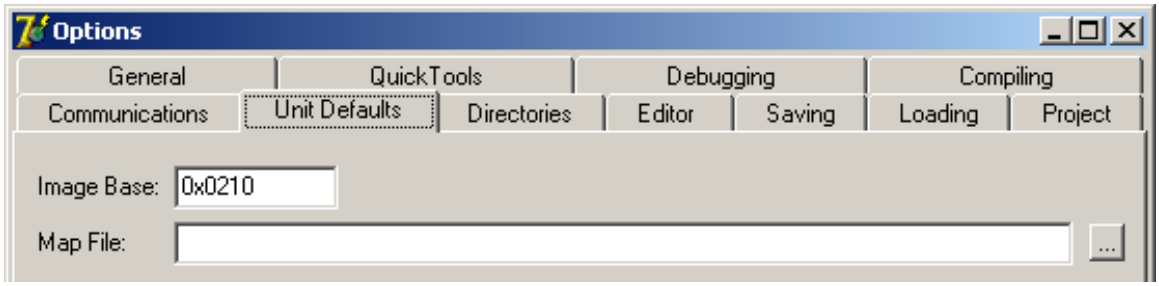
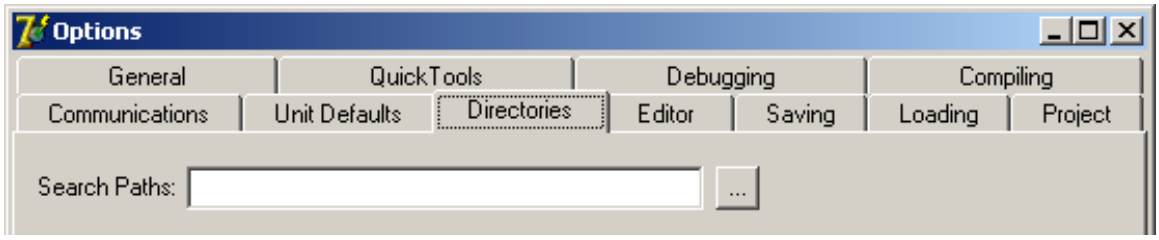


Image Base: is the base address for the downloaded compiled SALad bytecode. This should match the 'org' of the first executable code in the file you designate as 'main' in the *Options – Project* tab. The default is 0x0210.

Map File: lets you designate a unit map (UMP) file other than the one that the IDE defaults to by auto-sensing which firmware version is running in your PLC.

Options – Directories

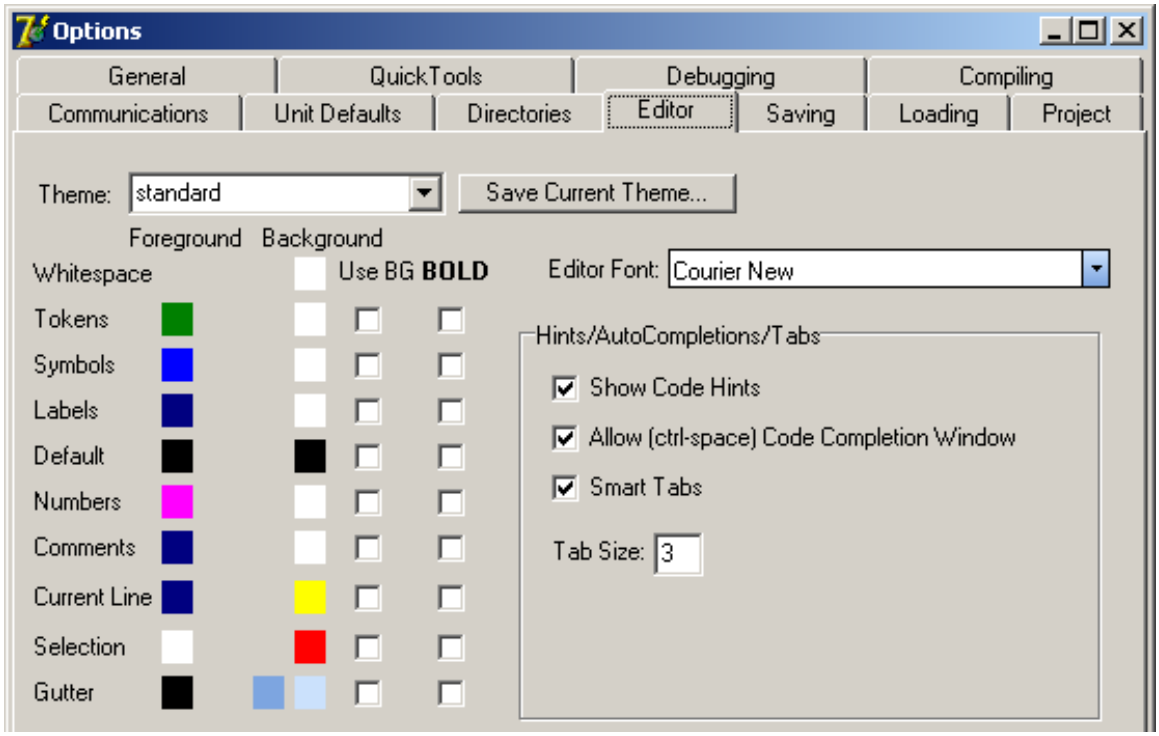
The *Directories* tab sets file search paths.



Enter the path to files that you would like the IDE to search for to be included in your project. Press the button to open a standard dialog box to find a directory. If you want multiple search paths, separate them with a semicolon. The default search paths are your current project's directory (usually under *Smarthome\SALad IDE\Projects*), then the *Smarthome\SALad IDE\Include* directory, then the *Smarthome\SALad IDE\Core* directory.

Options – Editor

The *Editor* tab controls how the editor looks and behaves. You can also launch this page directly from the editor by right-clicking anywhere in it and choosing *Editor Properties…*.



Use the *Theme:* pulldown box to select a previously saved collection of the color and font settings on this page. The default theme is called *standard*. You can create custom themes by choosing the editor options you want on this page and then pressing *Save Current Theme…* to store the settings under a name you choose. Whenever you launch the IDE, the editor will use the theme that last appeared in the pulldown box.

To change the foreground or background color for text of a given type, click on the color box and choose the color you want.

Check the *BOLD* box if you want text of the given type to appear bold, and check the *Use BG* box if you want the specified background color to actually be used.

Use the *Editor Font:* pulldown box to choose the font for the editor.

Check *Show Code Hints* if you want tooltips to appear as you type SALad instructions. The tooltips show the parameters for the instruction and a brief description of what it does.

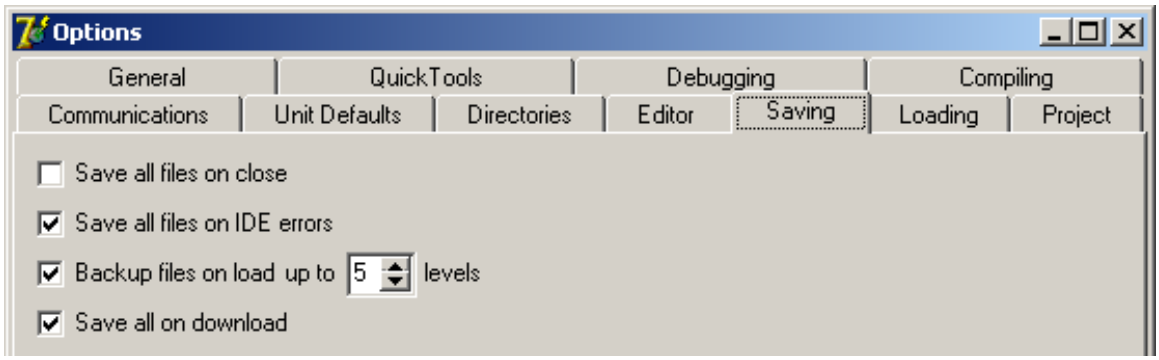
Check *Allow (ctrl-space) Code Completion Window* if you want to see possible SALad word completions by pressing Control-Space after you've typed a partial word.

Check *Smart Tabs* if you want the editor to automatically indent or outdent the next line, depending on context, when you press Enter at the end of a code line.

Set the number of spaces for a tab in the *Tab Size:* box. The default is 3. Tabs are converted to spaces in saved files.

Options – Saving

The *Saving* tab sets file saving and backup options.



Check *Save all files on close* if you want the IDE to automatically save all open source code files when you close the IDE. The files will be saved in whatever condition they are in at the time, so be careful when using this option. The default is unchecked.

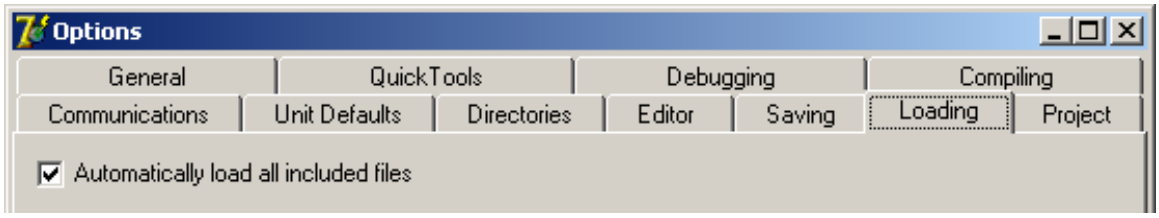
Check *Save all files on IDE errors* if you want the IDE to automatically save all open source code files whenever an error occurs in the IDE program. Then, if the IDE crashes, when you restart it you will get a dialog box informing you that your files were auto-recovered from backups, and giving you the option of keeping the auto-recovered files or not. The default for this option is checked.

Check *Backup files on load up to N levels* if you want the IDE to automatically save source code files with a *bak* extension whenever they are loaded. If you set the number of levels greater than 1 (the default is 5), then *bak* files are first renamed with an extension *bk1*, *bk1* files are renamed *bk2*, and so forth. This option is checked by default. **NOTE:** This option has not been implemented as of version 1.0.5.170 of the IDE.

Check *Save all on download* if you want the IDE to automatically save all open source code files whenever you download compiled code to your PLC program. The default is checked.

Options – Loading

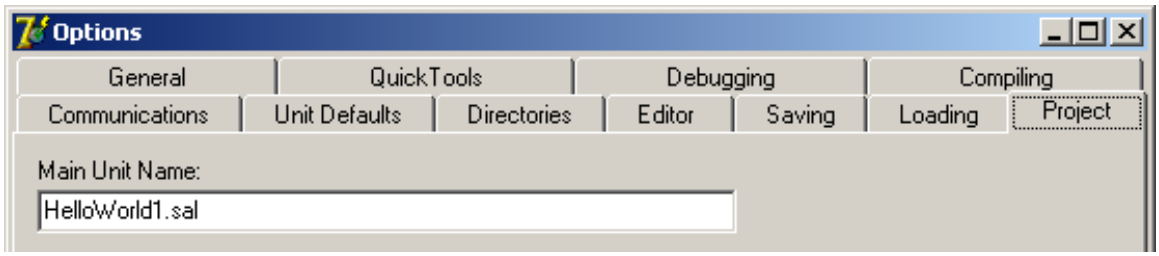
The *Loading* tab sets file loading options.



Check *Automatically load all included files* if you want the IDE to find and load any files that you have named in a source file using an `include "<filespec>"` statement. The default is checked.

Options – Project

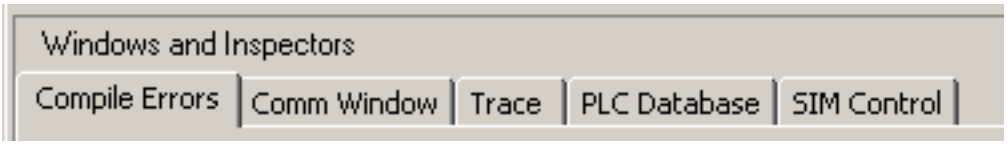
The *Project* tab lets you designate the file containing the beginning of your SALad program. If you do not do this, the compiler will not know where program execution begins. This setting is persistent, meaning that the IDE will remember it between sessions.



You can either type the name of the file in the text box, or you can right-click on the file's tab in the editor and choose *Mark as main unit*.

IDE Windows and Inspectors

Windows and Inspectors is a collection of tools that you will find very useful when using the IDE to create INSTEON applications using SALad. The main tools appear under a set of tabs that look like this:



Choosing *Comm Window* will display a series of subtabs with more tools.

You can make the *Windows and Inspectors* pane collapse or expand quickly by clicking anywhere in its title bar. You can resize the pane by dragging its top border.

In This Section

Compile Errors
: Lists errors the compiler finds and lets you jump to them in the editor for debugging.

Comm Window
: Has a collection of subtabs that help you to see what is going on in the INSTEON environment.

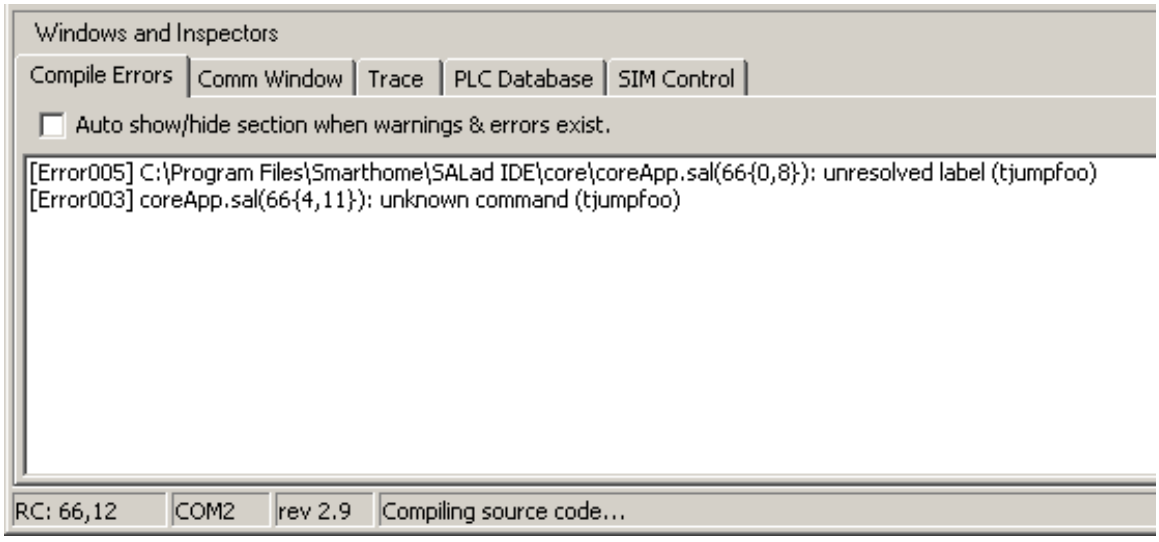
Trace
: Lets you inspect the execution history of your program to help you with debugging.

PLC Database
: A tool for manipulating the Link Database in your PLC.

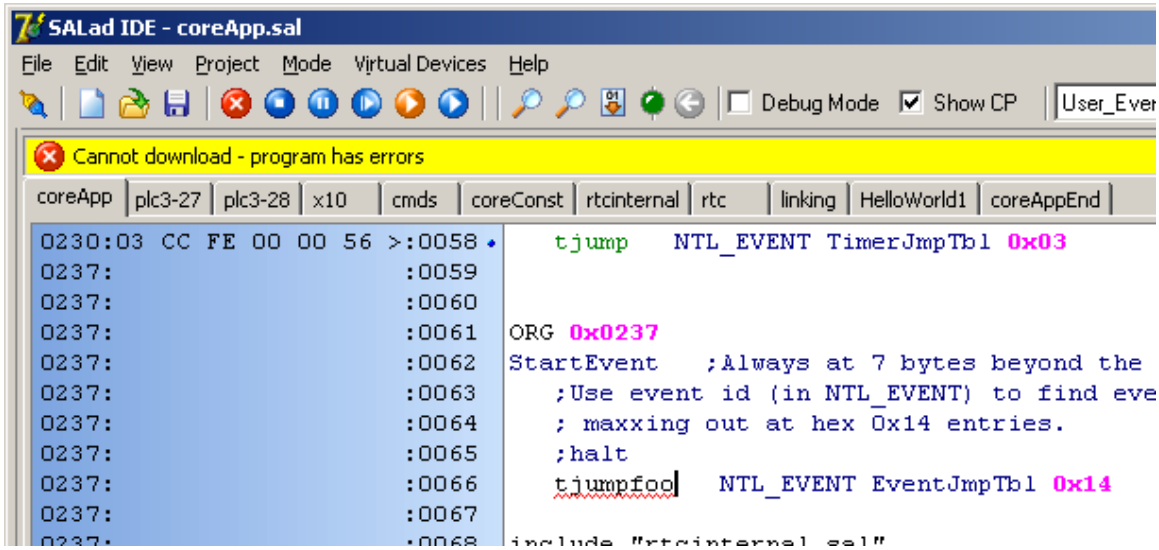
SIM Control
: Operates the PLC Simulator.

Compile Errors

This page shows errors that the compiler finds. Each line displays an error number, the filespec of the file containing the error, the line number and column positions of the error within the file, and an error message. Double-clicking an error places you on the offending line in the editor.

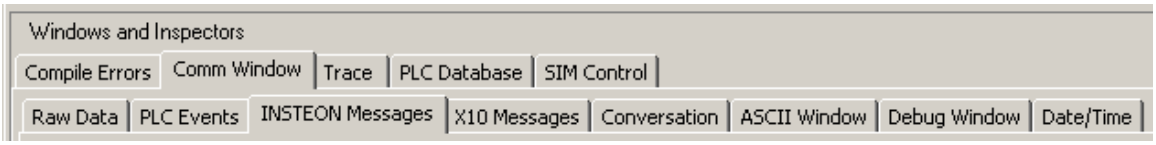


In the example above, an error was deliberately induced by changing a valid label (`tjump`) to an invalid one (`tjumpfoo`). Double-clicking on one of the lines above shows the error in the editor like this:



Comm Window

The Comm Window has a number of sub-tabs that look like this:



In This Section

Comm Window – Raw Data
Shows the serial data exchanged between the PLC and your PC.

Comm Window – PLC Events
Displays PLC Events that have occurred.

Comm Window – INSTEON Messages
Displays received INSTEON messages, and it lets you compose and send INSTEON messages of your choosing.

Comm Window – X10 Messages
Displays received X10 commands, and it lets you compose and send X10 commands of your choosing.

Comm Window – Conversation
Allows you to have a two-way serial conversation with the PLC.

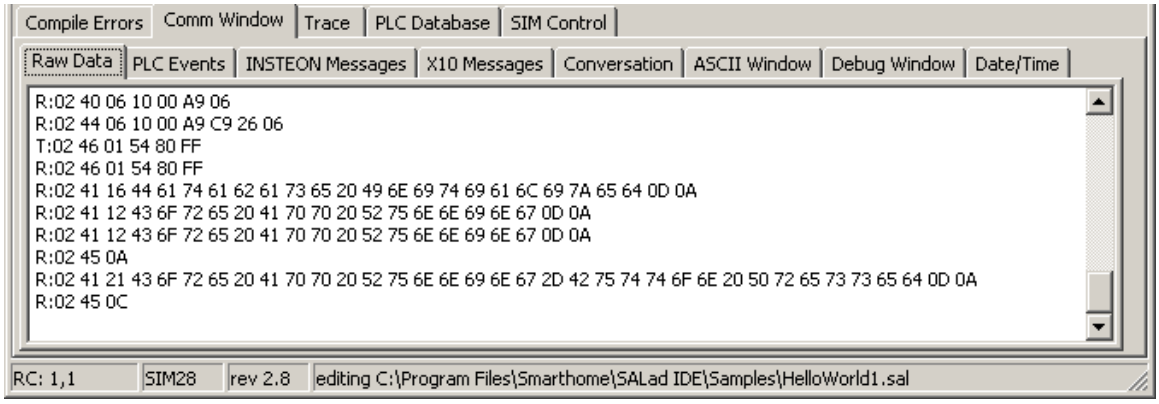
Comm Window – ASCII Window
Displays text explicitly sent from a SALad 2 program running on your PLC.

Comm Window – Debug Window
Lets you directly inspect and alter bytes within your PLC.

Comm Window – Date/Time
Gives you control over the realtime clock in the PLC and lets you test realtime events.

Comm Window – Raw Data

The *Raw Data* page shows the serial data exchanged between your PLC and your PC.



The data is displayed as hexadecimal bytes.

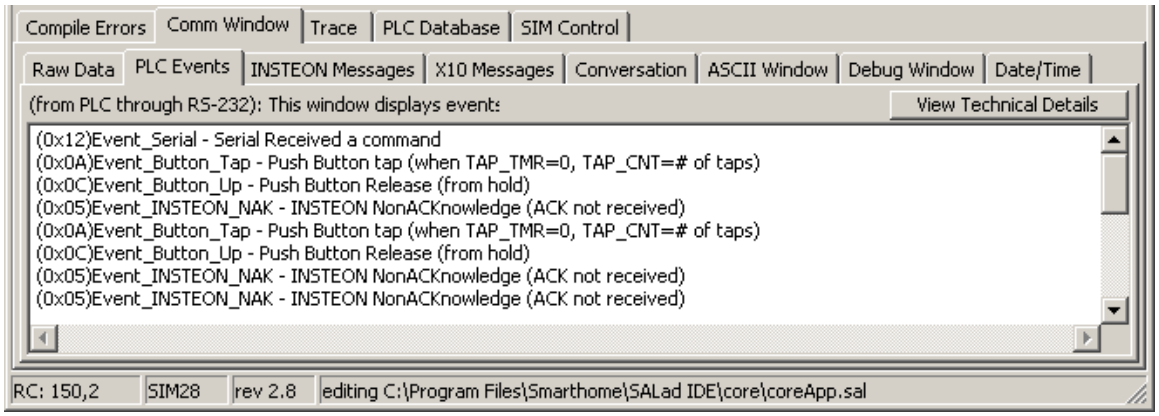
A **T:** precedes data transmitted from the PC to the PLC.

An **S:** precedes data transmitted from the PC to the PLC, but with any packet formatting that may have been applied also displayed. In particular, USB communications adds formatting as described in the section *IBIOS USB Serial Interface* above.

An **R:** precedes data received by the PC from the PLC.

Comm Window – PLC Events

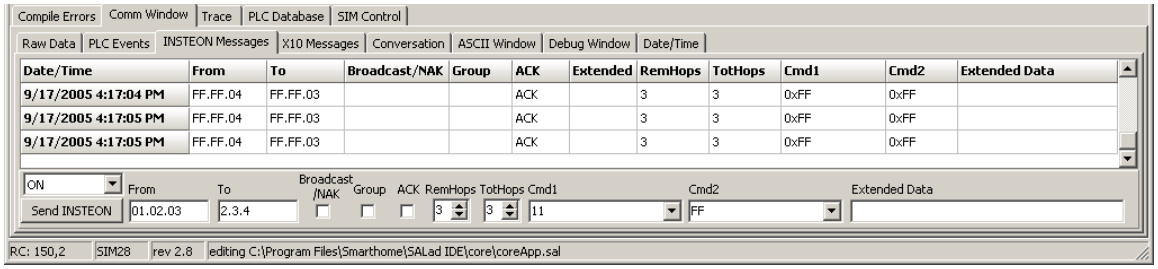
SALad is event-driven, meaning that SALad programs respond to events that occur in the host environment. This page displays IBIOS Events that have occurred in the PLC. See *IBIOS Events* for a list of events that SALad handles.



The left column displays the event number in hexadecimal, followed by the event name and a description of the conditions that caused the event.

Comm Window – INSTEON Messages

The *INSTEON Messages* page displays received INSTEON messages, and it lets you compose and send INSTEON messages of your choosing.



The top section of this page displays INSTEON messages received by the PLC. See *Message Fields* above for a description of the column headings.

The bottom section lets you compose and send an INSTEON message. You can choose from a number of pre-composed commands using the pulldown box at the left.

You can enter *From* and *To* *Device Addresses* in the text boxes by typing the 3-byte address as three decimal numbers separated by periods. This is similar to the way an IP address is specified on the Internet.

You can set the *Message Type Flags* the message using the check boxes.

Use the *RemHops* and *TotHops* boxes to set the *Message Retransmission Flags*.

Enter the *Command 1 and 2* that you want to send as hexadecimal bytes in the respective boxes.

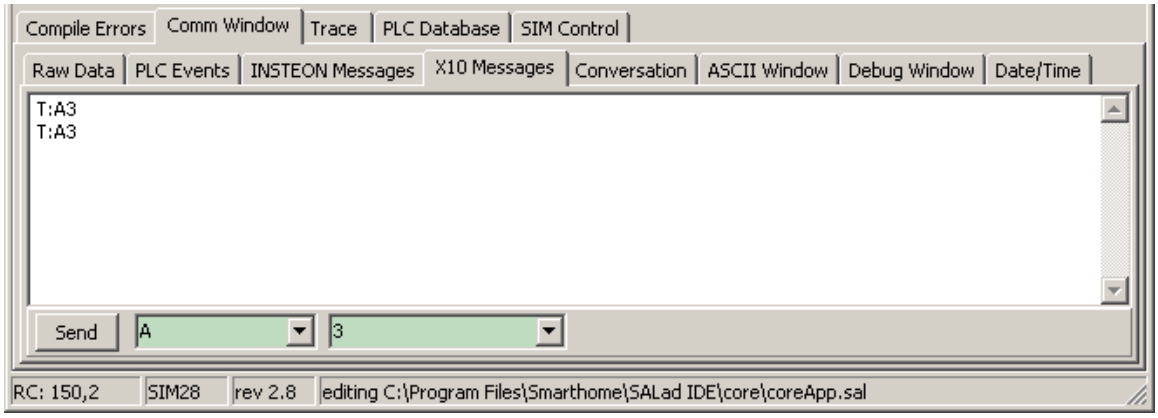
If you are sending an Extended Message, enter the *User Data* in the *Extended Data* text box. If you enter *any* data in this box, an Extended Message will be sent. If you enter more than 14 bytes, only the first 14 bytes will be sent. If you enter fewer than 14 bytes, the remaining bytes will be sent as 0x00. If you want to send a Standard Message, clear this box.

You do not have to deal with the *Message Integrity Byte* (CRC) because the INSTEON Engine handles this for you.

When you have composed the INSTEON message that you want, press the *Send INSTEON* button to transmit it.

Comm Window – X10 Messages

The *X10 Messages* page displays received X10 commands, and it lets you compose and send X10 commands of your choosing.



The text box displays X10 traffic.

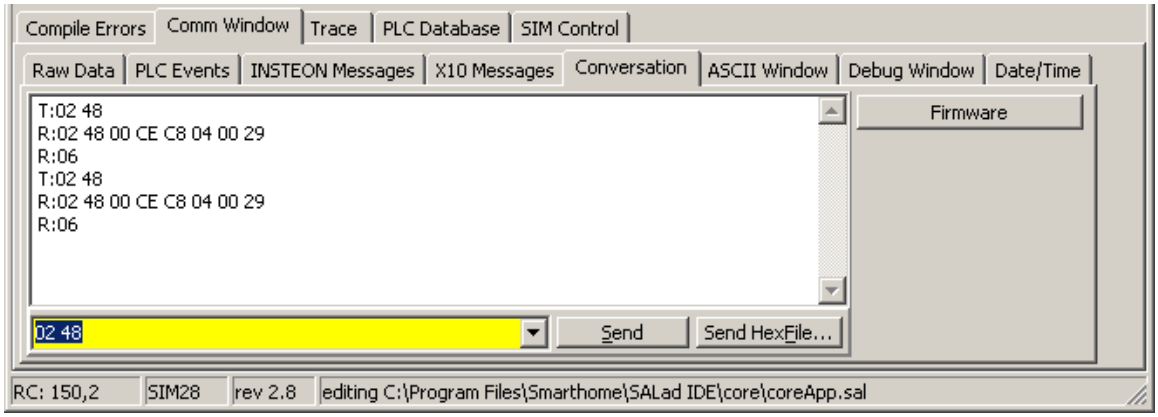
A **T:** precedes X10 commands transmitted by the PLC.

An **R:** precedes X10 commands received by the PLC.

To compose an X10 command, choose its two bytes from the pulldown boxes, then press the *Send* button.

Comm Window – Conversation

The *Conversation* page allows you to have a two-way serial dialog with the PLC.



The text box displays the conversation in hexadecimal bytes.

A **T:** precedes hex bytes transmitted to the PLC.

An **R:** precedes hex bytes received from the PLC.

The yellow text box at the bottom allows you to type in hex bytes to send to the PLC. The pulldown holds the history of your sent bytes.

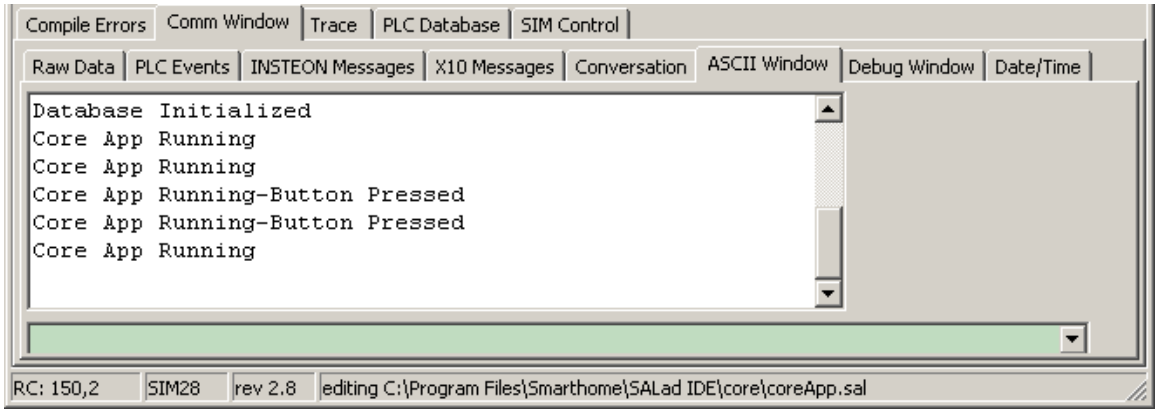
Press the *Send* button to transmit the bytes displayed in the yellow text box.

If you would like to send an entire hex file to the PLC, press the *Send Hex File…* button to choose and send the file. The hex file must be a text file containing 2-character ASCII hex bytes separated by spaces or newlines, such as the files produced by the compiler under the *Files->Save Hex Values As…* menu item.

The *Firmware* button sends 0x02 0x48, which requests the PLC's firmware revision. This is a convenient way to determine if a PLC is connected and listening.

Comm Window – ASCII Window

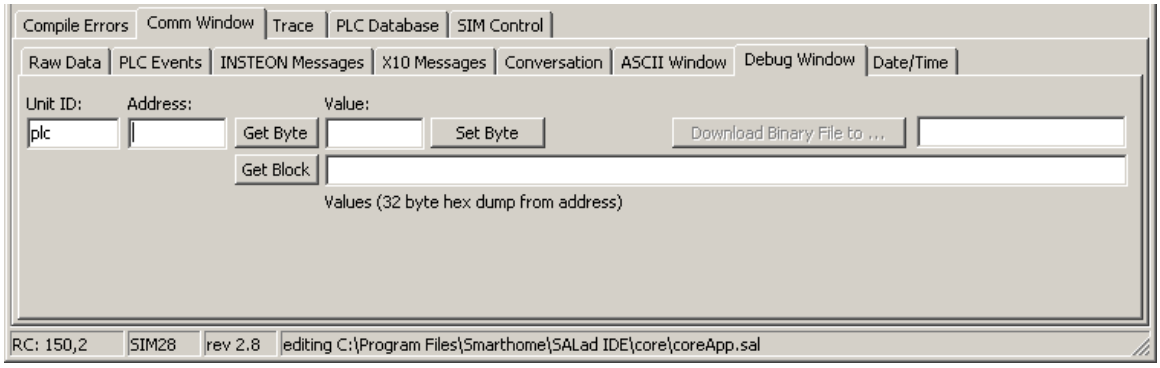
The *ASCII Window* displays text explicitly sent from a SALad 2 program running on your PLC.



Use the `SerialSendBuffer` command within a SALad program to generate messages that will be displayed in this text box.

Comm Window – Debug Window

The *Debug Window* lets you directly inspect and alter bytes within your PLC.



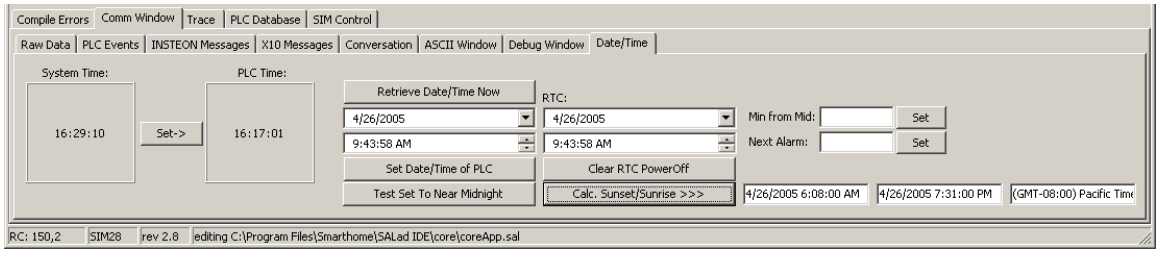
Enter *plc* in the *Unit ID:* box to inspect (peek) and write to (poke) bytes in your PLC. If you want to peek and poke bytes in a remote INSTEON device, enter that device's INSTEON Address as, for example, *1.2.3*.

Type the hex address you wish to inspect or write to in the *Address:* box. Press *Get Byte* to fetch a single hex byte at that address and display it in the *Value:* box, or else press *Get Block* to fetch a string of 32 hex bytes and display them all in the lower box.

To poke a byte, type the hex byte you wish to poke at the given *Address:* in the *Value:* box, then press *Set Byte*.

Comm Window – Date/Time

The *Date/Time* page gives you control over the realtime clock (RTC) in the PLC and lets you test realtime events. All times are in 24-hour 'military' format.



The *System Time* box displays the time according to your PC, and the *PLC Time* box displays the time according to your PLC.

To synchronize your PLC to your PC, press the *Set->* button. The time in the *PLC Time* box will update to the system time.

To set the PLC's RTC to an arbitrary date/time, enter the date and time you wish to set in the boxes above the *Set Date/Time of PLC* button, then press that button. The time in the *PLC Time* box will update within the next minute. To quickly restore the date and time boxes to the current PLC time, press the *Restore Date/Time Now* button.

Press the *Test Set to Near Midnight* button to set the PLC's time to 23:59:45 so you can test functions that occur at midnight.

Press the *Clear RTC PowerOff* button if you do not want the PLC's RTC to save the current time if the PLC loses power.

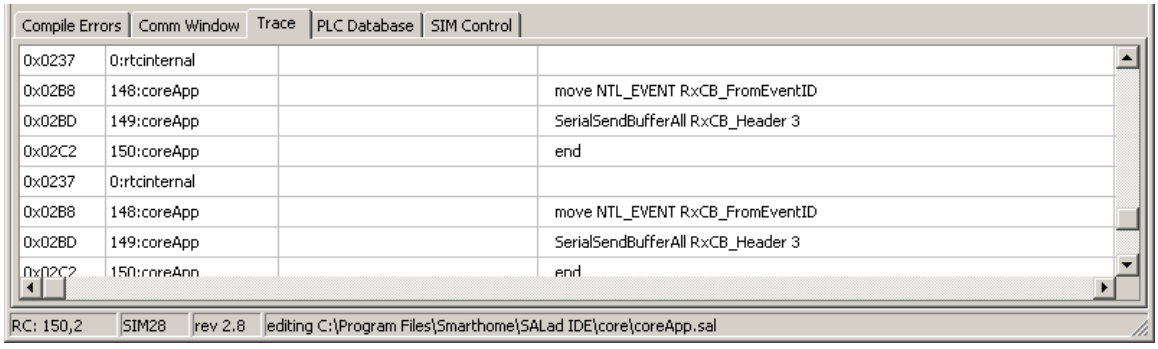
If you want to test timers that depend on the PLC's minutes-from-midnight counter (see *IBIOS Event Details* Note 8), you can set the counter by entering a value from 0 to 1439 in the *Min from Mid* (*Minutes from Midnight*) box and pressing the *Set* button to the right.

Enter a minutes-from-midnight value from 0 to 1439 in the *Next Alarm* box and press the *Set* button to the right to cause an **EVNT_ALARM (0x0E)** IBIOS Event to fire when the PLC's minutes-from-midnight value matches the value you entered. See *IBIOS Event Details* Note 8 for more information.

Press the *Calc Sunset/Sunrise >>>* button to fill in the boxes to the right with the sunrise and sunset times for the PLC date. The box to the right gives the hour-difference between the local time zone and GMT (Greenwich Mean Time).

Trace

The *Trace* page lets you inspect the execution history of your program to help you with debugging.



Tracing is active whenever you are in Debug mode (i.e. the *Debug Mode* checkbox is checked).

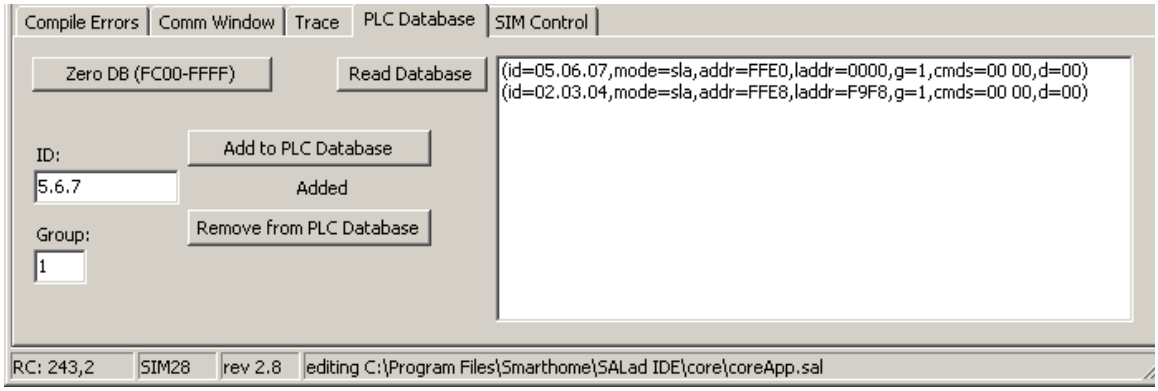
The first column gives the hexadecimal address of the object code that was executed.

The second column gives the line number and the source code filename of the line of code that was executed.

The rightmost column shows the program statement that was executed.

PLC Database

The *PLC Database* page is a tool for manipulating the Link Database in your PLC. See *INSTEON Link Database* above for more information.



To examine the contents of the PLC's Link Database, press *Read Database*. The contents will appear in the text box at the right. If the text box is blank, the Link Database is zeroed out.

To erase the Link Database by zeroing it out, press the *Zero DB (FC00-FFFF)* button. Zeros will be written to addresses 0xfc00 to 0xffff in the PLC's EEPROM (nonvolatile memory).

To add an entry to the Link Database, type the INSTEON Address of the device you wish to add in the *ID:* box, and the number of the Group you would like the device to belong to in the *Group:* box, and then press the *Add to PLC Database* button. When you enter the 3-byte INSTEON Address, type it as three decimal numbers, ranging from 0 to 255, separated by periods (for example: `126.23.4`).

To remove an entry from the Link Database, put the entry's ID and Group number in the appropriate boxes, and then press *Remove from PLC Database*.

After zeroing the Link Database or adding or removing a Link Database entry, you can press the *Read Database* button to see the effect of the change.

If you double-click on a line in the text box, the IDE will jump to the *Comm Window – Debug Window* and display a hex dump of the bytes in the Link Database starting at the address of the entry you double-clicked on.

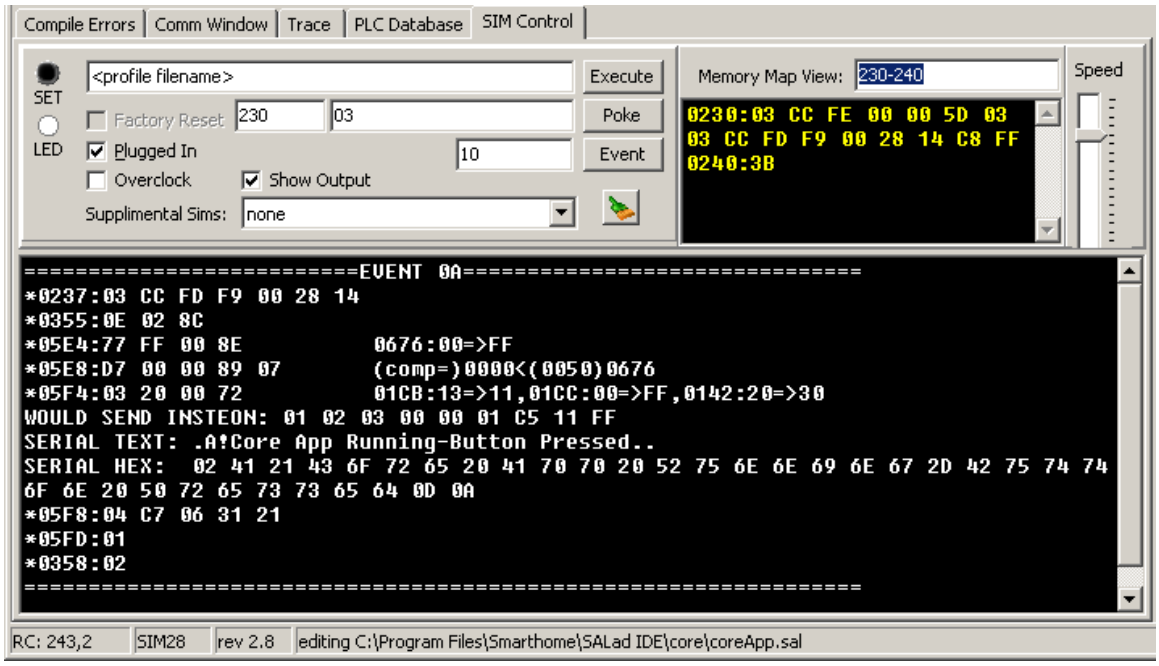
As an example, if you press the *Zero DB (FC00-FFFF)* button, enter an ID of `2.3.4` and a Group of `1`, press the *Add to PLC Database* button, and finally press the *Read Database* button, the following line will appear in the text box:

```
(id=02.03.04,mode=sla,addr=FFE8,laddr=0000,g=1,cmds=00 00,d=00)
```

See *PLC Link Database* above for the meaning of these fields.

SIM Control

The SIM Control window operates the PLC Simulator. To use the PLC Simulator in place of a real PLC, turn it on by choosing the *Mode->Simulator* menu item (see *Menu – Mode*).



The *PLC Simulator Control Panel* is at the upper left. The text box at the upper right is a hex *PLC Simulator Memory Dump*, and the text box at the bottom is a *PLC Simulator Trace* of code execution. You can vary the size of the Trace box by dragging its top border.

In This Section

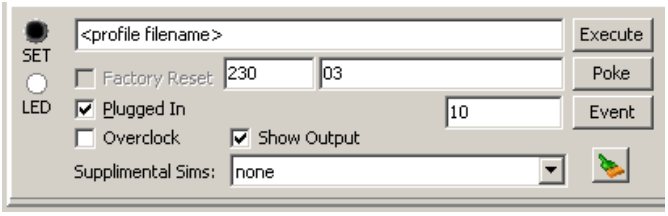
PLC Simulator Control Panel
: Explains the controls at the top of the window.

PLC Simulator Memory Dump
: Shows how to view memory in the PLC Simulator.

PLC Simulator Trace
: Describes the trace information in the bottom text box.

PLC Simulator Control Panel

This is what the PLC Simulator Control Panel section looks like:



A simulated *SET Button* (which you can press) and a simulated white *Status LED* (which you can view) appear at the left.

You can simulate unplugging and plugging in the PLC using the *Plugged In* checkbox. If you would like to simulate a factory reset, uncheck *Plugged In*, check *Factory Reset*, then check *Plugged In*.

Check *Overclock* to run the simulated PLC's realtime clock at very high speed between events. This lets you easily test code that depends on realtime events without waiting for the actual time to elapse or manually resetting the PLC's clock to fire an event.

Use the *Speed* control at the far right to slow down simulated execution speed. This can be useful for debugging.

If you are simulating a PLC with a Hardware Development Kit (HDK) added, choose the HDK from the *Supplemental Sims:* pulldown box.

You can cause the PLC Simulator to execute a string of ASCII text commands from a macro file by typing the filespec for the macro file in the text box to the left of the *Execute* button, then pressing the button. Contact info@insteon.net for the format for the macro file.

To poke a byte or bytes into the PLC Simulator's memory, enter the hex address to poke to, and the hex byte(s) you want to poke, in the text boxes to the left of the *Poke* button, and then press the button. If you are poking multiple bytes, separate them with spaces.

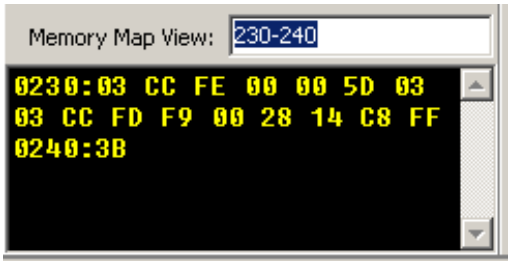
To simulate the occurrence of an IBIOS Event, enter the Event Handle number (see *IBIOS Events*) in the text box to the left of the *Event* button, then press the button.

Check the *Show Output* checkbox if you want the *PLC Simulator Trace* to display code execution information.

Push the *Erase* button  to blank the *PLC Simulator Trace* text box.

PLC Simulator Memory Dump

This is what the PLC Simulator Memory Dump section looks like:



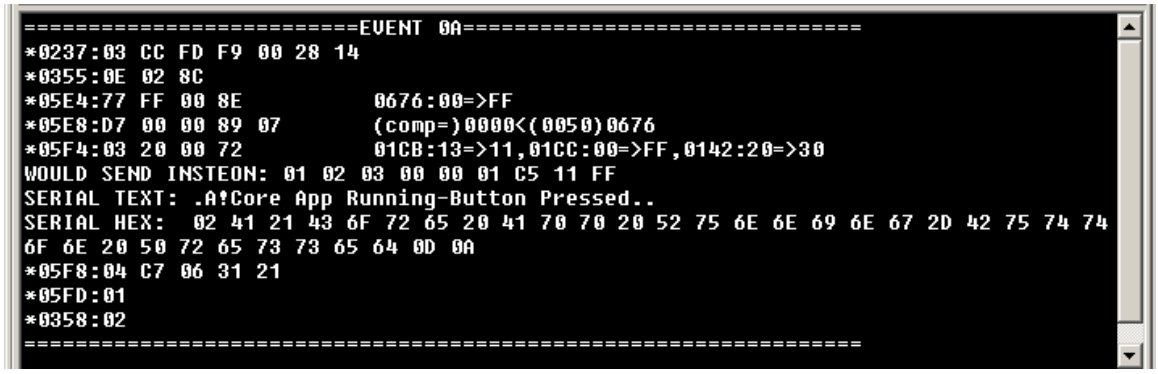
To view memory contents in the PLC Simulator, enter a hex address, a range of addresses, or some combination of addresses and ranges in the *Memory Map View:* text box. The text box will immediately display the memory contents you specified.

Indicate an address range by typing a beginning and ending address separated by a hyphen. Use a comma to separate multiple addresses or address ranges.

The memory dump normally refreshes automatically every few milliseconds. If for some reason refreshing stops (if you switch serial ports, for instance), you can restart it by typing anything in the *Memory Map View:* box.

PLC Simulator Trace

This is what the PLC Simulator Trace section looks like:



Drag the border at the top to resize the text box.

Push the *Erase* button  in the *PLC Simulator Control Panel* to blank the text box.

Check the *Show Output* checkbox in the *PLC Simulator Control Panel* if you want the text box to display code execution information.

When *Show Output* is checked, trace information like that shown above with an asterisk at the left will appear. Immediately following the asterisk is the hex address of the code line that was executed, followed by the object code itself. Effects that the code may have are shown to the right. For example,

```
0676:00->FF
```

means that memory location `0676` was changed from containing `00` to containing `FF`; and

```
(comp=)0000<(0050)0676
```

means that a compare was done between the literal value `0000` and the contents of the 16-bit value beginning at memory location `0050`, namely `0676`, and that `0000` is less than `0676`.

Whether or not the *Show Output* checkbox is checked, the text window will show INSTEON messages and X10 commands sent over the powerline by the PLC, as well as both ASCII and hex data sent serially to your PC.

IDE Virtual Devices

The *Virtual Devices* included in the SALad IDE allow you to develop SALad applications without being connected to any external hardware—in other words, you don't need a connection to a real PowerLinc™ V2 Controller (PLC).

The key tool is the *PLC Simulator*, which is a pure-software version of a real PLC running on your PC. The PLC Simulator can do everything that a real PLC can do, but it is more transparent, thanks to the special tools in the *SIM Control* window.

When you are using the PLC Simulator, you can also simulate a *Virtual Powerline* environment in software. Then, you can plug in any number of *Virtual LampLinc* devices to the Virtual Powerline to debug and test your SALad application.

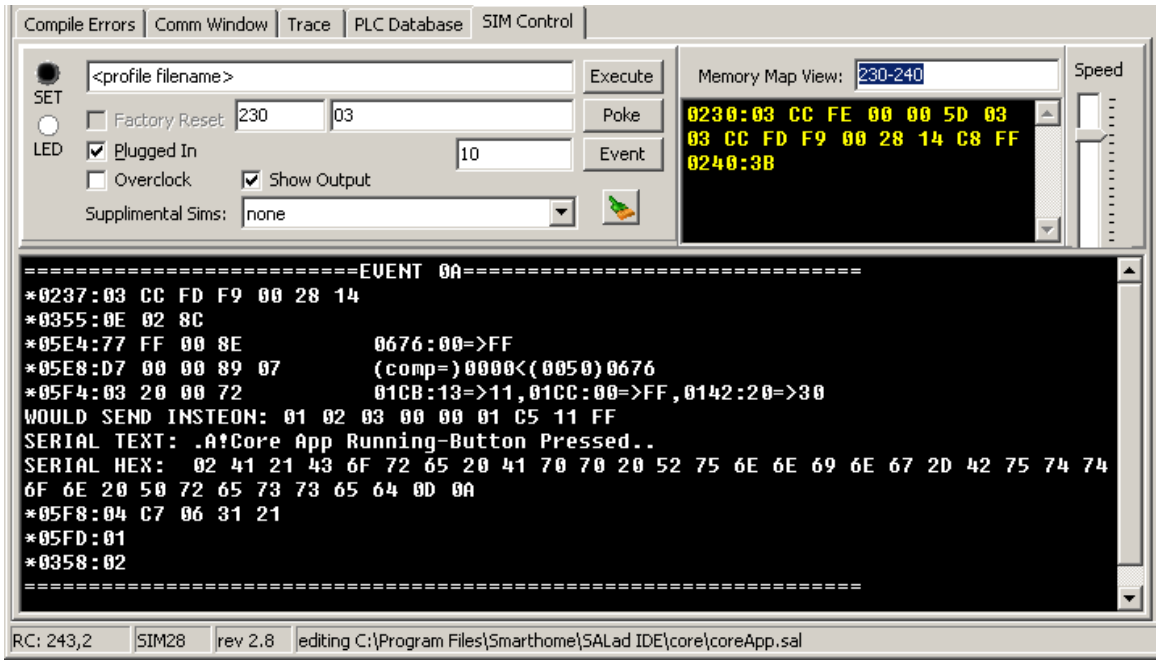
Although very convenient for code development, simulation is no substitute for real-world testing. Be sure to thoroughly shake out your application using real hardware before pronouncing it finished!

In This Section

PLC Simulator

The *PLC Simulator* is a pure-software version of a real PLC running on your PC. To use the PLC Simulator in place of a real PLC, turn it on by choosing the *Mode->Simulator* menu item (see *Menu – Mode*).

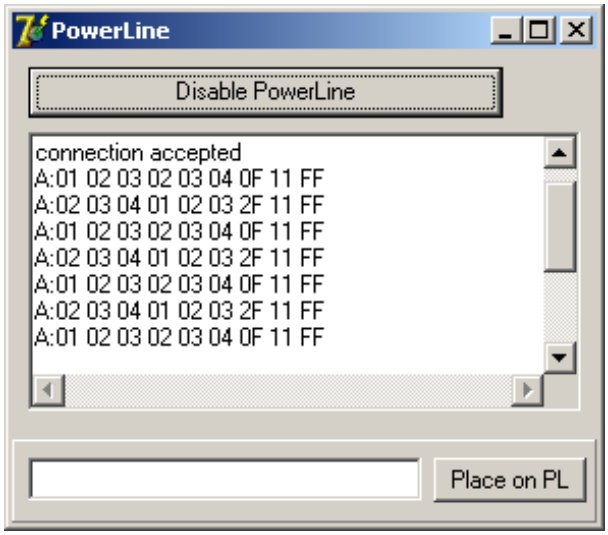
The PLC Simulator can do everything that a real PLC can do, but it is more transparent, thanks to a set of special software tools. These tools appear in the *SIM Control* window, which looks like this:



For a complete explanation of how to use these tools, see the *SIM Control* section.

Virtual Powerline

The *Virtual Powerline* is a software-simulated powerline environment that works in conjunction with the *PLC Simulator*. To turn it on, choose the *Virtual Devices->Powerline* menu item (see *Menu – Virtual Devices*). The Virtual Powerline will appear as a separate window that looks like this:



The button at the top will be labeled *Enable Powerline* if the Virtual Powerline is currently disabled, or else it will say *Disable Powerline* while the Virtual Powerline is enabled. To use the Virtual Powerline, enable it. If the PLC Simulator is not running, turn it on, or if it is running, you may have to 'Unplug' it then 'Plug' it back in. When the PLC Simulator is using the Virtual Powerline properly, the message

```
connection accepted
```

will appear in the Virtual Powerline's text box. The same message will appear every time you connect a virtual device, such as a *Virtual LampLinc*. When you remove a virtual device you will get the message

```
connection removed
```

You can place a message on the Virtual Powerline by typing the ASCII string you want to send in the box to the left of the *Place on PL* button, then pressing the button. The string you sent will appear in the text box.

INSTEON messages that appear in the Virtual Powerline will show up as hex bytes preceded by `A:` in the text box.

To stop using the Virtual Powerline, close its window.

Virtual LampLinc

Virtual LampLinc devices are software simulations of real Smarthome LampLinc™ V2 Dimmers. Virtual LampLincs are connected to a *Virtual Powerline*, which in turn connects to a *PLC Simulator*. To launch a Virtual LampLinc, choose the *Virtual Devices->LampLinc* menu item (see *Menu – Virtual Devices*). The Virtual LampLinc will appear as a separate window that looks like this:

The text box labeled *INSTEON Address (A.B.C)* at the top of the window contains the 3-byte ID of the Virtual LampLinc. This will be a unique number for each Virtual LampLinc that you launch. If you wish to assign a different ID, type it in the text box as three decimal digits separated by periods.

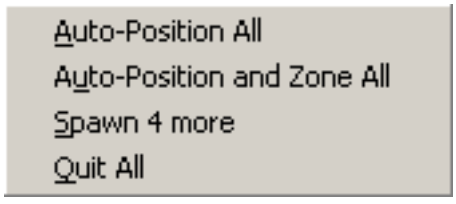
The *Zone:* pulldown box lets you place Virtual LampLincs on the Virtual Powerline at varying simulated distances from one another. The greater the difference between Zone numbers, the greater the simulated distance between LampLincs. This lets you simulate powerline environments that require multiple hops for INSTEON messages to get through (see *INSTEON Message Hopping*, above).

You can simulate plugging in or unplugging a Virtual LampLinc with the button that will be labeled *(IN) Click to Unplug* or else *Plug In*.

The black circle at the bottom right of the picture of the LampLinc is the simulated *SET Button*, which you can push with the mouse. The white circle below that is the simulated white *Status LED*, which you can view.

The square at the bottom shows the current dimming state of the Virtual LampLinc. Its color ranges from black for off, through various shades of gray, to white for full on.

You can launch multiple Virtual LampLincs by right-clicking anywhere in a Virtual LampLinc window to display a popup menu that looks like this:



Choose *Spawn 4 more* to create four more Virtual LampLincs, each in a separate window, and each with a different INSTEON Address. Choose *Auto-Position All* to neatly tile all open Virtual LampLinc windows. If you choose *Auto-Position and Zone All*, groups of Virtual LampLincs will appear in different Virtual Powerline zones. Choose *Quit All* to close all of the Virtual LampLinc devices at once.

IDE Keyboard Shortcuts

The table below shows how you can use the keyboard to perform common actions in the IDE.

Key	Action
Ctrl-A	Select all text in the currently displayed editor file
Ctrl-X	Cut selected text in the currently displayed editor file
Ctrl-C	Copy selected text in the currently displayed editor file
Ctrl-V	Paste selected text in the currently displayed editor file
Ctrl-S	Save the currently displayed editor file
Alt-G	Go to line number
Ctrl-F	Find a search string
Ctrl-R	Find a search string and replace it with another string
Ctrl-Down	Find next occurrence of search string
Ctrl-Up	Find previous occurrence of search string
Ctrl-F4	Close currently displayed source file
Ctrl-(left mouse click)	If cursor is on an 'Include' file, open the file
F2 (in debug mode)	Stop the program
F8 (in debug mode)	Single Step the program
F9 (not in debug mode)	Compile and Download the program to the PLC
F9 (in debug mode)	Run the program

Smarthome Device Manager Reference

This section documents the Smarthome Device Manager (SDM). SDM is in the class of *Manager Applications*, which are programs that run on computing devices external to an INSTEON network and expose a high-level interface to the outside world.

In This Section

SDM Introduction
Gives an overview of the SDM and lists system requirements.

SDM Quick Test
Gets you up and running using either a browser or SDM's Main Window.

SDM Commands
Lists the available SDM Commands.

SDM Windows Registry Settings
Gives the Windows Registry settings that SDM uses.

SDM Introduction

The Smarthome Device Manager™ (SDM) is a communication and translation gateway to *The Smarthome PowerLinc Controller* (PLC). Developers use simple text commands (Home Networking Language™) through ActiveX or HTTP calls to communicate over INSTEON or X10 without the hassle of dealing with USB packets or RS232 resource issues.

These commands will also be extended to PowerLinc/IP and other Internet transports.

Using the Smarthome Device Manager, developers can focus on their application, whether in Macromedia Flash®, Java®, .NET® or even in a Word® document, Excel® spreadsheet, or a PowerPoint® presentation.

Commands such as "SetOnLevelText=01.02.03,ON" perform the action and return a response, providing developers with simple and reliable control. Additional commands such as "speak," inet," and "mailto" further provide the developer with powerful capabilities.

SDM System Requirements

SDM Platforms

Windows XP, 2000, NT (serial only), Me, 98, and 95 (serial only).

SDM Programming Platforms

any language that provides either ActiveX, HTTP, or shell calls including (Java, VB, .NET, C#, C++, Delphi, Flash, Perl, ASP, PHP, VB/W/JScript, Tcl, Python and more).

SDM Resources

HTTP Server uses port 9020 and offers a server-pull method to facilitate firewalls. Connects via COM1~COM255 or USB.

PowerLinc Controller Firmware Requirement

2.8 or better

SDM Quick Test

You can get the SDM up and running quickly using either a browser or SDM's Main Window.

In This Section

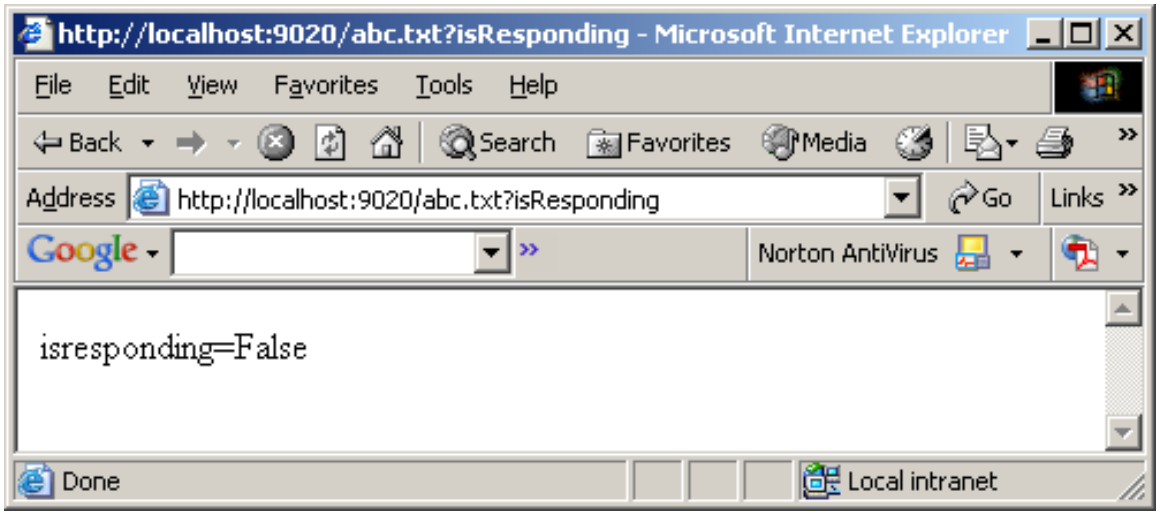
SDM Test Using SDM's Main Window
Use SDM's Main Window to interface with SDM.

SDM Test Using a Browser
Use a browser like Internet Explorer or FireFox to interface with SDM.

SDM Test Using SDM's Main Window
Use SDM's Main Window to interface with SDM.

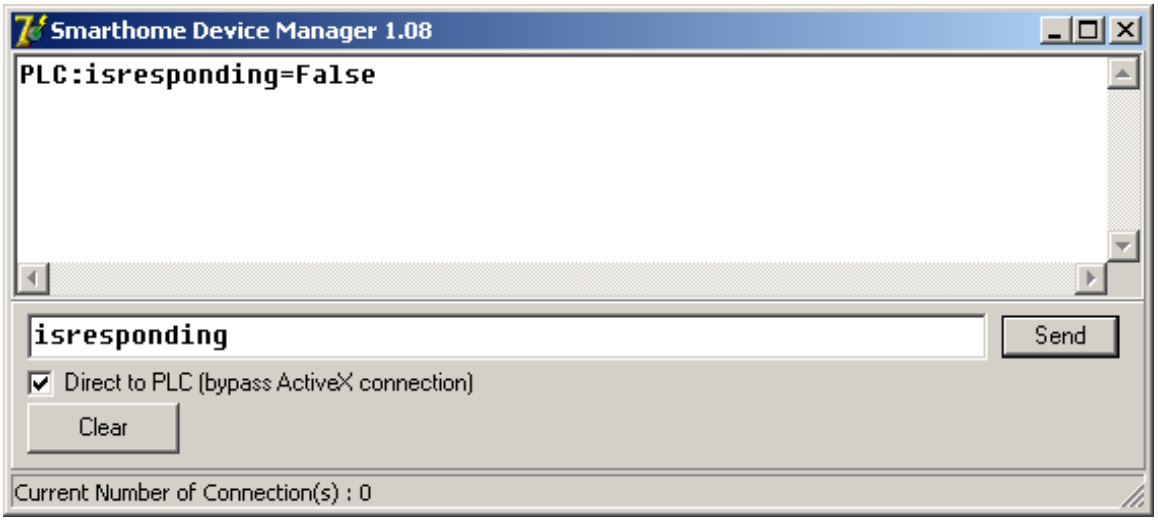
SDM Test Using a Browser

1. Run Smarthome Device Manager (SDM2Server.exe). An icon  will appear in your systray (normally at the lower right of your screen).
2. Run a browser (such as Internet Explorer or FireFox).
3. Type **http://localhost:9020/abc.txt?isResponding** into the browser as the URL.
4. You should see a textual response isresponding=False or isResponding=True with other cached responses.



SDM Test Using SDM's Main Window

1. Run Smarthome Device Manager (SDM2Server.exe). An icon  will appear in your systray (normally at the lower right of your screen).
2. Single-click on the icon and SDM's Main Window will appear.



3. Type **isresponding** into the bottom text box and press *Send*.
4. You should see a textual response isresponding=False or isResponding=True with other cached responses.

SDM Commands

This section lists and explains the available SDM Commands.

In This Section

SDM Commands – Getting Started

port=<PLCport> - sets the sticky, global PLC port (COM#,USB4,SIM28,?). Sending a question-mark '?' causes the port to be found. Use **getport=** to read the found port. The port is saved in the registry and reused upon restart of the Smarthome Device Manager.

Examples: port=COM1 or port=USB4 or port=? or port=SIM28

isResponding - asks the Device Manager if the port is responding (sends 0x02 0x48) and responds true or false. This also reads the map for proper name-based downloading. This is the ultimate heartbeat method to determine if the PLC is connected and talking.

Example: isResponding - returns isResponding=true

addid=<remoteINSTEONid> - add a device's ID to the PLC's database.

Example: addid=04.05.06

(only if necessary, such as the LED blinking or the PLC not communicating)
downloadCoreApp[=clear] - downloads the included core application to the PLC so that it can communicate to the Smarthome Device Manager. The optional =clear parameter also instructs the core application to clear the link table database after downloading.

Example: downloadCoreApp

setOnLevelText=<INSTEONid>, <onLevelCmdOrValue>[,<hops>] - set on-level status (ON/OFF/dec%/dec/0xHex) of an ID, optionally specifying the number of hops. (defaults to 3 hops).

Examples: setOnLevelText=04.05.06,ON or setOnLevelText=04.05.06,49% or setOnLevelText=04.05.06,127 or setOnLevelRaw=04.05.06,0x7F

SDM Commands – Home Control

setOnLevelText=<INSTEONid>,<onLevelCmdOrValue>[,<hops>] - set on-level status (ON/OFF/dec%/dec/0xHex) of an ID, optionally specifying the number of hops. (defaults to 3 hops).

Examples: setOnLevelText=04.05.06,ON or setOnLevelText=04.05.06,49% or setOnLevelText=04.05.06,127 or setOnLevelRaw=04.05.06,0x7F

getOnLevelText=<INSTEONid>[,<hops>] - get on-level status from an ID as a text representation, optionally specifying the number of hops. (defaults to 3 hops). Returns ON,OFF,OUT,or percentage of on (1-99%).

Examples: getOnLevelText=04.05.06. Returns getOnLevelText=04.05.06,ON or getOnLevelText=04.05.06,49%

sendX10=<addressOrCommand>[,<addressOrCommand>] - sends x10 addresses and commands.

Example: sendX10=A1,AON or sendX10=A1,A2 or sendX10=AON

sendGroupBroadcast=<groupID>,<cmd1>[,<cmd2>][,<hops>] - sends a broadcast from the PowerLinc Controller with the included group ID, and command1 (ON/OFF/hex), and optional command2 (defaults to 0) and hops (defaults to 3).

SDM Commands – Notification Responses

These are for uninitiated messages.

eventRaw=<eventID> - notification of an event in hex.

Example: eventRaw=0A

receiveX10=<AddressOrCommand> - notification of an X10 address or command received, in text.

Example: receiveX10=A1 or receiveX10=A On

receiveX10raw=<x10type> <AddressOrCommandByte> - notification of an X10 address or command received, in hex form. x10type is zero (0) for an address, or one (1) for a command.

Example: receiveX10raw=00 66

receiveINSTEONraw=<eventID> <messageBytes...> - notification of an INSTEON message received. The number of bytes varies depending upon the eventID and whether the message is standard or extended.

Example: receiveINSTEONraw=02 04 05 06 00 00 11 8F 01 00

enrolled=<INSTEONid>,<deviceInfoBytes>,<deviceName> - notification that an INSTEON device was enrolled into the PLC.
NOTE: linked= will replace enrolled= ... **linked=**<INSTEONid>, <groupID>, <deviceInfoBytes>, <responderOrController>[, <deviceSKU>, <deviceName>]

usbarrival=<VenderID>,<PorductID>,<Version>,<Company>,<ProductName> - notification that the PLC was connected. (Specific to PLC from Smarthome).

Example: usbarrival=4287,4,1024,SmartHome,SmartHome PowerLinc USB E,

usbunplugged=4287,4,1024,,, - notification that the PLC was disconnected (once connected). (Specific to PLC from Smarthome).

Example: usbunplugged=4287,4,1024,,,

progress=

SDM Commands – Direct Communications

These are for raw, low-level communication.

sendINSTEONraw=<9 or 23 hexbytes> - send the 9 (standard) or 23 (extended) INSTEONbytes from the PLC.

Example: sendINSTEONraw=01 02 03 04 05 06 0F 11 FF (send from unit 01.02.03 (which is overwritten with the actual PLC ID) to unit 04.05.06, with 3 hops, an ON at full brightness).

sendrecINSTEONraw/SRIR=<9 or 23 hexbytes> - send the 9 (standard) or 23 (extended) INSTEONbytes from the PLC *and wait for the response (ACK/NAK)*.

Examples: sendrecINSTEONraw=01 02 03 04 05 06 0F 11 FF or SRIR=01 02 03 07 08 09 0F 13 00. Returns SRIR=04,07 08 09 01 02 03 2F 13 00 (the returned ACK packet response).

getOnLevelRaw=<INSTEONid>[,<hops>] - get on-level status from an ID as a hexbyte representation, optionally specifying the number of hops. (defaults to 3 hops). Returns 00-FF.

Examples: getOnLevelRaw=04.05.06. Returns getOnLevelRaw=04.05.06,7F

setOnLevelRaw=<INSTEONid>,<onLevelCmdOrValue>[,<hops>] - set on-level status from an ID as a hexbyte representation, optionally specifying the number of hops. (defaults to 3 hops). Set/Returns 00-FF.

Examples: setOnLevelRaw=04.05.06,7F. Returns setOnLevelRaw=04.05.06,7F

sendX10raw=<x10type>,<AddressOrCommandByte> - send an X10 address or command byte. x10type is zero (0) for an address, or one (1) for a command.

Example: sendX10raw=00, 66 (sends A1 address).

sendPLC=<data...> - send direct raw hex bytes to the PLC, as if through a direct connection (such as serial).

Example: sendPLC=02 40 01 65 00 01 FF 33 66

sendEventRaw=<eventID> - sends an event (00-FF) for the PLC to execute (see INSTEON EVENTS).

SDM Commands – Memory

setImage=<address>,<hexdata...> - downloads data to the address (map-friendly).

Examples: setImage=0x0040,02 FF or setImage=NTL_TIMERS,02 FF

getImage=<address>,<length> - uploads the image to the PC, returns hex bytes. (map-friendly).

Examples: getImage=0x0040,2 returns getImage=0x0040,00 00

setBit=<address>,<bit>[,<setTo0or1>] - sets the bit (0-7) at the address (map-friendly) with an optional value of 1(set) or 0(clear) (defaults to 1).

Example setBit=0x0040,4 or setBit=0x0040,4,0

clearBit=<address>,<bit> - clears the bit (0-7) at the address (map-friendly).

Example clearBit=0x0040,4

downloadCoreApp[=clear] - downloads the included core application to the PLC so that it can communicate to the Smarthome Device Manager. The optional =clear parameter also instructs the core application to clear the link table database after downloading. Automatically resets the PLC after download.

Example: downloadCoreApp

downloadSALadFile=<filename> - downloads the SALad program filename provided (as a binary/compiled file, not hex). Automatically resets the PLC after download.

Example: downloadSALadFile=coreUser.slb

downloadBinaryFile=<address>,<filename> - downloads the binary file to the address provided. This command does *not* automatically reset the PLC after download.

Example: downloadBinaryFile=<0x2000>,myTable.bin

verifyCoreApp - returns true or false with information whether the currently download application matches the expected core application.

Example: verifyCoreApp

SDM Commands – PLC Control

port=<PLCport> - sets the sticky, global PLC port (COM#,USB4,SIM28,?). Sending a question-mark '?' causes the port to be found. Use **getport=** to read the found port. The port is saved in the registry and reused upon restart of the Smarthome Device Manager.

Examples: port=COM1 or port=USB4 or port=? or port=SIM28

getport - returns the current sticky, global PLC port.

Example: getport returns getport=USB4

getFirmware - returns the PLC firmware revision.

Example: getFirmware returns getFirmware=2.9

sendHardReset - resets the SALad application, reinitializes (executes the INIT event).

getClock - gets the current PLC clock (which is reset from the RTClock on initialization/plugin in the Core Application).

Example: getClock returns getclock=true,8/12/2005 5:03:39 PM

getRTClock - gets the current real-time clock.

Example: getRTClock returns getclock=true,8/12/2005 5:03:39 PM

setClock - sets BOTH the real-time clock and the running clock.

Example: setClock=8/12/2004 5:03:39 PM

setRunningClock - sets only the running clock, not the real-time clock. This means that when the PLC is plugged in next or reset, the core app will copy the real-time clock over the running clock.

Example: setRunningClock=8/12/2004 5:03:39 PM

setRTClock - sets only the real-time clock, not the running clock. This means that when the PLC is plugged in next or reset, the core app will copy the real-time clock over the running clock.

Example: setRTClock=8/12/2004 5:03:39 PM

connect - connect to the PLC after a disconnection (connecting is done automatically at startup).

Example: connect

disconnect - disconnect the PLC's port connection.

Example: disconnect

isResponding - asks the Device Manager if the port is responding (sends 0x02 0x48) and responds true or false. This also reads the map for proper name-based downloading. This is the ultimate heartbeat method to determine if the PLC is connected and talking.

Example: isResponding - returns isResponding=true

getplcstatus - returns set of statuses for PowerLinc Controller (port, connection, etc).

Example: getplcstatus

help - provides a quick visual list of commands.

Example: help

getMyID - returns the PLC's ID.

Example: getMyID returns getMyID=01.02.03

SDM Commands – Device Manager Control

setTextMode=<textmode> - sets the global communication format that the Device Manager uses to receive and respond. Currently 'text' (default) and 'flash' are supported. 'flash' mode adds ampersands (&) around each response in order for the loadVariables() function to receive values from Macromedia Flash or SwishMax type clients.

nop - does nothing, but allows an HTTP connection to return collected data to the client.

echo - echoes the text simply to see if the ActiveX or HTTP communication is working.

_localbytes=<true/false> - turns on/off local (window) Device Manager byte-level debugging.

remotebytes=<true/false> - turns on/off local (window) Device Manager byte-level debugging.

setBlocked=<true/false> - (default false) turns on/off global blocking of commands. When blocked, the action is executed before receiving a response and returning to the client. When not blocked, the client is returned a true response that the command was accepted for execution by the Device Manager. When the actual response is received by the Device Manager, it is sent to the client via ActiveX, or queued for return via HTTP. (use NOP to get results when no execution action is necessary).

setPageSizeThrottle=<true/false> - (default true) allows global throttling of the downloads in case of errors. The download page size shrinks when multiple consecutive errors are received. The page size grows when multiple consecutive successes are received.

setPageErrors=<maxErrorCount> - (default 7) sets the global maximum number of consecutive retries before a download actually fails.

setPageSize=<pageSize> - (default 32) sets the global packet size for downloading. When throttling is true, this value automatically shrinks and grows.

if=<command>,<matchResult>,<trueText>[,<falseText>] - executes the command, matches against the matchResult. If true, returns the trueText. If false (and the falseText is present), returns the falseText.

Example: if=getFirmware,2.9,good,bad

ifexec=<command>,<matchResult>,<trueCommand>[,<falseCommand>] - executes the command, matches against the matchResult. If true, executes the trueCommand. If false (and the falseCommand is present), executes the falseCommand.

Example: ifexec=getFirmware,2.9,"setOnLevelText=00.02.BA,ON",nop

setAuthUsername=<username> - allows user to set an authorization username for the http/web connection to require. Shipped default is empty - no authorization required.

Example: setAuthUsername=me

setAuthPassword=<password> - allows user to set an authorization password for the http/web connection to require. Shipped default is empty - no authorization required. To clear, send setAuthPassword with no password.

Example: setAuthPassword=12345

SDM Commands – Link Database Management

addID=<remoteINSTEONid> - add a device's ID to the PLC's database.

Example: addID=04.05.06

removeID=<remoteINSTEONid> - deletes a device's ID from the PLC's database.

Example: removeID=04.05.06

getRemoteRecord=<INSTEONid>, <record number>[,<end-range record number>] - gets and block-returns remote database records.

Examples: getRemoteRecord=04.05.06,2 or getRemoteRecord=04.05.06,2,3.
Returns getremoterecord=
=====RECORDS BEGIN=====
remoterec(04.05.06):#2:A2 01 00 D0 80 FE 1F 00
remoterec(04.05.06):#3:A2 02 00 6B C2 FE 1F 00
=====RECORDS END=====

getLinks - returns a block-list of records in the PowerLinc Controller.

Example: getLinks

getRemoteGroupRecord=[<recno>:] <remoteINSTEONid>, <groupID> [,<sourceINSTEONid>] [,<hops>] - scans the remoteINSTEONid's (non-PLC) database for the sourceID (defaults to PLC's ID) and groupID, or uses the recno provided. Returns full record information as getRemoteGroupRecord=<remoteINSTEONid>, <sourceINSTEONid>, <groupID>, <onLevelText>, <rampRate>

Example: getRemoteGroupRecord=04.05.06, 1 returns getRemoteGroupRecord=04.05.06, 01.02.03,1,50%,31

setRemoteGroupRecord=[<recno>:] <remoteINSTEONid>, <groupID>,<sourceINSTEONid>,<hops>, <newPresetDim>, <newRampRate> - scans the remoteINSTEONid's (non-PLC) database for the sourceID (defaults to PLC's ID), and groupID, or uses the recno provided. Sets full record information.

Example: setRemoteGroupRecord=1:04.05.06, 1, 01.02.03,3,50%,31 or setRemoteGroupRecord=04.05.06, 1, 01.02.03,3,50%,31

setPresetDim=<remoteINSTEONid>, <groupID>, <newPresetDim> [,<sourceINSTEONid>] [,<hops>] - sets the preset dim value for the PLC or sourceINSTEONid (defaults to PLC's id) in the remoteINSTEONid's database.

Example: setPresetDim=04.05.06, 1, 25%

setRampRate=<remoteINSTEONid>, <groupID>, <newRampRate> [,<sourceINSTEONid>] [,<hops>] - sets the ramprate dim value for the PLC or sourceINSTEONid (defaults to PLC's id) in the remoteINSTEONid's database. (RAMPRATE VALUES ARE 0x00-slow to 0x1F-fast).

Example: setPresetDim=04.05.06, 1, 0x1F

getPresetDim=<remoteINSTEONid>, <groupID>, [,<sourceINSTEONid>] [,<hops>] - gets the preset dim value for the PLC or sourceINSTEONid (defaults to PLC's id) in the remoteINSTEONid's database.

Example: getPresetDim=04.05.06, 1

getRampRate=<remoteINSTEONid>, <groupID>, [,<sourceINSTEONid>] [,<hops>] - gets the ramprate dim value for the PLC or sourceINSTEONid (defaults to PLC's id) in the remoteINSTEONid's database. (RAMPRATE VALUES ARE 0x00-slow to 0x1F-fast).

Example: setPresetDim=04.05.06, 1

SDM Commands – Timers

USAGE NOTES: Before adding/using timers, you must once download the timerCoreApp.slb file and use the (**clearTimers**) command. This resets all internal tables before you can add timers. Also, for using sunset/sunrise, you need to set your state/city (**setStateCity**) or lat/long (**setLatLong**) and download the sunset table (**downloadSunTable**).

downloadSALadFile=timerCoreApp.slb - downloads the timer core application, required for timer usage. Downloading WILL prevent you from listing timers until clearTimers is used.

clearTimers - clears the timer tables. Eliminates all timers and resets timer variables.

listTimers - block-lists the currently existing timers. Returns <recordnumber>, <active/inactive>, <time/sunrise/sunset+-mins>, <INSTEON-ID:onlevel / INSTEON: 6bytes>, <DOWfilter>, <SEC/NOSEC>. See addTimer for DOW/Sec info.

Example: listTimers returns listtimers=beginlist
timer=1,active,19:53,04.05.06:OFF,SuSaFThWTuM,NOSEC
timer=2,active,20:00,04.05.06:50%,SuSaFThWTuM,NOSEC
timer=3,active,19:59,INSTEON:04 05 06 4F 11 00,M,SEC
timer=4,active,sunset+20,04.05.06:OFF,SuSaFThWTuM,NOSEC
endlist

getTimer=<recordnumber> - same as listTimers, except lists a single timer (not block-listed) specified by <recordnumber>.

Example: getTimer=2 - returns
timer=2,active,20:00,04.05.06:50%,SuSaFThWTuM,NOSEC.

addTimer=<time/sunrise/sunset+-mins>,<INSTEON-ID:onlevel> [,<DOW>] [,<Sec/No>] [,<active/inactive>] - at the time specified, send on/off/0xhex/dec/brightness% to the INSTEON-ID. <time> can either be military, or sunrise or sunset. If 'sunrise' or 'sunset' is specified, you can specify the number of offset minutes. <DOW> uses M, Tu, W, Th, F, Sa, Su as specifiers (without spaces or commas). 'All' is an abbrieviation for MTuWThFSaSu. The defaults are <DOW=All> <Sec/No=No> <active/inactive=active>.

Examples: addTimer=14:30,04.05.06:25% or addTimer=sunset+10,04.05.06:OFF or addTimer=sunrise-25,04.05.06:ON,MTuWF,SEC

addTimer=<time/sunrise/sunset+-mins>,<INSTEON:-6 hex bytes-> [,<DOW>] [,<Sec/No>] [,<active/inactive>] - at the time specified, send 6 bytes via INSTEON. <time> can either be military, or sunrise or sunset. If 'sunrise' or 'sunset' is specified, you can specify the number of offset minutes. <DOW> uses M, Tu, W, Th, F, Sa, Su as specifiers (without spaces or commas). 'All' is an abbrieviation for MTuWThFSaSu. The defaults are <DOW=All> <Sec/No=No> <active/inactive=active>.

Examples: addTimer=14:30,INSTEON:04 05 06 0F 11 80 or addTimer=sunset+10,INSTEON:04 05 06 0F 13 00 or addTimer=sunrise-25,INSTEON:04 05 06 0F 11 FE,MTuWF,SEC

setTimer=<recordnumber>, <time/sunrise/sunset+-mins>,<INSTEON:-6 hex bytes-> [,<DOW>] [,<Sec/No>] [,<active/inactive>] - same as addTimer, except

setTimer overwrites a specific timer record specified by recordnumber, which is listed in the listTimers.

Example: setTimer=1,14:30,04.05.06:25% overwrites record 1.

deleteTimer=<recordnumber> - deletes the timer specified by <recordnumber> as listed by listTimers.

Example: deleteTimer=1

setStateCity=sets the latitude and longitude and sunrise/sunset information based on a state and city name. Information located in places.csv and places.idx for UI to enumerate. You can query getLatLong or getSunrise or getSunset after setting the State and City.

Example: setStateCity=CA,IRVINE or setStateCity=California,tustin

setLatLong=sets the latitude and longitude and sunrise/sunset information. You can query getLatLong or getSunrise or getSunset after setting this.

Example: setLatLong=33.684065,-117.792581

getLatLong - returns the currently set latitude and longitude used for sunrise and sunset calculations.

Example:
getlatlong - returns 33.684065,-117.792581

downloadSunTable - verifies that the TimerCoreApp is installed and downloads the sunrise/sunset table (based on latitude and longitude) to the PLC for use within the TimerCoreApp. This is required once to enable the sunrise/sunset time specifications.

Example: downloadSunTable

getsunrise - returns today's sunrise time as calculated with the setStateCity or setLatLong.

Example: getsunrise returns 8/30/2005 6:24:00 AM

getsunset - returns today's sunset time as calculated with the setStateCity or setLatLong.

Example: getsunrise returns 8/30/2005 7:20:00 PM

getnextalarmtime - returns the PLC's next scheduled alarm time. Note that this may not exactly match the time specified for timers where SECurity is set. When the minutes from midnight (getminutes) matches the getnextalarmtime, the alarm(s) that matches (or not for security) will fire.

Example: getnextalarmtime returns HH:MM such as 14:25

setnextalarmtime=HH:MM - set the PLC's next alarm time. Useful to skip alarms, if necessary, while they are reset for the next day. When the minutes from midnight (getminutes) matches the getnextalarmtime, the alarm(s) that matches (or not for security) will fire.

Example: setnextalarmtime=18:00 - essentially skips (today) timers until 6pm.

getminutes - returns the PLC's current minutes from midnight, set on initialization and auto-incremented each minute.

Example: getminutes returns HH:MM such as 15:25.

setminutes=HH:MM - set the PLC's current minutes from midnight, normally set on initialization and auto-incremented each minute. Useful to debug timers by setting the PLC's match of alarms without actually changing the PLC's clock.

Example: setminutes=18:00

SDM Windows Registry Settings

The Device Manager's root location is:

HKEY_CURRENT_USER\Software\Smarthome\ SmarthomeDeviceManager

Valuename: **port** = <port> - sticky global port for Device Manager to connect - USB4 or COM1..COM255. Set from the port= command.

Valuename: **servername** = <exefilename> - Device Manager's executable for clients to autorun.

Valuename: **usehttp** = <true/false> - allow the http server to accept connections. (Defaults to true)

Valuename: **httpport** = <portnumber> - when the http server activates, the server uses this port (Defaults to 9020).

INSTEON Hardware Development Documentation

In This Section

Hardware Overview

The HDK consists of a Main Board and a Daughter Board.

Main Board

The Main Board comes in 2 varieties: **Isolated** (for interfacing INSTEON networks sensitive equipment), and **Non-Isolated** (for equipment that has direct access to 120 VAC). The Main Board and Daughter Board schematics are shown in later sections of this document. The Main Board consists of:

- INSTEON Powerline Interface
- INSTEON Micro Controller Unit (MCU)
- INSTEON Enrollment User Interface (Button / LED)
- Power Supply

Daughter Board

The Daughter Board has an experimental area for you to develop your circuitry and makes available the signals from the Main Board. The Daughter Board consists of:

- Experimental Design Area
- Button for connecting Daughter Board Interrupt line to ground
- LED connected to 1 General Purpose I/O line

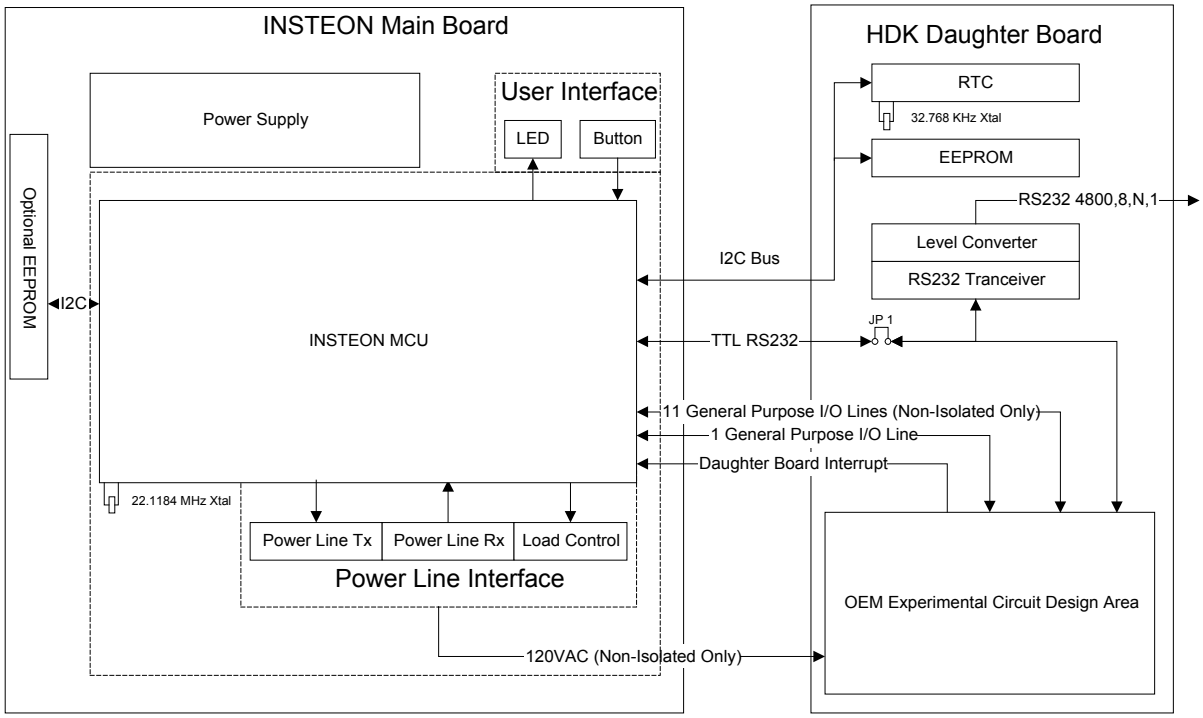
When connected to the **Non-Isolated** Main Board, the Daughter Board can make use of the following hardware:

- LED to simulate Load Control
- 11 Extra General Purpose I/O lines
- 120 VAC

Functional Block Diagram

This diagram shows the functional blocks on each board. The Main board serves as the INSTEON modem. The INSTEON chip executes SALad application code. The Daughter Board functions as a place to develop OEM circuits.

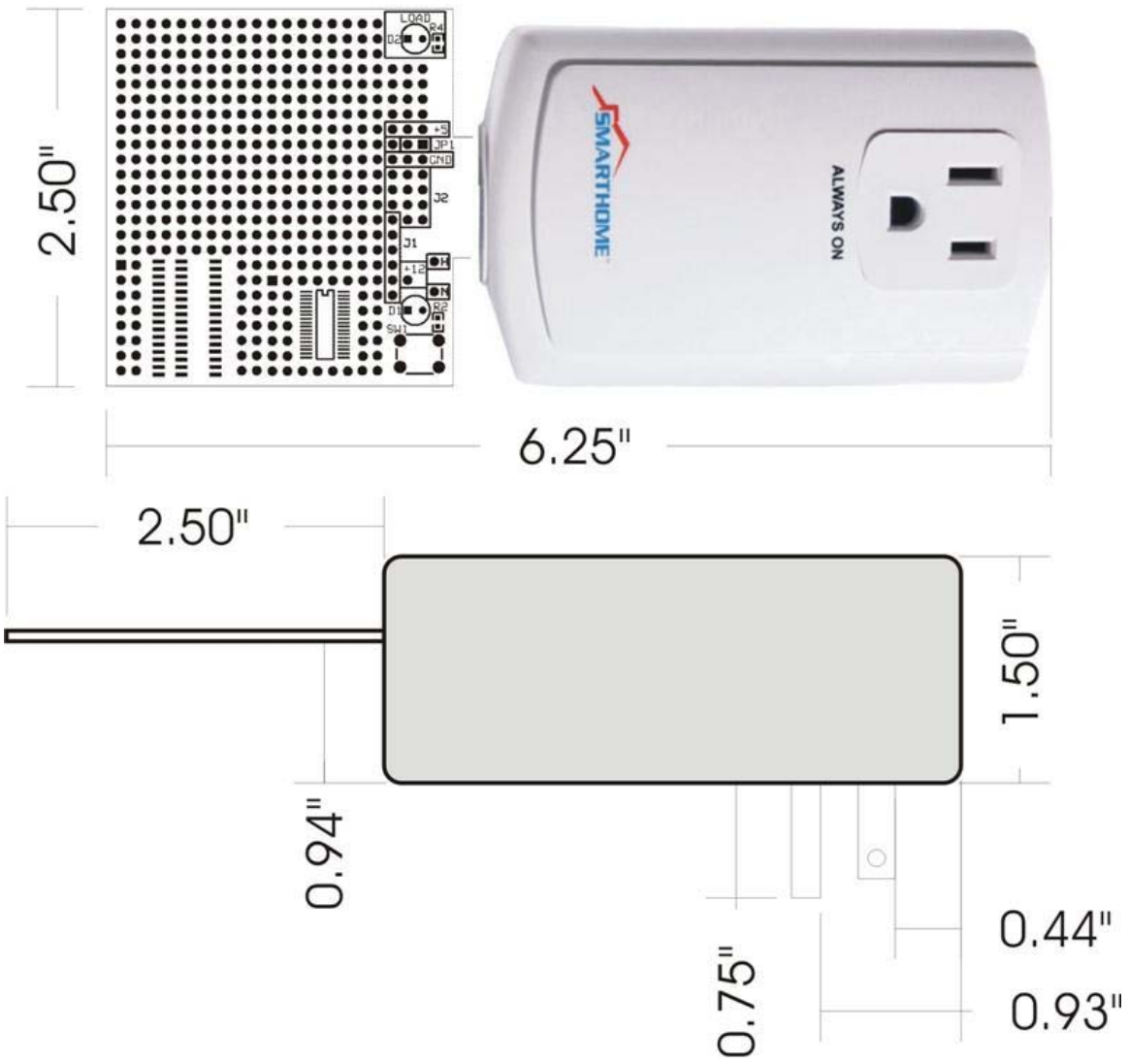
INSTEON HDK Hardware Block Diagram



HDK Physical Diagrams

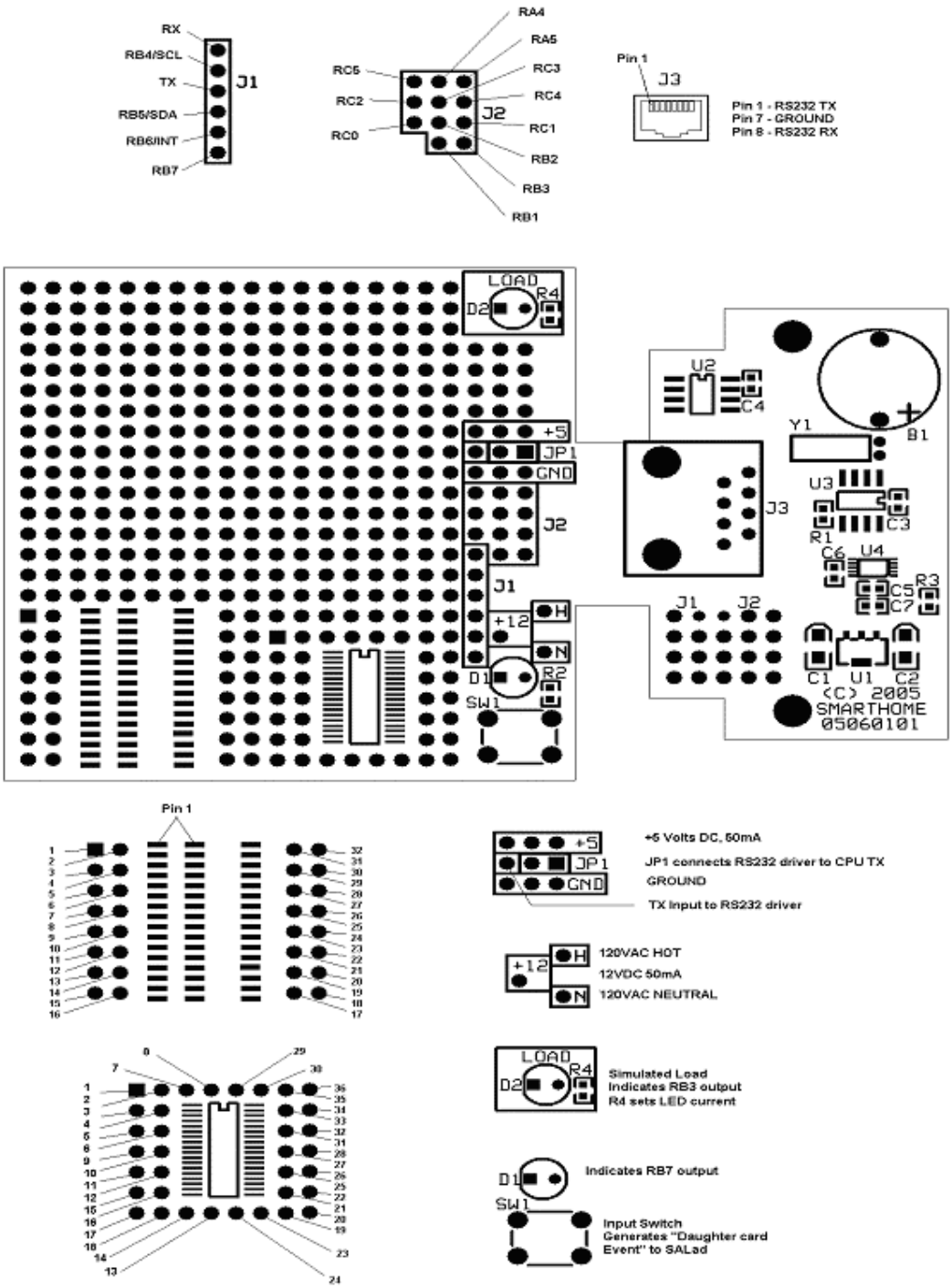
This diagram shows the physical dimensions of the HDK unit.

HDK Physical Dimensions



HDK Daughter Card

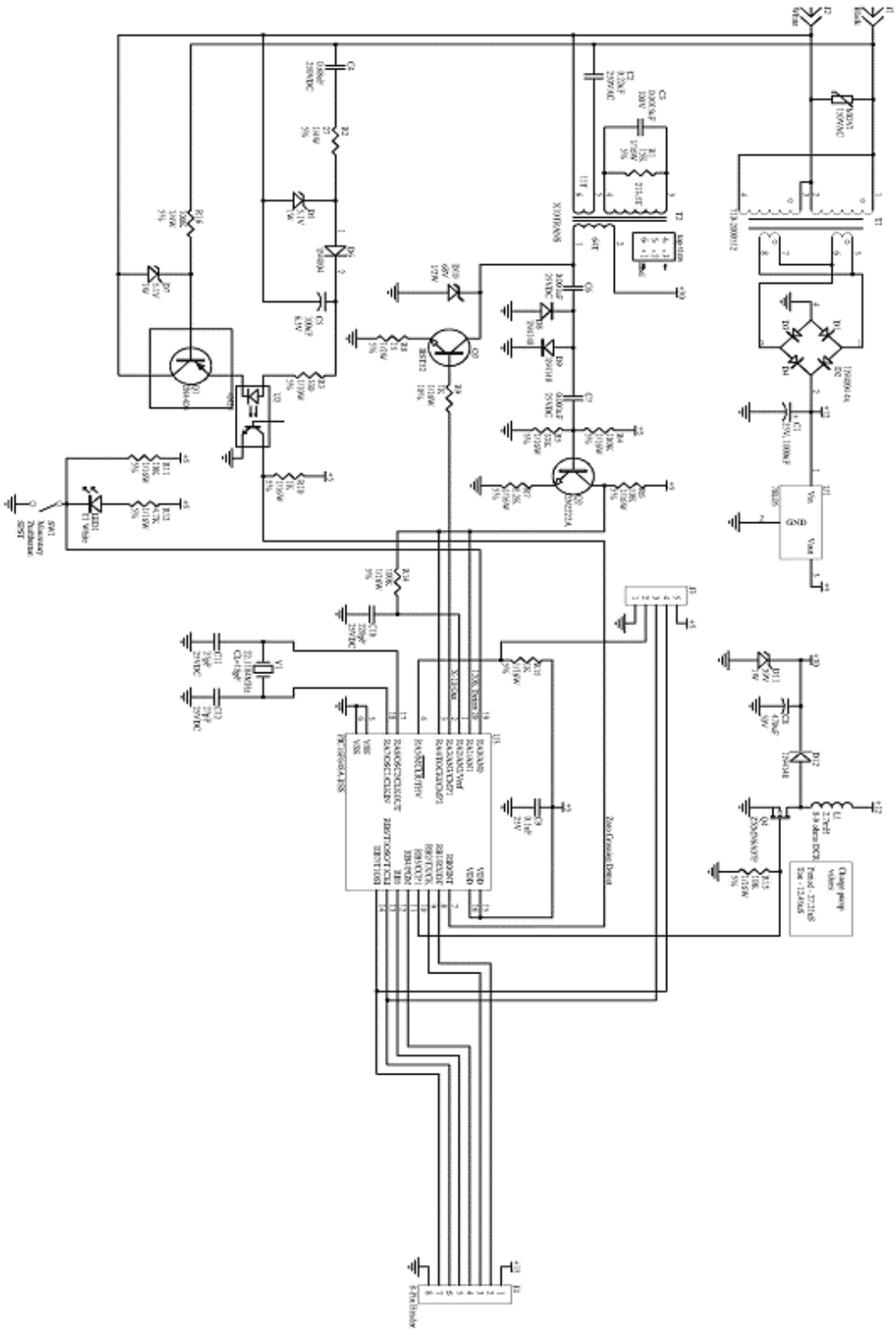
This diagram shows the HDK Daughter Board, and where signals and parts are located on the board.



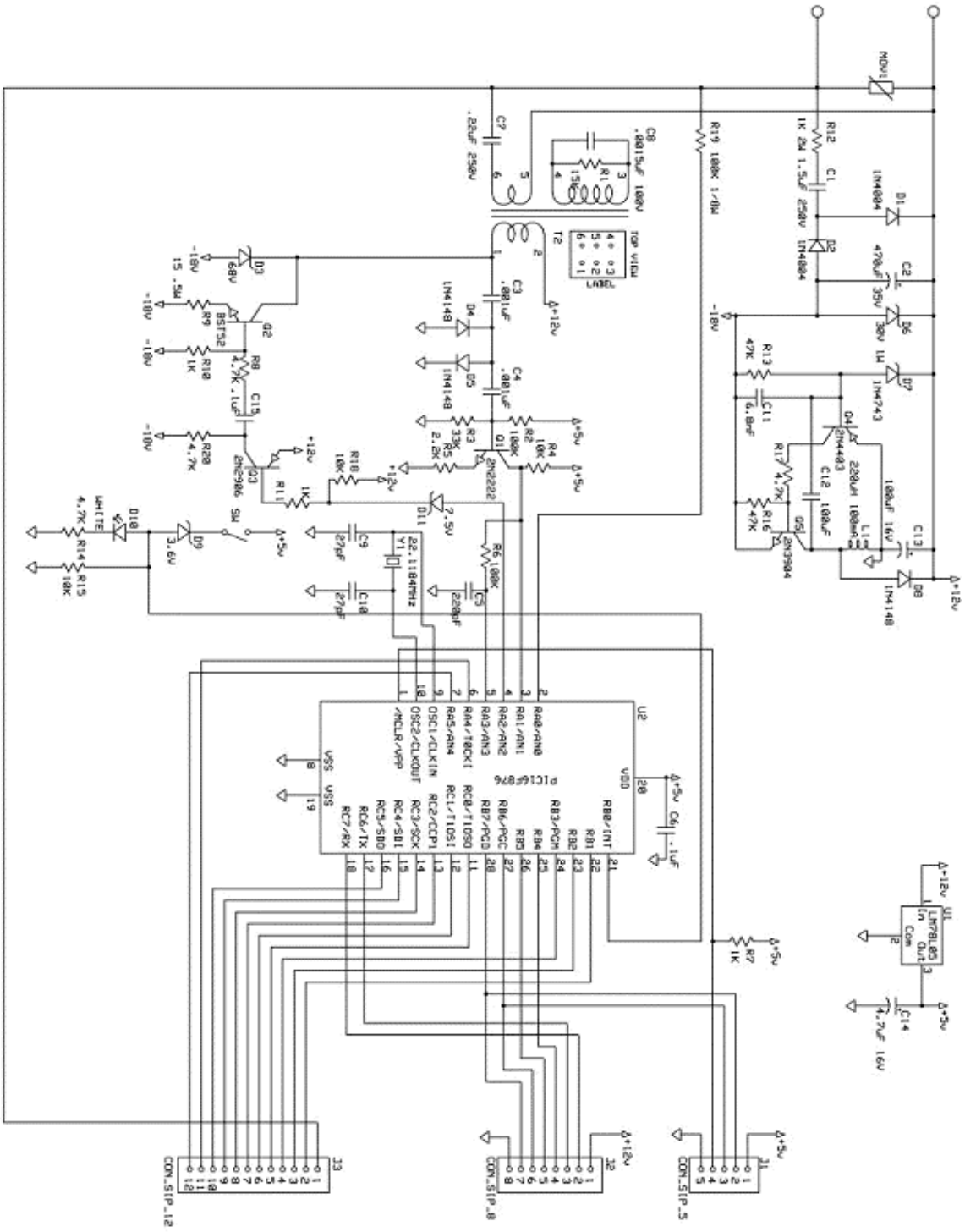
Hardware Reference Designs

In This Section

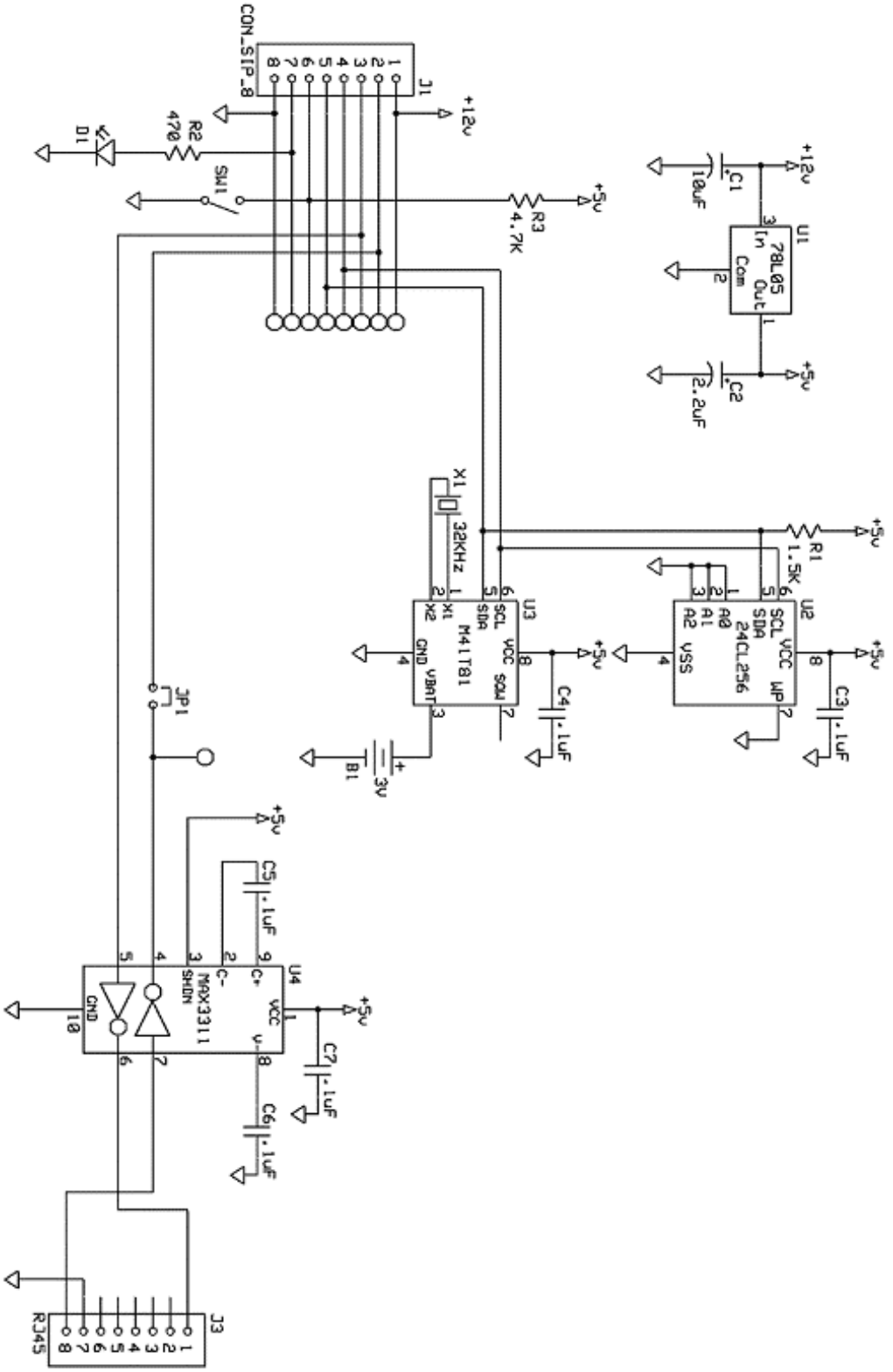
HDK Isolated Main Board



HDK Non-Isolated Main Board



HDK Daughter Board



CONCLUSION

"Everything should be made as simple as possible, but not simpler."

Albert Einstein (1879-1955)

Electronic Home Improvement™ is poised to become a major industry of the twenty-first century. Three-quarters of homes in the U.S. now have computers and over two-thirds of those are connected to the Internet. WiFi wireless networking is in 17% of homes. High-def TV is gaining momentum. But light switches, door locks, thermostats, smoke detectors, and security sensors cannot talk to one another. Without an infrastructure networking technology, there can be no hope for greater comfort, safety, convenience, and value brought about through interactivity. Homes will remain unaware that people live in them.

For a technology to be adopted as infrastructure, it must be simple, affordable, and reliable. Not all technology that gets developed gets used. Sadly, a common pitfall for new technology is overdesign—engineers just can't resist putting in all the latest wizardry. But with added performance, cost goes up and ease-of-use goes down.

Simplicity is the principal asset of INSTEON. Installation is simple—INSTEON uses existing house wiring or the airwaves to carry messages. INSTEON needs no network controller—all devices are peers. Messages are not routed—they are simulcast. Device addresses are assigned at the factory—users don't have to deal with network enrollment. Device linking is easy—just press a button on each device and they're linked.

Simplicity ensures reliability and low-cost. INSTEON is not intended to transport lots of data at high speed—reliable command and control is what it excels at. INSTEON firmware, because it is simple, can run on the smallest microcontrollers using very little memory—and that means the lowest-possible cost.

Developing applications for INSTEON-networked devices is also simple. Designers do not have to worry about the details of sending and receiving INSTEON messages, as laid out above, because those functions are handled in firmware. Application developers can use a simple scripting interface (Home Network Language™) and Smarthome's Device Manager to further simplify the interface to a network of INSTEON devices. Designers who wish to create new kinds of INSTEON devices can write the software for them using Smarthome's Integrated Development Environment and the SALad embedded language.

Although INSTEON is simple, that simplicity is never a limiting factor, because INSTEON Bridge devices can connect to outside resources such as computers, the Internet, and other networks whenever needed. SALad-enabled INSTEON devices can be upgraded at any time by downloading new SALad programs. Networks of INSTEON devices can evolve as the marketplace does.

Smarthome's mission is to make life more convenient, safe and fun. INSTEON provides the infrastructure that can make that dream come true. Anyone can now create products that interact with each other, and with us, in remarkable new ways. What an interesting world it will be!

NOTES

1. Battery operated INSTEON RF devices, such as security sensors and handheld remote controls, must conserve power. Accordingly, they may optionally be configured so that they do not retransmit INSTEON messages from other INSTEON devices, but act as message originators only. Such devices can nevertheless both transmit and receive INSTEON messages, in order to allow simple setup procedures and to ensure network reliability.
2. At a minimum, X10 compatibility means that INSTEON and X10 signals can coexist with each other on the powerline without mutual interference. INSTEON-only powerline devices do not retransmit or amplify X10 signals. But X10 compatibility also means that designers are free to create hybrid INSTEON/X10 devices that operate equally well in both environments. By purchasing such hybrid devices, current users of legacy X10 products can easily upgrade to INSTEON without making their X10 investment obsolete.
3. Firmware in the INSTEON Engine handles the CRC byte automatically, appending it to messages that it sends, and comparing it within messages that it receives. Applications post messages to and receive messages from the INSTEON Engine without the CRC byte being appended. See *Message Integrity Byte* for more information.