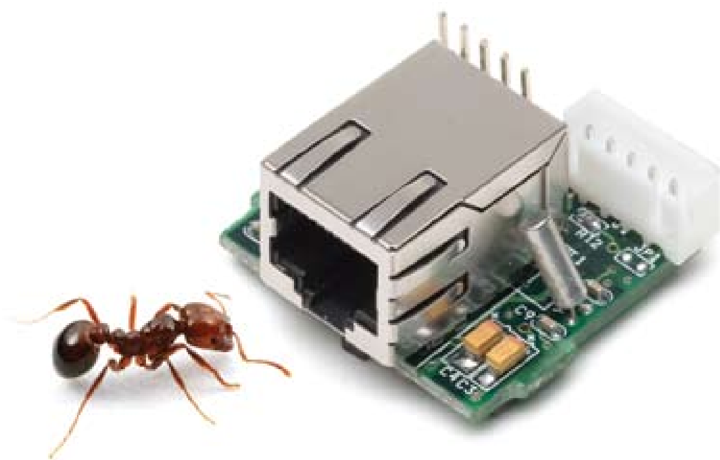
Low-Cost Web Servers



Web-Enable Almost Anything for Almost Nothing

By the Staff of Geist Technology

GEIST

Create low-cost web servers with easily maintained software. Simple circuit designs.

Low cost components; access to all source code.

"They have all the pieces and the experience. We were up and running in six weeks."

Gary Tong, Engineer, Global Steel Forge

"We get the advantage of commodity parts, and one-call support - no finger pointing."

Jonathan Briggs, Apex Power Management

"Their initial costs are low. They did the software and hardware - and did it fast."

Tully Edson, Software Engineer Keynote Systems

GEIST

Low-Cost Web Servers

How to Web-Enable Almost Anything For Almost Nothing

by the Staff of Geist Technology

Copyright © 2007 Geist Technology, edited by Gerry Cullen
All rights reserved. No part of this book may be used
or reproduced in any manner whatsoever without written
permission except in the case of brief quotations
embodied in critical articles and reviews. Printed in
United States of America. For information address:

Geist Technology
12885 Research Blvd.
Suite 108A
Austin, Texas 78750

Draft printing: November, 2007

Library of Congress Cataloging-In-Publication Data:
2007925232

Cullen, Gerard (Gerry) L., editor

"Low-Cost Web Servers"

Alternate title:
"How to Web Enable Almost Anything for Almost Nothing"
edited by Gerry (Gerard) Cullen
p cm.

Summary: How low-cost embedded processor make web accessing of data economical.
ISBN 9-781599-161730 Review copies

1. Technology - Data communications - non fiction 2. Electronics and software - Non-fiction 3. Internet - Non Fiction
PZ7 T5735 Su 2006 2006001345
{[Non Fict]

Contributors: Jason Cohen, Pedro DeKeratry, Ron McCormack, Gary Akins, David Karoly, Charlie Mayne, Patrick Nance, and Steve Gettel.

Typography & Cover art by Layne Lundstrom
Illustrations by Roman Haliziw
Edited by Shelly Kochhar, MSEE
Review Copy Printing, November, 2007
Second revision, December, 2007

www.geisttek.com

Google, Adobe, Microchip, and ITWatchDogs are trademarks.
Excel, Outlook, Fiddler are Microsoft trademarks.

All information contained in this publication is subject to change.

Contents

Introduction — 5
- Low-Cost and Very Small
- What to do. What to Avoid
- Three Rules to Build By

Typical Applications — 11
- When to Use Internet Access
- Typical Data Sources
- Power Strip Instrumentation
- Air Conditioner Example
- Control of Devices

Software Architecture — 31
- Embedded Processor Architecture
- Software Modules
- What You Don't Need
- Mistakes to Avoid

Software Modules — 53
- Operating Systems
- TCP/IP Stack
- Sensor Drivers
- E-Mail
- Threads
- Memory Management

Finch: Lowest Cost Web Server — 73
- $15 US in Components
- In-Line Code Structure
- Small Footprint
- Microchip tm Processor

Owl: High Performance Web Server — 85
- $32 US in Components
- The Owl Cube
- Operating System
- Flexible I/O Structure
- ARM 7 Processor

Web Data Transfer Methods 101
HTTP Data Transfer
Data Handshaking
Static HTML Pages
Dynamic Pages
Web GUI Techniques 119
Lightweight, Fast Web Pages
Using Cascade Style Sheets
Examples of Styles
Comparing the Two Servers 157
Hardware Comparison
Software Comparison
Physical Size
SNMP 165
Communicating with Network
Monitoring Systems
Finch: Software Specifications 173
Code Functions
Protocols
User Defined Features
Owl: Software Specifications 179
Code Functions
Protocols
User Defined Features
MicroGoose: A Sample Product 189
Low-Cost Computer Room Climate Monitor
Owl Processor
Power-over-Ethernet
Temperature & Humidity Sensors
Built in Six Weeks
Data Input and Output Methods 195
Analog Data Input
Analog Data Output
Digital Data Input
Digital Data Output

Introduction

Low-Cost and Very Small

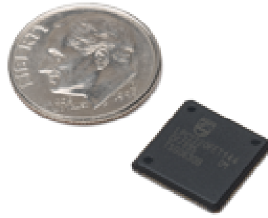
What to do. What to Avoid

Three Rules to Build By

The Case for Web Access

When you hear the word "server" you probably think of a rack-mounted computer costing thousands of dollars living in a chilled room. Most of the Internet information we see daily is supplied by millions of these computers.

But there is another type of server available; it's about the size of a cube of sugar and costs tens of dollars. Just like the traditional server it generates web pages upon request. Feed these little servers information and you can see this data from anywhere in the world. But these tiny engines can live in fairly hostile environments and don't require a staff to keep them running.



This dime-sized computer powers the Owl Web Server. No dimes where you are? That's 11/16" or 1.9 cm. It costs about thirty dimes.

These miniature servers are called "embedded" processors because they don't stand alone in a large metal box as traditional computers do. They are small and inexpensive enough to be included inside an office machine or medical instrument, which probably explains the etymology of the term "embedded."

Many manufacturers build products that contain a lot of information that would be useful to know, and I want a lot of these products to let me know, over the Internet, what's going on and how they are doing.

Here's an example. Our Texas climate is very hot and humid

about half of the year and we depend on air conditioning all year. Because I like to know how the air conditioning system is doing, I call a technician twice a year to perform several diagnostic tests. He typically reports, "Things are fine, Buddy." (You can call people Buddy in Texas with no ill effects - it's considered a default name for all males including children.) Three months later the air conditioner fails sending our entire shop home as the office temperature shoots through 95°F before lunch.

I want to be able to see from my house or a Montreal hotel room what's going on with the office air conditioner. I want to know what the inlet, discharge and compressor temperatures are, and I want to see these values graphed over a month using only a web browser. If I see the temperatures creeping up, I may be able to prevent a complete outage and save a day of work.

But the manufacturer of my air conditioner didn't put a web server in my air conditioner. Even the air conditioner I bought six months ago didn't have one. I asked the installer for this feature but the answer was a simple "no."

Why? Certainly for the last ten years the answer was "too expensive, too much trouble to design and a lot more trouble to develop the software."

It's Getting Easier

Not any more. Now, typical development time for a low-cost web server is about six weeks. The air conditioner company could easily put our embedded web server in their products and keep worriers like me happy. But no, they haven't checked on embedded server technology lately.

The process of adding a web server has changed from giving careers to half a dozen software developers to simply adding a about 30 components to an new design or running wires to an existing product to get the data.

Didn't Happen Overnight

We've built embedded web servers into dozens of products over the last eight years. Some projects were incredibly hard to get going and others were difficult to maintain, but we eventually succeeded. We decided there must be a better way to build embedded web servers without the headaches and high costs.

Our experiences gave us the insight to know what it takes to build embedded web servers fast and at the lowest possible cost. Of course, we would like to be your source of embedded processor circuits and software, but if you want to go it alone, here's what we have learned (in level of importance):

1. **Source Code:** you or your vendor must have all the source code. If you only have the compiled (binary) code which you licensed from some company you are headed for big trouble. The day will come when your software people say, "the problem's with Green Cookie's code - and they say they can't fix the problem." No matter that you paid for a software license, you are now a victim: you can't fix Green Cookie code and they won't fix it because they have other, more pressing, things to do. We learned this the hard way (several times). *Never, ever, build a product without no-hassle access to the source code. Ever.*

2. **Proprietary Electronic Components**: many manufacturers that build nifty combination packages of microprocessor and software. Three years later they change the package or discontinue their microprocessor and you get to do it all over again.

3. **One Shop Does All:** Circuit design and software development are done under one roof means no finger pointing when something doesn't work.

If you are considering using embedded web servers in your products, give us a call. We can get you there without the horror show of multi-vendor projects with poorly supported software.

Gerry Cullen - Geist Technologies

Typical Applications

When to Use Internet Access

Typical Data Sources

Power Strip Instrumentation

Air Conditioner Example

Control of Devices

Typical Applications

Embedded processors show up in all kinds of places such as medical instruments, office equipment, traffic signals, room temperature controls and industrial machines. Many of these applications would be improved with Internet access because users who are far away from them would like to know what's going on.

Embedded processors that control a car's cabin humidity control are typical of "blind" controllers. If the humidity feels right, they must be working, otherwise their work goes unnoticed and probably unappreciated. A modern car can have dozens of microprocessors.

On the other hand there are thousands of other applications where the embedded processor's functions are vital to a company's operation, such as the controller in a manufacturing machine. Operators and managers would like to know what the controller knows without having to walk up to it for inspection and to see the read-outs.

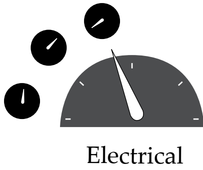
When the information contained in these controllers needs to be made available to a wide group of personnel, then the addition of an Internet (web) interface becomes a useful function. This is purpose of our tiny embedded web-servers - being able to access systems data over the web.

When Is Internet Access Worthwhile?

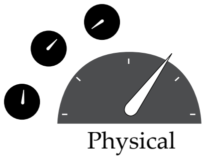
There are three requirements: first, the information needs to be accessed over a network - it's not enough to walk over to the device and see what's going on. The information contained inside the device must be available to users of the Internet, locally and, perhaps, worldwide.

Second, the device that is to be web enabled must have some type of data to be monitored. A list of typical sensor inputs to Low-Cost Web Servers is shown on the following page. These

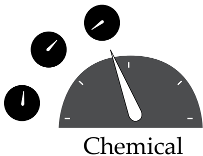
Typical Sensor Inputs to Low-Cost Web Servers



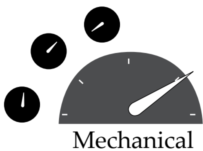
Voltage
Current
Power Factor
Watts



Temperature
Humidity
Pressure — Hydraulic / Barometric / Differential
Speed
Torque
Attitude (x,y,z)
Sound
Wind Speed
Altitude
GPS location



pH
Density
Mass Flow
Chromatograph
Color



RPM
Vibration
Motor direction
Open/Closed Position

A wide variety of sensors are available which produce digitally-formatted data. These sensors can be used as part of a new design or existing products can be adapted to add a web server.

properties are easily monitored by embedded web servers.

Third, the utility of the web interface expands greatly when the user desires to control some function of the embedded processor. For example a remote generator's operating test schedule could be easily changed to accommodate a work crew on-site who doesn't want to be bothered by the generator's roar as they repair the roof. And once the roof is repaired, the manager can remotely start the generator again, and even check the output to see that nothing was disturbed during the repairs.

The remote manager doesn't even have to travel to the generator to know if the roof leaks - a water detector attached to the embedded processor will inform him if water is on the floor.

While managing a single generator across town doesn't seem a formidable challenge, imagine if you had to manage fifty generators in six states. Having web access to each one would greatly simplify the task of knowing if the generators are ready for operation when needed and scheduled.

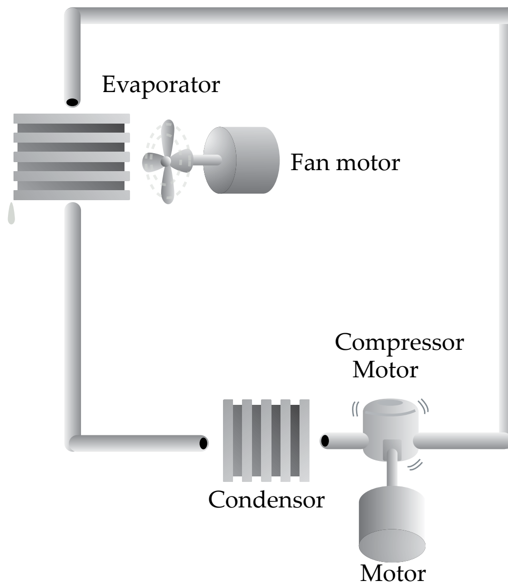
The Basic Components

The number of electronic components necessary to construct the physical aspect of an embedded web server amounts to four integrated circuits and several dozen supporting components.

Geist Technology offers two designs: one is the lowest possible cost to manufacture and has a minimum software feature set that can be used in any application. This unit is referred to as the Finch. The Finch is based on a microprocessor manufactured by Microchip, Inc..

A companion unit called the Owl, is based on a much more powerful processor using an Advanced RISC Machine (ARM) architecture. We chose this because the ARM consortium makes the process of selecting a microprocessor straightforward by standardizing the design and giving the reassurance that if one processor vendor stops producing their chip, another vendor's chip can be substituted with minimal changes.

Air Conditioner - No Web Monitoring



A typical air conditioner system without any monitoring capabilities. A common office air conditioner can cost $10,000 US and be vital to keeping production going. Unless you have a modern (expensive) building management system, you have no idea how this critical machine is performing.

The Owl uses a mid-range ARM processor produced by Philips Semiconductors. The Philips line of processors offers a wide range of capabilities.

Air Conditioning Monitoring and Control Example

The illustration on the facing page shows how a typical refrigerant air conditioning web monitor might be constructed. The

Monitoring		
Temperature 1	Return Air	Room Leak
Temperature 2	Discharge Air	Efficiency
Temperature 3	Condenser Temp	Refrigerant
Current 1	Fan Motor	Filter
Current 2	Compressor Motor	Too hot
Control		
Relay	Aux. Air Conditioner	On/Off
Display		
LED panel	Scrolling conditions	Status

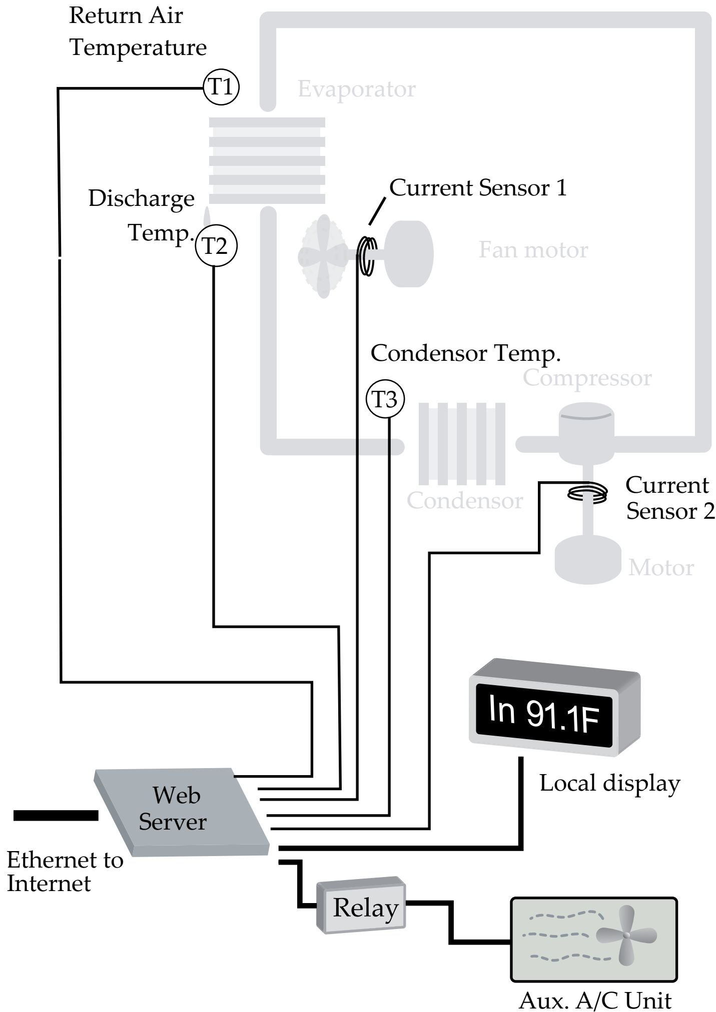
following table shows the data sampling and control points:

By adding these monitor, control and display points as shown on the following page the user can open a browser and see the current state of the air conditioner. By looking at the graphs showing historical data, the user can see if any trends are apparent that the machine may be deteriorating or need maintenance. The graphs are a key part of the web page display. If you have only numeric values, the steady change in a machine's performance from week to week is not apparent.

Now take monitoring one step further because the internal logic of the microprocessor allows an e-mail to be sent to building management notifying them that the room has become hot and the back-up air conditioner has been turned on.

And from your condo in Aspen, *you* can see all this happening

Adding Sensors and Web Access to the Air Conditioner



A variety of sensors and a small display are added to the air conditioner. Even a small auxiliary air conditioner control has been added. By observing the graphs of the unit's operation, the user has a much better chance of predicting failure.

using only an Internet browser.

The typical cost of electronic component parts to do all this is probably less than $200.

How much more valuable is the air conditioner if it has a web interface? Depends how hot your climate is and how critical is temperature control to your processes. If you are storing critical medicines in a chilled room, it would be very important.

Power Strip Monitoring - Watch the Costs

Electrical power costs continue to rise. A price of $0.10 US per kWh (kilowatt hour) is not uncommon in many countries. In legacy computer rooms and large data centers, power can be a major expense if not the most expensive item. Google released a study showing that their highest corporate expense was electricity[1].

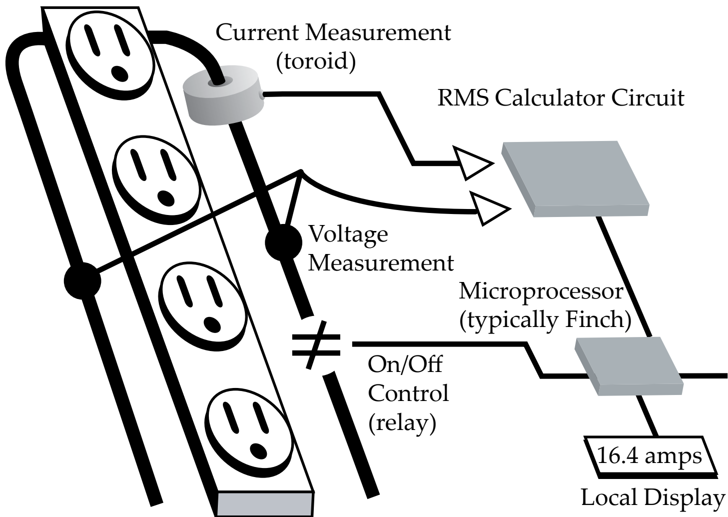
Considering that a typical power strip can consume 20 amps continuously and that the yearly cost of power continues

Power Strip	Electrical	Consumption
20	amps	Fully loaded strip
120	volts	Typical voltage
2400	watts	Watts = Volts * Amps
2.4	kiloWatts	Divide by 1000
24	hours	Full time operation
5.76	$US/day	Cost per day
365	days	Multiply by days
$2,102	$/year	Expensive power strip!

to escalate it would be valuable to know what each power strip is consuming. But learning how much power a strip is drawing is not

1 *There are a number of sites describing this ongoing concern about spiralling energy costs at Google. We recommend you use Google to get the latest about Google's efforts.*

Instrumenting the Power Strip



The drawing shows a single power receptacle instrumented for current, voltage and control functions. Watts and power factor can be derived from knowing these values. A relay can be used to remotely control the power on/off via the web. An embedded processor could control dozens of receptacles in a single power strip. A local display is shown attached to the microprocessor as an optional component.

a simple task.

With conventional non-metered power strips a common method is to modify the power strip by having an electrician insert a temporary in-line ammeter, power the equipment up, and read the meter. Once the meter has been read, the process is reversed and the ammeter removed.

There are two difficulties with this method: first, the power has to be removed temporarily from the equipment usually irritating the computer manager, and second, the reading is only accurate until electrical equipment is removed and/or replaced. If the power strip is only partially used the reading is accurate only until the next piece of equipment is installed.

When power strips are conveniently and continuously monitored, users report dramatic improvements between power consumption and cost reduction.

By monitoring the individual receptacle, the facility manager can determine which devices are energy hogs and replace them with more power-efficient equipment. By knowing each receptacle's current usage the data center management can administratively account for energy usage and distribute energy costs back to each device.

Add to this the cost of keeping the equipment cool which is typically as much as the cost of powering the gear. This could translate into a $4,000 US a year to power and air condition the devices on one power strip.

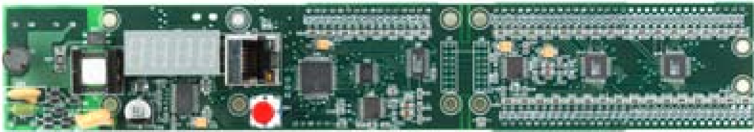
With one power strip consuming so much money, it makes sense to monitor the quantity and the quality of the power.

Instrumenting the Power Strip

There are a number of variables that will be useful to know about the operation of each power strip.

The diagram on the opposite page shows the basic components of a typical power meter assembly using the Finch-

Typical Multi-Receptacle Monitoring Board



This Finch-based board will monitor up to 40 receptacles for voltage and current (RMS) and provide a local display. The circuit board will easily fit into most existing power strip housings. The number of receptacles to be monitored can be easily customized.

Using a Toroid For Current Monitoring

An inexpensive toroid measures the strength of the magnetic field in a wire to determine the current in the wire. A Finch can typically measure the current in dozens of these devices. A single toroid can measure the aggregate current in a power strip used on an individual power receptacle.

based web server component.

- Total Amps (current consumed by entire strip)
- Individual Amps (current consumed at each receptacle)
- Volts (level)
- Amps
- Power Factor (quality of power)

The Importance of RMS Power Measurement

When doing power measurements, the current and voltage measurements must be done in phase, Watts = Volts x Amps and both values are measured at the same instant in time. In other words, the current and voltage must be measured at the same time going into the same device, and the multiplication is done before the power is supplied to the device (not after.)

Power usage in power strips should be measured in using the Root Mean Square (RMS) method, not the peak-to-peak method as used in many earlier power monitors. RMS is a fundamental measurement of the magnitude of an AC signal.

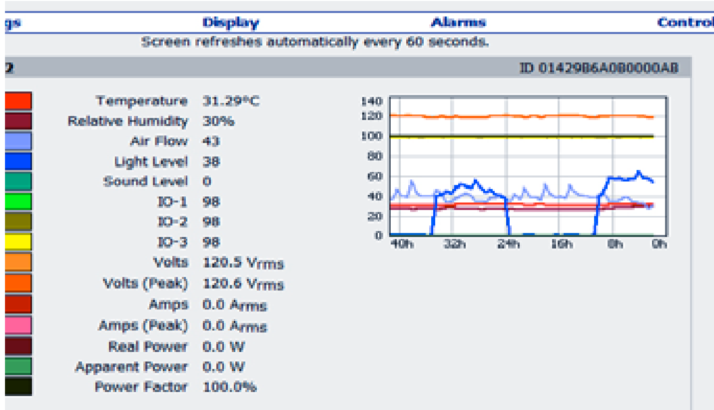
The RMS value assigned to an AC signal is the amount of DC required to produce an equivalent amount of energy in the same load. This is the real amount of power consumed. The peak-to-peak method as used in many earlier power monitors is invalid as it does not give usable results. All of Geist power metering circuits use RMS measurements.

Graphing Power Data

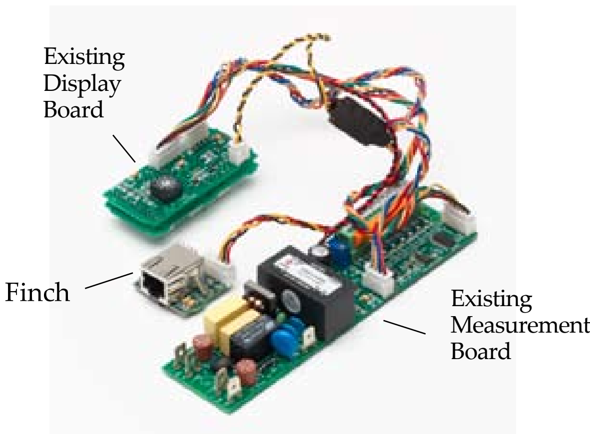
Many data center managers believe electric power is a steady, uniform value, but this is not the case. Electric utility power constantly fluctuates in magnitude, frequency and quality.

In one office location our small server room's power approached brown-out conditions almost nightly as generators switched off or as industrial plants came on-line. Initially, we

Typical Power Graph



Graphs are available in the Owl-based power meter. Here, a typical power strip graphs not only the power values but also the climate of the computer room using optional sensors. The graphs clearly show trends such as voltage sags or spikes.



The photo shows two printed circuit cards which were part of an existing power distribution product. The cards provide an LED readout of voltage, current, and watts consumed by the power strip. The small device between the two larger cards is the Finch web server which converts the serial data on the boards to a web interface.

suspected intermittent processors or aging disk drives as the cause of the nightly re-booting of servers, but upon adding a graphing-capable power monitor, we discovered the real cause was the nightly sagging of the voltage on the electric utility's power lines.

These brown-out conditions cost us time and production. We added a UPS to solve the problem but it too disconnected from the same low voltage conditions. After several requests, the electric utility fixed the problem, but only after we showed them the trends. We needed to be armed with this information to get the utility to understand the situation.

Today, graphing and viewing current and historic data, has become an essential component of monitoring devices.

Climate Monitor - Product Example

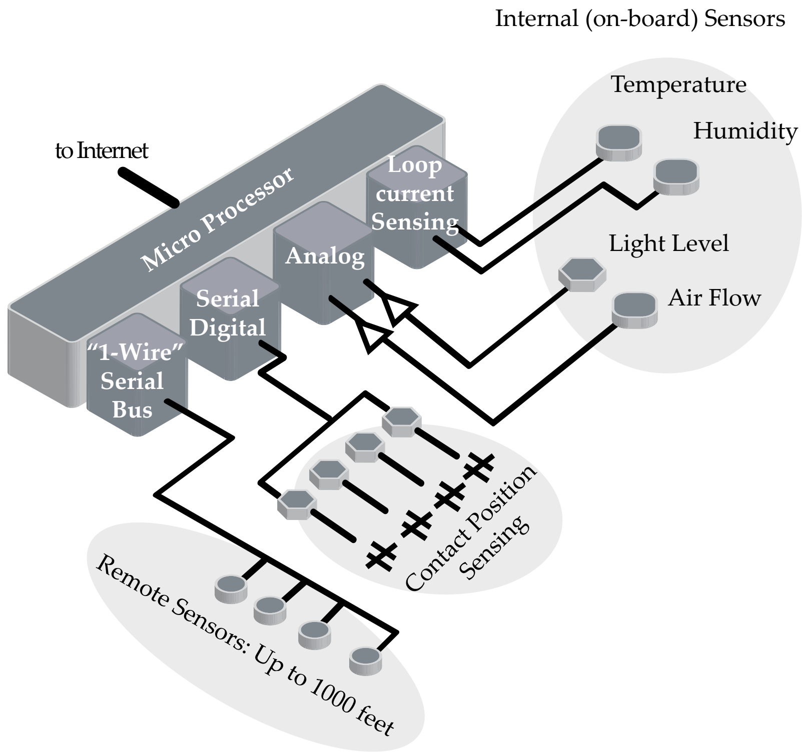
Both the Finch and Owl devicess have been used to make a variety of climate monitors. The chapter "Sample Products Using Embedded Processors" has more detail and photographs of these products.

Climate monitoring is a good example of an application where multiple types of sensors provide a wide variety of inputs. Applications for climate monitors are endless and include computer server rooms, animal testing cages, pharmaceutical testing chambers, wine storage buildings, freezers, cell phone control buildings, and data centers.

Common sensors include:

- Temperature
- Humidity
- Light level
- Air Flow
- Door position (open/close contacts)
- Sound level
- Water on floor
- Video Camera (external webcam)

Climate Monitor Application



Multiple Types of Remote Sensors
- Serial Connection Flow

A Finch or Owl with a variety of sensors makes a low-cost climate monitor suitable for monitoring computer rooms or other rooms where a stable environment is needed. Note the use of both on-board and remote sensors.

A climate monitored with these sensors provides enough information to the facility manager to give him a good idea of the physical condition in the remote room. With graphing capability added, the manager can see the present condition of the room as well as the last few weeks of monitored operations data.

The example used in this application uses the Owl web server using internal and external sensors. This techniques makes the embedded web server even less expensive because the cost of the remote sensors amortizes the cost of the embedded web server over multiple sensors.

Control Applications

Many applications require that something be controlled remotely. In the air conditioner application, the auxiliary (back-up) air conditioner is turned on by a remote user through a web interface. Many users have concerns about hackers finding ways into the remote sites and changing settings, or, in this case, turning on (or off) the remote air conditioners.

There are three ways to cope with this:

- Only control harmless devices
- Use multiple password level protection
- Use encryption (SSL, multiple types)

The harmless approach could be the air conditioner application. A hacker could turn on the air conditioner which would increase the power bill and make the room colder which is a relatively harmless event. A hacker who turned *off* the auxiliary air conditioner when the main air conditioner was broken becomes a dangerous situation. If the device being remotely controlled is not critical, perhaps no security is required.

Passwords are the most typical method of security and multiple passwords (admin, user level 1, user level 2, etc.) are the most common. Hackers have to go to extremes to learn passwords which could require hardware monitoring equipment. In the

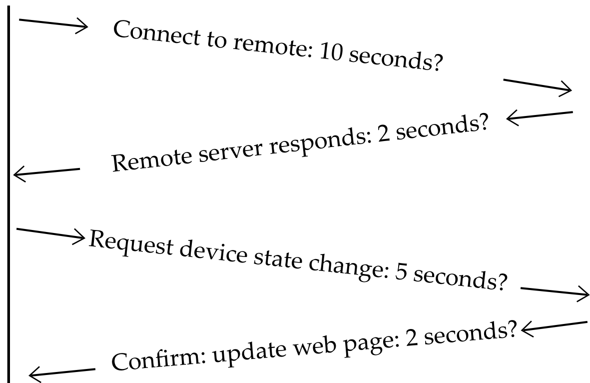
level of equipment is not harmful or not dangerous to personnel if hacked, password protection is likely to be adequate. In our experience, unhappy or terminated employees are the most likely to hack web-accessible control operations.

The third and most protective method is encryption. This is bank-level transaction security and can be implemented in many levels. While this encryption defeats transmission-cable "sniffing" intruders, the system is still vulnerable to password misuse.

Levels of Encrypted Security

Finch:	None
Owl:	Secure Socket Link - 64 bit
	Secure Socket Link - 128 bit

Internet Delays



Waiting for the light to turn on a thousand miles away? Internet packets can have lots of delays. The example shows almost twenty seconds of delay between a request for a control change session. Users who expect almost instant response face disappointment on a heavy internet traffic day.

Use of SSL algorithms imposes a considerable processing burden on the embedded processor and only the Owl-class processor can implement this security.

Delays in Internet Communications on Control and Sensing

You flip a light switch and the lamp illuminates with no perceptive delay. This is not the case with Internet-based devices - the amount of delay is indeterminate. If you have a remote lamp that is connected to a web interface there are multiple delays possible.

The timing table diagram on the previous page shows possible traffic delays on a busy Internet day. Products developed on high-speed Intranets and then moved to dial-up Internet access can disappoint users - plan on slow Internet speeds.

Feedback on Control Signals

We once built a remote coffee pot Internet control in order to save going downstairs in the office. A pre-loaded coffee pot could be remotely turned on by a second-floor user. Since there was no way to know if the pot actually did turn on, a user walked downstairs to confirm the pot had started heating the water.

The message was clear, a commercial grade coffee maker would need an internal temperature sensor to see if the water had begun heating: entering a command is one thing, knowing that the machinery actually responded is another. (Control engineers call this a feedback loop.)

If your remote coffee pot is located in another country, you need to know if it responds to your control signal.

Since relays are a common way of controlling devices, Geist Technology has a patented method of determining whether the relay has moved. The problem with a relay is that the electrical part may work but the mechanical part may not.

By using an electrostatic sensor, Geist's method can tell if the relay physically moved. This method can easily be integrated into a

circuit board for less than ten cents per relay.

Is Internet Web Control Useful?

All remotely controlled functions have some element of risk. Consider the missile silos during the cold war - there were dozens of these silos with enough automation to remotely control everything from the warm and cozy offices of Washington, D.C. One phone call and the missiles would launch, maybe. The military placed people there to assure the command was carried out. Feedback was and is the key element.

Thankfully, most applications aren't this demanding. If you can wait a few minutes to see if your remote command has been carried out, Internet control will be adequate for your application. If you have confirmation that the control event was carried out, like the coffee pot temperature sensor or the missile guys watching the rockets fly, so much the better.

Gerry Cullen - Geist Technology.

Software Architecture

Embedded Processor Architecture

Software Modules

What You Don't Need

Mistakes to Avoid

The Embedded Processor Race: More for Less

Embedded processor architecture is a moving target. In just a few years we've watched processor speeds, memory capacities, and the availability of third-party tools, double several times over.

It's very much like the desktop computer explosion of the 1990's. Every 18 months we'd be blown away by new capacities and speeds, all for a fraction of the cost we'd expect. Nowadays the desktop computer world has slowed – we're happy with Windows XP (now four years old) and 2-3 Ghz processors (also four years old).

Now the embedded world is exploding. Four years ago a 20 Mhz processor and 128K of memory was a state-of-the-art, low-cost package; now 200Mhz processors and 16M of RAM is cheaper. Four years ago, just having a web page at all was a miracle; now IT administrators look to every object in the data center and demand web pages, security protocols, integration to console software, email and SMS support, XML data streams, and remote monitoring and control.

In short, IT administrators look at their iPhone and wonder why it can't be the same with their power strip / cabinet / thermostat / UPS / building control / door / lock / whatever.

All this in four years.

In this chapter we give a high-level overview of what goes into modern embedded processor design. Later chapters will delve into the nitty-gritty of how each of those components work.

Outside the Black Box

Let's start with the requirements of the system as seen by users of the device. Knowing what other systems and human beings expect from your black-box device sets the perspective for what we're doing and also drives the design of the insides.

There me be a lot to digest here, but it's important because the IT administrators expect all these features. And every IT guy

has his favorite protocol, so if you don't support everything – and support it well – you'll risk alienating whole groups of IT people. And if your competitor supports something and you don't, that could turn the tide of the sale.

IE, FireFox, and Safari- the User's View

Nowadays, he first way humans want to interact with any device is through web pages. This has become the standard way of accessing everything from printers to power strips. HTTP, and its secure counterpart HTTPS, are the protocol standards for all browsers.

And it's important to support all modern browsers. Gone are the days of one-browser-to-rule-them-all; FireFox is now the preferred browser of geeks, IE still wins on most desktops, and the recent rise of Mac OS X and Apple's porting of their browser to Windows puts Safari on the map as well. They're all used heavily, so having poor support for one of them will piss off a lot of people.

It's critical that the web server be fast, efficient, and simple. The first impression of your device will come from its web page, so everything depends on getting it right. If it takes 20 seconds to load, your device feels sluggish, unresponsive, and perhaps untrustworthy. If the page looks like "My First Website" from 1994, your device feels unfinished and unprofessional. If your web page depends on having Active X installed, most users will send it back right away -- seasoned IT guys know that's the path to security problems and firewall hassles.

Mail - Who Speaks Anymore?

Consider an IT administrator who lives in San Jose. He's in charge of 300 power strips for a data center in Omaha. Each power strip has a web page with a wealth of information – Amps, Volts, Watts, Phases, Deci-coulombs, and who knows what else. His real concern, however, is which power strips are about to blow a breaker. He knows the breakers pop at 20 amps, so he probably chooses to keep things below 15 amps.

How does he monitor all of these strips? By opening up Internet Explorer and visiting all 300 pages, every day? Impossible.

What he needs is an alert – a proactive notification when any power strips reads more than 15 amps. And the most common form of notification is email.

Email is great because everyone has it, everyone knows how to read it, filter it, take it with them on laptops, get messages on their cell-phones or pagers or Blackberries or iPhones, back it up, log it, and so on.

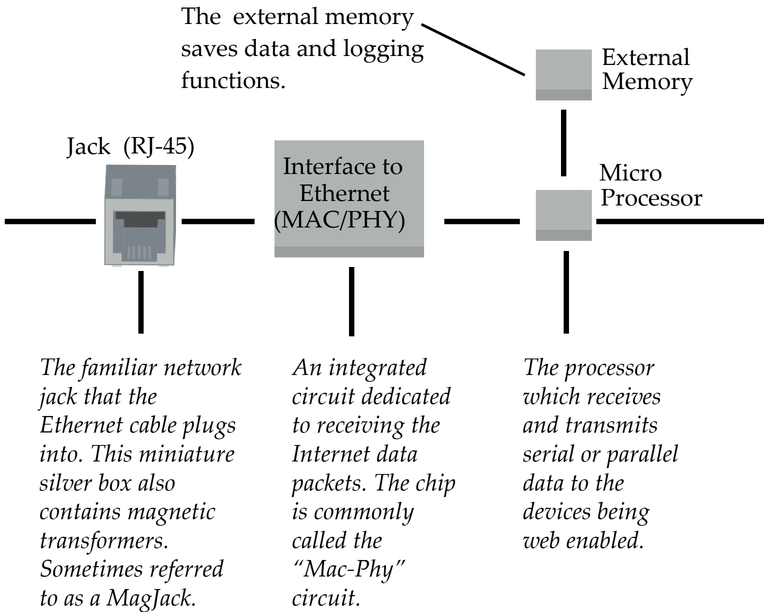
And IT guys know even more – they can set up special addresses that trigger other things, so perhaps an email about 16 amps could trigger another email to the local administrator asking what's going on, plus a special "high priority" email to the IT guy's PDA.

There are lots of email protocols on the Internet in general, but few of them are needed for embedded applications. To name a few: POP3, APOP, SMTP, ESMTP, IMAP. Fortunately, most of the protocols involve features that a simple notification application doesn't care about. All we need to do is send an email to various addresses.

Turns out you need three protocols. SMTP is the standard "Send a message" protocol, and for some installations that's all you need to configure. You specify the SMTP server (a machine that IT administrators will already have set up for general office email), list some email addresses you want to notify against, and off you go. ESMTP is a cousin – the E stands for "Extended" – which you sometimes need because of increased security.

POP3 is the protocol for checking email. It seems counter-intuitive that you need to get email from an embedded device – we just need to send email, right? – but as you may have noticed when your Outlook client was set up, some email servers require you to first check your email before you can send. More of that security stuff. So because some systems are locked down that way,

Components of an Embedded Processor



These three components plus the supporting electronic components and a power supply form the structure for web-enabling a device. The microprocessor shown uses internal memory.

sometimes the embedded system actually does need to check his own mailbox before he can send!

All these servers, protocols and rules get complex, and we cover it thoroughly in a later chapter.

Another unexpected use for email is in SMS (Short Message Service text messaging) paging. This is the "text messaging" service that has been common in Europe and Asia for years and has recently caught on in America as well. Fortunately it's easy to send SMS messages using email protocols. So we get SMS alerts "for

free" as long as we do it the right way.

Big Brother is Watching

So far we've been talking about things that are familiar to most Internet users today. But IT administrators have other tools and protocols that you probably haven't heard of before. They can be mystical if you ask an IT guy about it – most can't be bothered to explain their complex and intricate details – but really these are quite understandable if you just want to know how they work.

More importantly, if you walk into an IT guy's office and your device doesn't support his special protocols and tools, you just lost the sale. Web pages and emails are nice, but not enough.

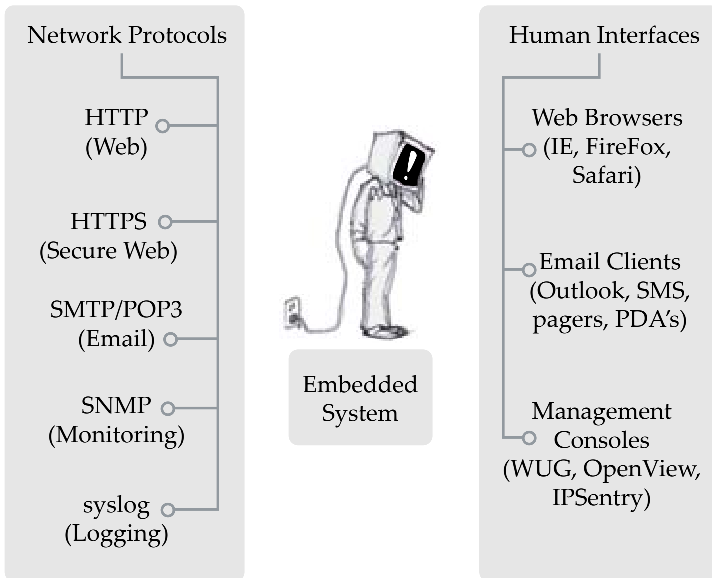
The first special tool IT managers have is called an "SNMP Console" or "Management Console." If you've ever walked into a Network Operating Center (NOC) you've seen one of these – one of the functions of such a console is to display current status and alerts on big screens.

These are very useful. You can see things like network bottlenecks, devices that are down or in trouble, warnings and alerts from all around the data center, current network traffic levels, and more. And you thought they put up those big flat panels "because we can." (Well, that too.)

Already all of the IT guy's devices are talking to this central console. Depending on how advanced the data center is, this can include web servers, email servers, network switches, database health, UPS battery levels, power usage, temperature settings, door positions, water levels, and a variety of other inputs.

And since all this data is coming into one console, not only can you display it, you can react to it.. NOC operators can set rules about when to notify people of alarms, how to log errors, when to escalate, and so on. Some console software can even take action, like turning on a backup air conditioner if it gets too hot. This is monitoring, alerting, and control for all devices in the data center.

How Humans Talk to Embedded Processors



The world of the embedded processor. It must conform to the network protocols and also give the user a variety of means to access the data. The two aspects are complex and critical to the system developer.

No wonder the IT guys love it!

Of course if you walk in with your device and it doesn't plug into their console, you're sunk. You need support for a protocol called SNMP, and this is harder than it sounds. Here's why.

SNMP is the protocol used by all those consoles. Some support other protocols, but all support SNMP. SNMP stands for "Simple Network Management Protocol," and that in itself an inside joke among network gurus. Why? Because it's anything but simple.

In 2003 we first added support for SNMP in our WeatherGoose product line. We tested it in-house four different

ways: What's Up Gold (one of the most popular commercial consoles), ipSentry (another popular, low-cost commercial console), netSNMP (the gold standard SNMP open-source tool for Linux), and Ethereal (the premier network packet-sniffer that can decode data and tell you if the data is malformed).

We beat up (software term for stress testing) our code until it passed all the tests. We were coming up properly in all four test scenarios. So we released it.

Over the last four years later, we've had to revise the SNMP protocols no fewer than 17 times. Why? Because in the process of supporting over 12 other SNMP consoles, we discovered a few things. The standards documents? They're just guides. Every console did whatever it felt like. Some could read these fields, others couldn't, and others just crashed. Some needed alerts to come in this way, others that way. Some had bugs with the encoding of certain numbers so we had to make sure we encoded them in special ways.

It Was a Nightmare

We learned a lot during those years. The good news is that, today we have that knowledge encoded in our own embedded SNMP server. We don't have to worry about it any more. We have the know-how now, but I wouldn't want to go through that again!

Before we leave management consoles, you need to know about another little protocol called syslog. Not an acronym this time, just a contraction of "System Logger." This is a Unix protocol for collecting logging information from multiple sources and giving the IT administrator a single place to store the log messages on disk, organize those files, rotate old files into storage, and filter which logging messages he really wants to keep.

Although not as prevalent as SNMP, syslog is another way that you can keep many IT administrators happy because you're plugging into their system rather than making them figure out how

to get data from your device.

Plus syslog has another advantage – it allows us to log all sorts of diagnostic information that is invaluable when tracking a problem. Most of the time it's a network problem or system configuration problem, but whether the bug is external or internal it's great to be able to get detailed log information from a device in the field.

Things You Don't Need

There are other protocols on the Internet. Generally it's best to support as few protocols as you can while providing all the same functionality, because more protocols means more code to support, more chances for bugs, and more ways a hacker might be able to get at the device.

Lots of devices support FTP as a way to transfer files, usually to upload software updates. Being able to load software updates in the field is critical, but why support a whole new protocol just for that? FTP doesn't play nice with most firewalls, so it has additional strikes against it.

Instead, we allow users to upload software updates through the web interface. It's as easy as attaching a file to an e-mail from a web-based e-mail client like Gmail, Yahoo or Exchange.

Another common one is TELNET, one of the oldest protocols still active on the Internet. TELNET provides text-based access to the device. If you've ever used a modem to connect to a BBS (bulletin board service) from the 80's or 90's, it's like that. If you haven't, just saying "Like in the 80's or 90's" should be enough to give you the idea.

Geeks like text-based interfaces because it allows them device control and information access without using pesky things like mice and web browsers. Instead they can write programs that interact directly with the device.

That's an important attribute to keep – geeks need to write

their programs! But with the advent of the web page it turns out you don't need text-based access any more. Why? Because as long as you design your web pages properly, they are just as easy to read by a machine as by a human. And nowadays there are many free tools that assist geeks in making their automated programs, so it's actually easier to control the device through the web page than through TELNET!

On top of that, there's a new data format in town called XML. Without going into details, it has become the lingua franca for machine-to-machine communications (as opposed to machine-to-human, like a web browser). So if your device supports XML, you're also supporting the geeks with their automated tools. We support XML as well, but this is a technical subject covered in a later chapter.

Four Rules - We Learned Them the Hard Way

Now that we've identified what the unit has to do from the outside, what are the design considerations for the insides? Years of experience in embedded design have led us to some important design rules that will save you millions of dollars of time and mistakes.

The first rule is: ***Only use commodity parts***. That specialized system-on-a-chip component might seem like the perfect fit for your project, and maybe it is… today. But what happens a year from now?

On one project the part manufacturer went out of business just as the first 100 units rolled off the assembly line. The product died before it ever saw the inside of a catalog.

Another time the manufacturer declared the product line obsolete. We had to scramble around reseller's inventory bins to keep our units going until we could design around it – at exorbitant prices, of course. We lost money on units for four months and spent two months redesigning the board.

Still another time the manufacturer was bad at inventory control, so sometimes we'd be stuck without being able to build product for weeks at a time. Sure we could have inventoried more parts ourselves, but this was an expensive part and we didn't have hundreds of thousands of dollars laying around for inventory.

So now we buy only commodity parts. That means multiple vendor sources and multiple resellers, so if (or when) any one place goes out of business, we're not hurt. And because parts are being made in the millions, not thousands, there's always someone who can overnight 10,000 units if necessary.

The Second Rule - Trusted Vendors

The second rule is: ***Only buy from trusted vendors***. That doesn't mean "buy from large companies," in fact often it's the large companies that treat you poorly and the small, starving companies that will dig you out of a hole, even if it's your fault.

This means developing relationships with the vendors. Will they answer your phone calls? Do you have access to technical people on their side so your engineers can talk to their engineers directly without sales people or "Level 1 Support" slowing things down and mis-communicating? Are they passionate enough about making you successful that they'll go out of their way to help?

We've developed great relationships with several vendors, so we know we have help if we need it.

The Third Rule - Source Code Access

The third rule is: ***You have to have all the source code***. This doesn't mean everything has to be open-source (as in, you have the source code and it's completely free), but you must have easy access to the code.

There are many reasons for this. The most important one is that you can fix any bug. We've had a vendor on the WeatherGoose Climate Monitor project whose code had more than twenty open bugs – twenty! – that we couldn't fix (because

we don't have the source code) and which the vendor refused to fix. (Who knows why, maybe they think adding features is more important than making the features work).

It's called a hostage situation. Except that sometimes you can't even pay them to do it. You're a victim, at their mercy. Not fun, especially for your own software engineers who are helpless and having to ship code they know doesn't work.

On the Finch project we also had twenty bugs, but this time we had the source code. We fixed every one of them without help, and currently we have no known open bugs. That's peace of mind.

The other reason is that it helps to be able to read the code. Sometimes the vendor code works, but it's hard to understand what it's doing. Allowing the developers access to the insides can save hours of sleuthing. Otherwise it's like taking your car to the mechanic and saying, "Fix this car, but you're not allowed to open the hood or put the car up on jacks."

The Fourth Rule - Automated Testing

The fourth rule is: ***All components must be testable through automation.*** This is a rule that the non-embedded software development world has already embraced, but for some reason it's all too common for embedded developers to ignore automated testing completely.

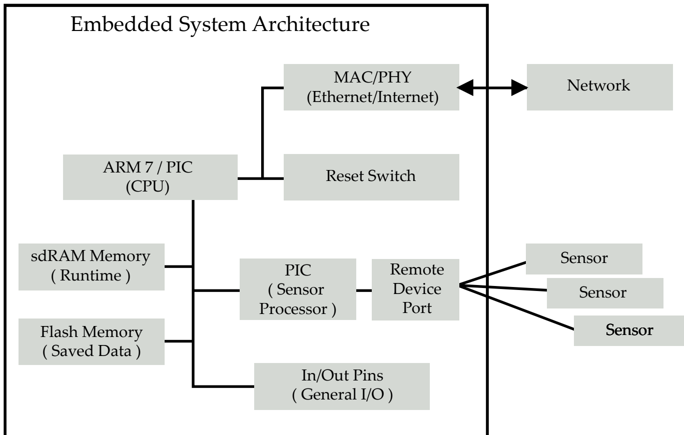
The most basic form of software test is called the "unit test," and NASA's been using this for years to achieve zero-bug software. The idea is that you isolate a tiny bit of code and rig up a test harness to exercise it. For example, let's say it's a bit of code that takes a number and gives you back text that makes it an ordinal number. So 1 would become 1st, 2 would become 2nd, 1011 would become 1011th.

You would write some other program that would run that code against various inputs. The inputs I just gave would be an example. This separate test code compares the actual output with

the expected output, and alert the software developer if the two don't match.

The trick is that because the test is another program, and not a human, it can be run at any time, like every night, and before any code goes out the door. This is important for several reasons.

First, it's great when bug reports come in. Let's say my program used the following rule: If the input ends in the number 1, use "st," if ends in 2 use "nd," if 3 then "rd," and otherwise



A typical embedded processor architecture is shown. Each rectangle represents an integrated circuit. Don't let the simplicity lull you into thinking that these are unsophisticated devices. A typical Owl system can have over 30,000 lines of code

use "th." This would work for most tests. But then QA reports a sighting of "11st" in the web page. Oops! 11 is a special case! That's "th" even though the last digit is a 1.

So here's how you fix it. First you make new tests for 10, 11, 12, 13, and anything else you can think of that might also be broken (e.g. 111, 1011). Run the unit tests. Of course they will fail – that's

good, because now they're actually testing these cases properly. Now you fix the code until the tests pass again.

You've enhanced your tests and you're sure the code works now. But because there were other tests as well, you've ensured you didn't also break cases that used to work. For example, in fixing this bug I might have thought "Oh, if it ends in 1 just make it 'th,'" but then that would break the tests for 21 and 101, so the tests prevent me from breaking one thing as I fix another.

The second major thing it does is enable safe code rewrites. For example, let's say we run this ordinal program a lot, and it turns out it's really slow. So we decide to rewrite that code to make it faster. Fine, but that's exactly the kind of thing where you end up putting a bug in the code because you forgot some corner case. Here's where the tests come in again – if you know your tests cover all normal and corner cases, you can rewrite code with the confidence that you're not putting in new bugs.

In our code we have thousands of tests. Just having thousands of tests that pass the gives developers comfort. It's nice to know that at least in thousands of normal cases, the code definitely works! And it's nice to know you can rewrite code when you need to without fear. Otherwise the code base can become stale out of fear of change.

Cracking the Black Box

So much for the outside of the black box and general design requirements. Let's crack it open and see what we've got.

This is a high-level discussion intended to communicate the big picture of what it takes. In later chapters we'll describe in excruciating detail exactly how all this works.

The brain of the system is the CPU, or Central Processing Unit. All subsystems eventually connect to this central processing plant. When you think about "loading the code" into the system, this is where it goes.

For the Owl family, the CPU is an ARM 7. This is possibly the most popular processor in the industry due to its low cost, flexibility, and extensive software tool support. The Finch family uses a Microchip PIC processor, less powerful than the ARM 7 but much less expensive and it also comes in several packages.

All computer systems have various kinds of "memory," the place where data is stored permanently or temporarily. On common desktop computers you're probably familiar with at least two kinds of memory – hard disks and RAM. The disk is where things are stored "forever," even when you power off the computer; whereas RAM is for "working memory," which programs use like scratch paper while they're running and which is completely lost when the computer is turned off.

You might be wondering why we don't use hard disks for everything – after all, it's all bits and bytes, and hard disk data remains after powering down, so isn't it better? It turns out there are engineering trade-offs that make it useful to have two kinds of memory. Hard disk memory is very slow but quite inexpensive. RAM memory is thousands (yes, thousands) of times faster, but costs hundreds (yes, hundreds) of times more.

Embedded systems have the same kinds of needs and trade-offs, just on a smaller scale and with different products. You still need RAM, but less of it. Our typical Owl system comes with 16 megabytes of RAM (your laptop probably has 500-1000 times more RAM) and instead of a hard drive we keep permanent memory in a Flash memory chip of 16 megabytes (your laptop probably has 2500 times that much memory). So it's smaller, and the names have changed (i.e. "Flash" instead of "hard drive,") but the functionality and the trade-offs are the same.

In fact there are even more kinds of memory than this, with deeper technical and cost considerations, but we'll delve into this in a later chapter.

The internal part that talks to the Internet is called the MAC /

PHY (pronounced “Mac Fy”). This is actually a set of components including a tiny microprocessor. A full discussion of this system and how it communicates appears in a later chapter.

Dinosaurs Showed the Way

For systems that support external sensors, we often employ an additional out-board processor. This is akin to the Brontosaurus who had a small brain in its butt to complement the bigger (but still not that big) brain in its head. The butt-brain was useful because without it, it took too long to get signals from the head to the tail.

Similarly, we use a small, cheap, Microchip PIC processor as our out-board butt-brain. This processor is fast enough to do the time-sensitive protocols needed to communicate with many types of sensors. This also allows the CPU to do things like service Internet requests and do signal processing without messing up the delicate timings often needed with sensors. The CPU and PIC communicate with a simple, human-readable serial protocol, so all communication is easily understood and debugged.

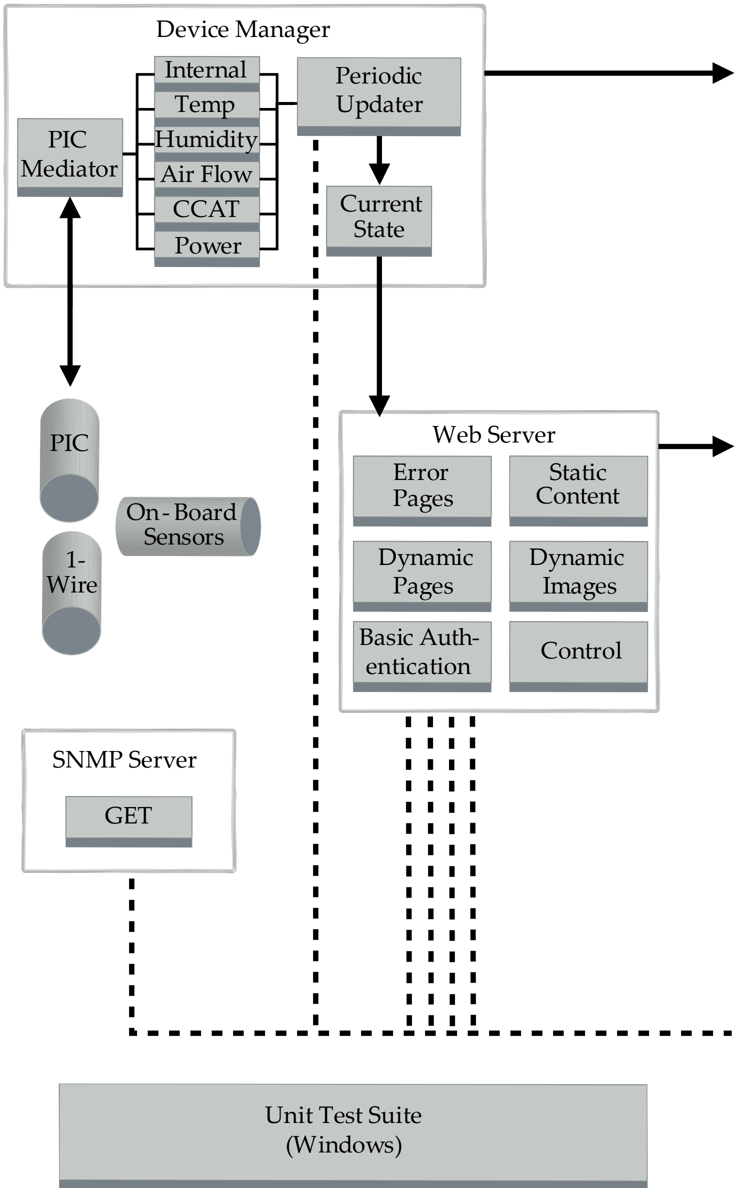
The Finch architecture looks the same as the ARM, but with fewer and cheaper parts. Here’s a high-level comparison:

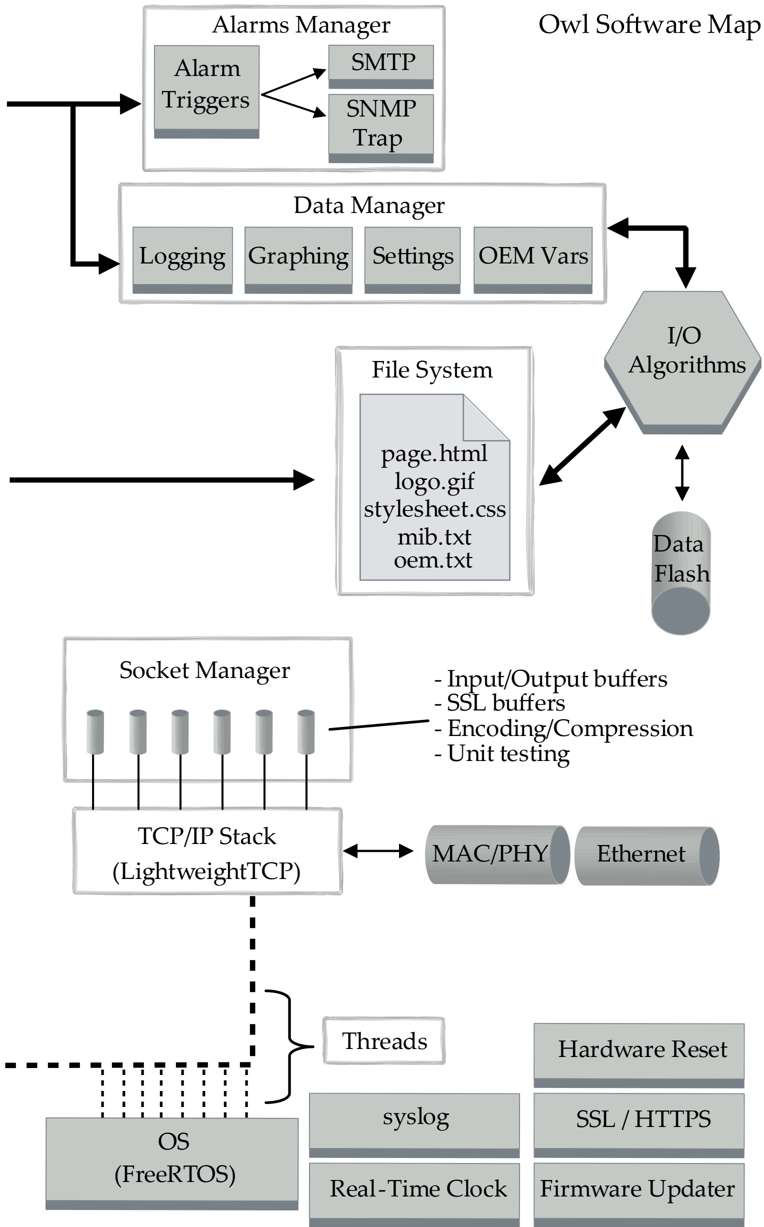
	Owl	**Finch**
Processor	ARM 7	Microchip PIC
CPU Clock Speed	70MHz	25MHz
Cost of Components (no assembly)	$33	$15

The Soft Spot

Fifteen years ago, embedded systems meant 95% hardware with 5% software. Now it’s the reverse. Software runs everything – schematics often show little more than various software-powered microprocessors talking to each other.

What does the software look like inside the Owl? There’s a lot on this diagram shown on the previous two pages, so let’s take it


Device Manager
PIC Mediator
Internal
Temp
Humidity
Air Flow
CCAT
Power
Periodic Updater
Current State
PIC
1-Wire
On-Board Sensors
Web Server
Error Pages
Static Content
Dynamic Pages
Dynamic Images
Basic Auth-entication
Control
SNMP Server
GET
Unit Test Suite (Windows)


Alarms Manager
Alarm Triggers
SMTP
SNMP Trap
Owl Software Map
Data Manager
Logging
Graphing
Settings
OEM Vars
I/O Algorithms
File System
page.html
logo.gif
stylesheet.css
mib.txt
oem.txt
Data Flash
Socket Manager
- Input/Output buffers
- SSL buffers
- Encoding/Compression
- Unit testing
TCP/IP Stack
(LightweightTCP)
MAC/PHY
Ethernet
Threads
Hardware Reset
OS
(FreeRTOS)
syslog
SSL / HTTPS
Real-Time Clock
Firmware Updater

piece by piece.

The Prime Mover

The “operating system” or OS is the heart of any computer system. In our case we’re using the open-source FreeRTOS (Real Time Operating System) package.

The OS controls which programs execute at what time. Some code runs all the time, like a web server, but only needs to be active occasionally, such as when it’s actually servicing a web request. Other code runs at regular intervals, such as periodically checking the values of all the external sensors.

These different programs, executing at various times, are called “threads,” and it’s the OS’s job to manage them in an efficient and fair way. Efficient means we shouldn’t waste processor time on threads that don’t have something to do right now, and fair means we shouldn’t allow any one thread to take up all the processor time, which stifles other threads that need to run too.

The TCP/IP Stack

The “Stack” is programmer jargon for the part of the code that implements the basic Internet protocol of TCP/IP. Although we discussed many protocols before (e.g. HTTP, SMTP, SNMP), all of these actually ride on top of the same horse, called TCP/IP. We have much more on all this in a later chapter.

The stack is responsible for running all the Internet protocols. This means keeping up with web browsers, email servers, and management consoles. It means making sure connections aren’t dropped, data is re-transmitted if the other end didn’t get the message, and other more complex work. The TCP/IP stack is a complex beast, so we want to insulate the rest of the code from its activities.

The “Socket Layer” is another subsystem who’s job it is to provide a simple model of the Internet for the rest of the code. For example, the web server can tell the Socket Layer that it would like

to be notified when a web browser wants to talk.

The web server thread can rest until this event occurs. Once the connection is made, the web server receives data from the web browser in an in-box, and can talk back by putting data into an out-box. The Socket Layer is responsible for delivering data to and from the in-box and out-box.

Thus, although the TCP/IP stack has to do complex work to manage all these bytes flying around, the Socket Layer provides a simple in-box/out-box mechanism for the web server, so the web server doesn't have to contend with anything difficult.

The Socket Layer can do other things too. For encrypted protocols (such as HTTPS, the secure web browser protocol), the Socket Layer can do all the encryption on the spot. So the web server still puts its bytes in the out-box and the Socket Layer encrypts them before sending them out on the Internet.

Finally, the Socket Layer provides an invaluable service in quality assurance. After all, since the web server just knows about its boxes, it doesn't know whether there's a real web browser on the other end or whether it's actually a special test harness that we've created! The Socket Layer can mimic a web browser without an Internet connection. This means we can set up all kinds of complex, automated test cases, and make sure our web server is doing the right thing. Assuring high-quality quality control before our units ever leave the building.

Servers, Servers, Servers

For every one of the Internet protocols described above, we have a program called a "server" to manage the particulars of that protocol. There's a web server to handle HTTP requests, an SNMP server to handle management consoles, and an email server to handle sending SMTP alarms.

These are all described in detail elsewhere in this book. What they have in common is that all send and receive data through

the Socket Layer, and all have one or more threads from the OS to control when they execute. Some, like the web server, have more than one thread. This allows us to service requests from more than one web browser at the same time, which makes everything appear to run faster for the end user.

External Devices and Sensors

External devices and sensors is a complex topic on its own, so an entire subsystem is devoted to it. Our subsystem was built from years of experience with over 30 different kinds of sensors from different vendors. We can handle all sorts of protocols, rules, dynamically-changing external sensors, error conditions and more.

Error-handling is especially difficult with remote sensors. All kinds of things can go wrong – the sensor could be bad, the cabling to the sensor could be flaky, electromagnetic noise in the room can introduce errors in the dataflow, a connector could be bad, and a human can unplug and plug in sensors at will. However we have a system that ensures only "good" data is ever saved, and that we properly recover from any kind of error that could occur along the way. The user sees only good data.

In the Owl system we also store sensor data in an internal log. This log is used for several things, but the most interesting use is graphing. Graphs show trends of a sensor over time, which is often more informative than the actual sensor value.

For example, when an A/C unit goes bad, you usually see signs months in advance. If you look at a graph of temperature, humidity, and airflow of an A/C duct, you see these nice patterns of rise and fall as the condenser kicks on and off. When the system starts to go bad, the pattern becomes more erratic, swings more, and the values don't stay in nice boxes. With such a warning, IT personnel can replace the unit during scheduled down-time rather than in crisis mode when it finally gives out completely.

Jason Cohen - CTO of Geist Technologies

SOFTWARE MODULES

Operating Systems

TCP/IP Stack

Sensor Drivers

E-Mail

Threads

Memory Management

Processor Functions

An embedded web server performs many operations throughout its lifetime. Besides presenting a web interface, some of its other functions might include: sending e-mail, synchronizing its clock, querying a variety of sensors, managing user preferences, responding to various network requests, and even updating its own firmware code.

Each of these tasks are distinct, however there is one major component in common with them all - the hardware. All the tasks execute on the same Central Processing Unit (CPU), memory, and Input/Output devices (I/O). While it is possible for a skilled programmer to design each task so that it allocates its own processing time and memory space, and interfaces directly to peripheral devices, this method introduces complexity, redundancy, and is prone to error.

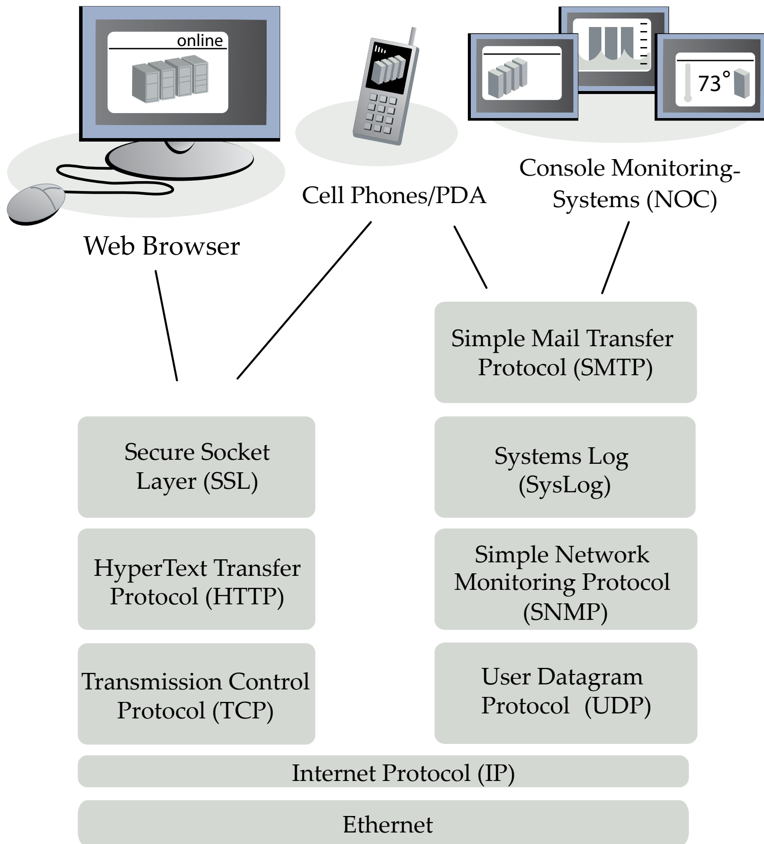
The Operating System as Manager

The software that facilitates in managing these resources is termed the operating system (OS). Operating systems are available in a wide range of capabilities and features, but in general they are all geared toward the same goal: allowing user applications to define how to use the system's resources in order to solve a particular problem. A system designed with an OS should at the very least obtain two basic benefits: efficiency and convenience.

Efficiency is possible because the operating system has knowledge of what all user applications are doing. Contrast this to a single user application that has limited knowledge, if any, of what other user applications are doing. It stands to reason that a user application is in an inherently inferior position regarding resource management decisions.

In other words, an operating system can perform operations efficiently at the system level, whereas a single user application simply cannot.

Protocol Stack for Internet Communications



These are the building blocks that enable web and e-mail communication. Each requires thousands of lines of program code. Both the Finch and the Owl contain these components although the Finch does not have SSL or SysLog due to memory limitations.

The second basic benefit, convenience, is an equally important and powerful advantage that the OS brings through the use of abstraction. Abstraction allows the operating system to hide away the complexities of underlying hardware and software mechanisms so that user applications get the OS provisions they need without getting bogged down in unnecessary details.

Remember, the goal of an operating system is to manage hardware such as the CPU, memory, and I/O devices, and make them easily and efficiently available to user applications. Some operating systems focus solely on one aspect, while others balance a combination of both. Selecting which operating system to use depends on the software goals and the kind of hardware it is expected to run on. In general, the more varied the user applications and hardware are, the more flexible and complicated it must be.

The gamut of operating systems range from the nonexistent, to the basic, to the fully managed. Each of these operating systems are described very briefly.

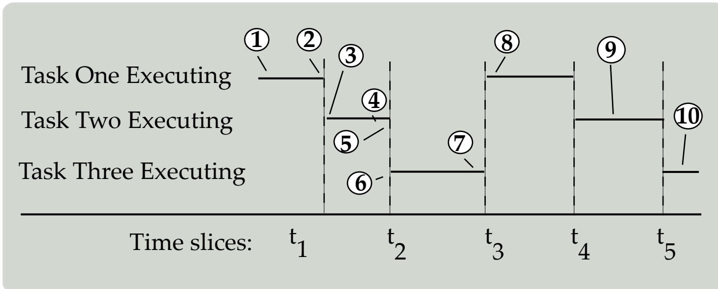
Using No Operating System (Finch)

A nonexistent operating system is a viable option if your software requirements are simple or the hardware is simply incapable of running an OS. Naturally, without an OS, no major conveniences through abstraction nor resource-wide efficiency advantages are possible. However, they are not necessarily needed because ideally the software is simple enough to execute commands serially.

That is, each function the software needs to perform executes correctly even though it is always executed in the same order relative to other software functions. Further, since the hardware is accessed directly, there are no complex software maneuvers for operation nor extensive timing coordination with other software functions.

Examples of nonexistent OS setups are prevalent in 8-bit

Tasks Running in Processor



The status of the tasks is indicated by the number above;

** At (1) task 1 is executing.*

** At (2) the kernel suspends task 1 ...*

** ... and at (3) resumes task 2.*

** While task 2 is executing (4), it locks a processor peripheral for it's own exclusive access.*

** At (5) the kernel suspends task 2 ...*

** ... and at (6) resumes task 3.*

** Task 3 tries to access the same processor peripheral, finding it locked task 3 cannot continue so it suspends itself at (7).*

** At (8) the kernel resumes task 1.*

** Etc.*

** The next time task 3 is executing (9) it finishes with the processor peripheral and unlocks it.*

** The next time task 2 is executing (10) it finds it can now access the processor peripheral and this time executes until suspended by the kernel.*

microcontrollers such as the Microchip PIC which is the processor on the Finch.

Basic Operating System Features

A basic OS provides the minimum feature set required for classification as an OS, namely, the concepts of tasks and synchronization. Systems of moderate complexity cannot usually be arranged in a strict linear order for execution, due to timing requirements of their software components. If these systems are executed without the time allocation management an operating system provides, their software timing requirements would be very difficult, if not impossible, to ensure.

Operating systems that provide this time allocation management are called realtime and use an abstraction called a task to achieve it. The task abstraction provides a convenience to programmers that allows them to write task code as if it were the only code executing on the CPU. The task code may be tailored specifically to its goal without worrying if and how its timing requirements are met. In this way a programmer is free to think of tasks as software components that are executing simultaneously on the system. Of course, the tasks are not actually all executing at the same time, although they appear to do so through a method called timesharing.

Multiple Tasks and Synchronization Semaphores

The other abstraction convenience provided by a basic OS is synchronization. Sometimes multiple tasks must share a resource that may only be accessed by one or a few entities at a time. Tasks must have a way of knowing if they can safely access the protected resource.

This is referred to as synchronization between tasks. In the jargon of operating systems, synchronization is implemented using something termed a semaphore. Each semaphore is created with a count that signifies the maximum number of entities that may use it at any given time.

A semaphore with a count of one is so commonly used that it has its own term, a mutex. The basic idea is that any code that accesses a protected resource is encapsulated by a semaphore so that synchronization is ensured.

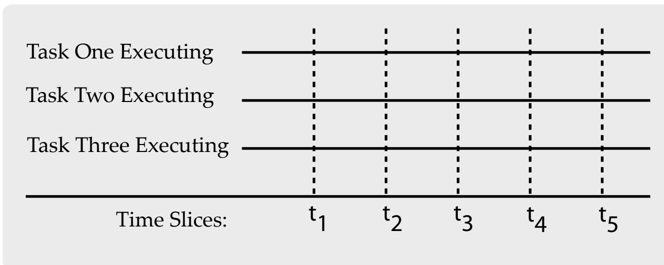
Examples of basic operating systems are uC/OS-II and FreeRTOS.

Threads and Fibers

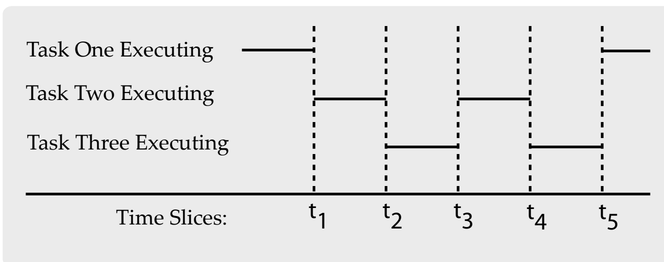
A fully managed operating system provides significantly more features than just task and synchronization management.

Time-Slicing Tasks

All tasks appear to be executing simultaneously



But each task only gets a fraction of the processor's time



The speed of the embedded processor plus the ability of allocation time segments makes multiple tasks appear as if they are all running simultaneously. The amount of time allocated to each task can be adjusted.

First, direct hardware access by a user application is simply never allowed. Hardware driver infrastructure allows an interface for user applications to access the hardware. Besides providing protection from illegal hardware access, the infrastructure is flexible allowing support for a variety of hardware devices. Second, the concept of a task is more flexible in an advanced OS; processes, threads, and fibers are the terms used to describe these specialized tasks.

Processes are heavy duty tasks that have much more system resource information associated with them. This extra information allows the operating system to make even better decisions on how best to efficiently fulfill process needs. Threads and fibers allow user applications to timeshare code fragments within a process while sharing the parent process's contextual information. And lastly, memory is abstracted away and tightly controlled.

Memory Management

The operating system uses a technique called virtual memory that allows a process to execute as though it has the maximum amount of system memory available for its own personal use. The operating system translates virtual memory addresses to real hardware memory addresses invisibly and keeps tabs on them to ensure processes access only their designated memory areas. Examples of fully managed Oses are the Windows and Unix based operating systems.

Armed with some basic knowledge of operating system offerings, it is now possible to select an OS based on a project's requirements. What follows is a discussion on OS selection and implementation for two web server platforms: Owl and Finch.

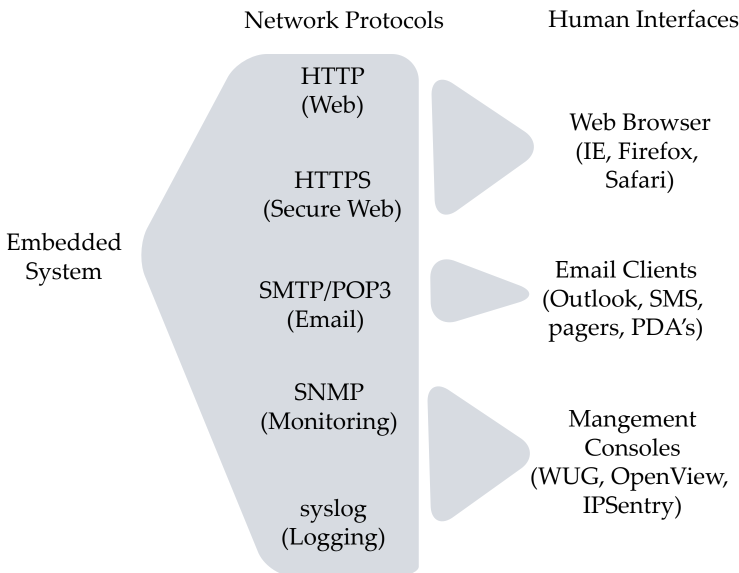
Owl Platform

The Owl platform's primary goal is to provide moderate software and hardware features at low cost. The Owl software is expected to provide a multi-threaded web server capable of serving

up to four requests simultaneously and at the same time perform other duties such as fulfill SNMP requests, send out e-mails, synchronize to time servers, query domain name servers, manage a flash file system, log data, and interface to a realtime clock.

Furthermore, it must perform more intense tasks like encrypting HTTP/web server transactions using SSL. All in all, many different kinds of tasks are expected from the Owl, and for this reason the use of an OS to help manage these tasks is undoubtedly needed. The only question is which class of OS to use. Although it is tempting to immediately select a fully managed OS for use because it has more features, these features come at a price. The biggest drawback being the resources required to support it.

Communication Protocols



The Embedded Processor has multiple tasks to accomplish just to maintain Internet communications. Add to this the task of collecting and processing the data that needs to be displayed, and the tasks are considerable.

The Owl contains 4MB of RAM and an effective 4MB of Flash memory. A fully managed operating system, even one specifically trimmed down for use in embedded systems, will still eat a significant portion of these resources.

Other drawbacks include the Owl CPU's lack of hardware for taking full advantage of all OS features and it requires a more complex installation. Overall, the features provided by a fully managed operating system are simply not needed for the Owl because it runs and interfaces to predetermined tasks and hardware.

End-users are not expected to run and interface arbitrary code and hardware, so the general purpose nature of a fully capable OS will go mostly unused. The reasonable choice is to select a more basic operating system.

FreeRTOS - The Owl's Operating System

FreeRTOS was selected for use on the Owl platform. This OS is free and comes with full source code already ported to many target hardware platforms. In fact, a specific FreeRTOS implementation for the Owl CPU and compiler is readily available. It has a very small memory footprint and requires few CPU hardware features for it run.

These hardware features are a timer interrupt and a software interrupt both of which the Owl CPU already has. In concert these two features work together allowing effective CPU timesharing. The hardware timer is used to implement a particular FreeRTOS configuration called preemptive task switching. This translates to forcibly switching out the running task with a new pending task every specified period of time.

When a hardware timer triggers an interrupt, FreeRTOS understands it is to save the state of the current running tasks and restart another task that has been waiting for its prescheduled time. By adjusting the trigger interval of the hardware timer it is possible

to control the maximum amount of time any one task may execute per task switch. A typical trigger interval is ten milliseconds and is commonly called a timeslice. With this the method the CPU seemingly executes tasks simultaneously.

The second hardware feature, the software interrupt, allows timesharing to occur even more efficiently.

Allocating Resources - Semaphores

Take the case of a task that need only perform operations every few minutes: if it uses up the entire timeslice each time it switches in, then thousands of tasks switches are wasted over the span of those idle minutes. That is wasted time that is possibly better spent by other waiting tasks.

Instead of waiting for its timeslice to expire, a task may trigger a software interrupt and give control back to FreeRTOS, effectively saying that it yields the remainder of its current timeslice. The same method is also used to communicate the desire to yield a specified number of future timeslices. In this way FreeRTOS ensures that a task is switched in only when it is the correct time to do so and executes only as long as needed.

The other basic feature provided by FreeRTOS, semaphores, is also implemented by manipulation of interrupts. Recall that the purpose of semaphores is to allow only a specific number of tasks to access an encapsulated code section at any one time. When the the time comes for a task to enter that protected code section, the hardware timer interrupt is disabled. Then and only then is the semaphore's count checked to see if access to the code fragment is permitted.

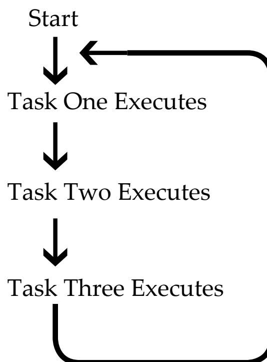
If access is granted then the semaphore count is incremented, the hardware timer interrupt is re-enabled, and the task is allowed to go forth and execute the protected code. If access is denied because the semaphore count is already at the maximum permitted value, the hardware timer interrupt is still re-enabled, but the task is forced to yield or execute other nonprotected code.

The disabling of the hardware timer interrupt is critical and ensures that the semaphore's count value is tested and changed, if at all, by one and only one executing task at a time because while the interrupt is disabled it cannot trigger and thus force another task to switch in. FreeRTOS provides a few more features, however, tasks and semaphores are its fundamentals and provide enough operating system functionality to run Owl's software demands.

Finch Platform

The Finch platform's primary goal is to provide the lowest cost

Tasks Running - No Operating System (Finch)



In the Finch, there is no operating system. This saves memory space and keeps the cost down. The processor executes each task, completes that task and moves to the next task.

web server with acceptable software and hardware features. The Finch software is expected to provide a single-threaded web server, handle SNMP requests, and run a basic FTP server that accepts software component updates.

Naturally, CPU capability on the Finch is far less than that on the Owl; lower operating speeds, smaller instruction set

architecture, fewer hardware features, and smaller memory sizes forces the Finch to perform fewer software duties at a significant cost savings.

Given that, there are really only two operating system choices: no operating system or a reduced basic operating system. In general, an operating system may be preferred, but it was decided to run Finch without one. At the time the Finch was originally developed, the main constraint was time to market, and furthermore, all of the main software components were already provided by the Finch CPU's manufacturer, Microchip.

These software modules were designed to run in an environment with no OS, and in the end it took less time to modify them to suit our design requirements than it would have to re-design them to run correctly on an OS.

Pedro DeKeratry - software developer

Sensor Interface - Device Drivers

The Owl obtains information about the real-world conditions being monitored from a variety of sensor devices. A device may have one or more sensors. The product into which an Owl is integrated may have one or more devices.

The product will always have at least one sensor device, known as the "internal" device, which is always available because it is permanently attached to the Owl. Again, there may be multiple sensors on the internal device. The product may have zero or more "external" devices of various types which can be added or removed during normal operation. The Owl software dynamically detects when external devices are attached and removed.

External devices all have unique serial numbers which enables the Owl to distinguish between multiple external devices of the

same type. If an external device be removed or replaced, the Owl recognizes it as having been previously attached or updates the system with the replacement serial number.

The Owl stores information about all of the attached devices, including their unique serial numbers, in non-volatile memory; so the Owl recognizes previously attached devices when the system starts up after being powered off. This is important because system alarms (for example, a high temperature alarm) are linked to individual sensors by the sensor's device serial number.

Select the Devices

Naturally, the Owl software is able to communicate only with device types it recognizes. All necessary information about each known device type is kept in system tables in the Owl software. This information consists of specifics about the measurements a given device's sensors can take and how to communicate with the sensors to obtain these measurements.

The Owl system software polls all devices on a regular interval, typically every few seconds. At the beginning of each polling cycle, a complete scan is made to find devices that have been added (plugged in) since the previous cycle. All currently attached devices are contacted using the methods specified in the system device type tables described above.

Measurements obtained during the device scan are stored so that they can be checked for any alarm conditions and so measurement values are available for display on web pages, SNMP, logging, etc.

If communication with an attached device cannot be successfully completed (it has either been unplugged or is unavailable for any reason), the device is marked as being unavailable and appropriate actions are taken by the Owl system software, such as reflecting this on various web pages and generating alert e-mails.

The method for communicating with a sensor is governed either by the communication protocol that the sensor recognizes or by the device's PIC microprocessor, if that type of device has one.

On devices that have a PIC, the communication protocol may be as simple as sending a single command to the PIC and the PIC responding with the sensor measurement. The protocol depends on the program running on the PIC.

How Serial Data Is Usually Received:

- comma delimited data
- string data, non delimited

Typical External Data Sources (sensors):

- remote temperature
- airflow/humidity/temperature
- door position
- water
- current monitor
- millivolt
- dewpoint
- digital

On devices that do not have an integrated PIC, the protocol will be determined by the manufacturer of the sensor. For any protocol that is more complex than sending a single command, the Owl side of the protocol is implemented in a software routine and is linked into the Owl system via a pointer to the routine in the device tables described above.

Typically the routine is a function written in C, but it could be in any language with a compatible Application Binary Interface. This mechanism allows the integration of devices with arbitrarily complex protocols. For example, some protocols incorporate computed check sums to validate transmitted data buffers.

Charlie Mayne – Software Developer

Email

The Owl architecture is able to use its networking facilities to send email messages. In particular, two widely used email protocols are supported: Simple Mail Transfer Protocol (SMTP) and Post Office Protocol version 3 (POP3).

A very natural application of sending email would be to alert a user when something of note has occurred. In the WeatherGoose, for instance, an email message is automatically sent to the user whenever one of the environment sensors reaches a user-defined value (e.g. when the temperature exceeds 70°F).

An email message sent on a regular interval might also be used to give the operator a status update on the device. In particular, receipt of the message confirms that the device is both alive and talking to the network.

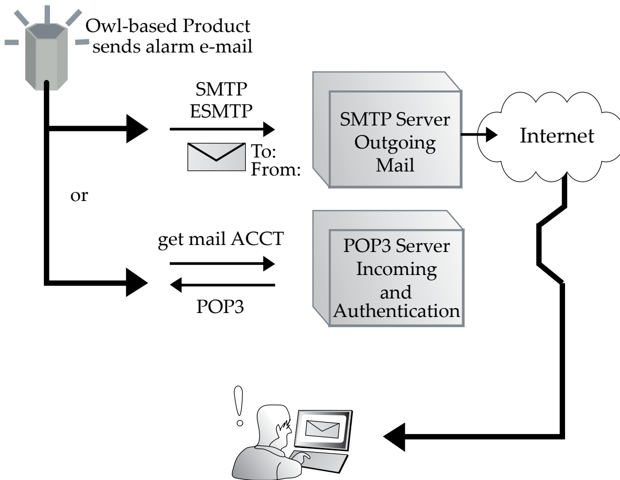
Although the Owl is able to send email, it will not act as the mail server; the user must provide his own SMTP and/or POP3 server and configure the Owl device to interact specifically with it. Most SMTP servers are implemented using the Sendmail mail transfer agent (http://www.sendmail.org/) but Postfix (http://www.postfix.org/), Exim (http://www.exim.org/), and qmail (http://cr.yp.to/qmail.html) are all also very common. POP3 servers will typically be implemented using either Qpopper (http://qpopper.sourceforge.net/) or popa3d (http://www.openwall.com/popa3d/).

SMTP (Simple Mail Transport Protocol)

SMTP is the protocol the Owl uses to create and send each email message. When a message is to be sent, the Owl connects to the user's specified SMTP server (usually on TCP port 25) and executes the appropriate commands.

In particular, each email message must contain an email address that it's coming from, an email address that it's going to, and the content of the message, all of which are specified by the user. Behind the scenes, the SMTP server decides which path the

E-Mail Goes to POP Server or E-Mail Client



Three ways that mail is sent:

1. Behind firewall - SMTP	2. POP3 - authenticates SMTP message	3. ESMTP - Extended version includes authentication

The Owl e-mail alarm and status reports are sent directly to a SMTP server or authorized first through a POP3 server. Multiple e-mails can be sent through an escalation method as conditions continue to decay (or improve).

message must take on the network in order to get to where it needs to go.

If the message is destined for the local server, it will simply be deposited in the user's mail box and the work is done; if it's destined for a remote server, then it will be routed appropriately and handled by the remote machine's SMTP server.

There is no user access verification built into SMTP – that is, a user need not authenticate with the SMTP server before being

allowed to send messages. Access control is usually handled by restricting incoming access to the server by IP address or by only allowing emails to be sent which are specified to come from a particular domain.

If a more traditional method of authentication is desired – one which allows a user to send mail if he is able to provide a valid username/password pair – it is possible for the user to configure a POP3 sister server to act in conjunction with the SMTP server (this is discussed further in the POP3 section). Extended SMTP (ESMTP), a separate protocol, is a direct offshoot of SMTP, which does allow for authentication, but the authentication mechanism is not currently supported by the Owl architecture.

It should be noted that emails are sent on an honor system, of sorts, in that the user is not required to even specify a valid sender email address. This means that it is very easy to impersonate somebody else with critical inspection of who was the actual sender of the email. As a result, it is very important that an SMTP server relay email only for trusted users.

POP3 (Post Office Protocol 3)

Although POP3 is most often used as a means of retrieving email messages from a remote server and storing them on a client machine, the Owl uses it to help overcome the absence of user authentication in SMTP.

Advantages to Using Email for Notification

E-mail is an inexpensive medium to use for notifications, both in terms of network traffic and computer processing power. E-mails do not require much to generate and send and, they're always sent in an on-demand fashion so they don't consume disk space while latent.

Mail servers are typically very easy to configure and, most Linux or UNIX distributions come with them ready to go out of the box. E-mail is a very well-understood and accepted medium for

transferring information and most computer users these days are familiar with it.

Although POP3 is most often used as a means of retrieving e-mail messages from a remote server and storing them on a client machine, the Owl uses it to help overcome the absence of user authentication in SMTP. This is accomplished using a program called smtp-poplock (http://www.davideous.com/smtp-poplock/) and will typically be chained with the qmail mail transfer agent.

Whenever a user wants to relay a message through the SMTP server, smtp-poplock will check to see if the user's IP address is located in a recent authentication database. If so, the email will be allowed to be sent; if not, the user must authenticate through the POP3 server.

Upon successful authentication, the user's IP address will be stored in the database for a defined period of time and, thus, be allowed to send email freely.

Disadvantages to Using Email for Notification

Because email messages do not require credentials verifying the sender's address is valid and accurate, a user can be easily impersonated by simply claiming the message is from a person when it is not.

Much like the postal service, an email message is neither guaranteed to arrive nor will the sender be given a receipt proving its arrival. Most SMTP servers will bounce back a "message undeliverable" email to the sender if the emails cannot be delivered. If the email contains critical or timely information, it is best to send the mail to several receipients through separate SMTP servers.

Of course, this is not helpful if the SMTP server itself is not up and running.

Patrick Nance - Software Developer

Finch: Lowest Cost Web Server

$15 US in Components

In-Line Code Structure

Small Footprint

Microchip^{tm} Processor

Finch - Lowest Cost Web Server

When low cost and a small space footprint are the primary factors, the Finch web server meets the need. This server is based on Microchip, Inc. integrated circuits. Microchip has a history of manufacturing reliable components with low prices.

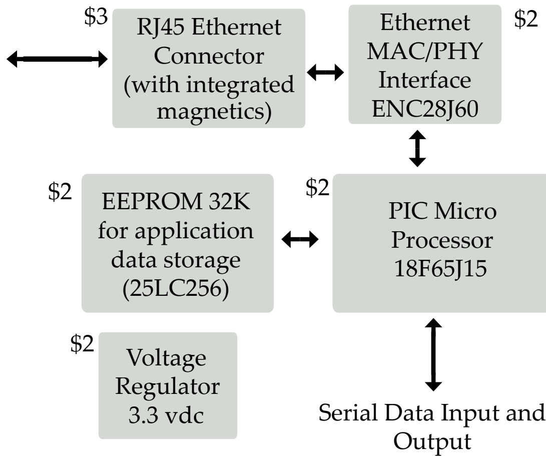
The largest component of the Finch web server is the Ethernet connector. The printed circuit board is about one inch square. This design was built to replace a serial data connection. Other configurations are easily produced. The components (not assembled) are about $15 US and are readily available.

A customer came to us with a request: he wanted to add Ethernet, and a web browser interface to his entire product line of power-monitoring products. However, the interface had to meet the following conditions of size, cost, development time and source code availability.

Small Size: It had to be small enough to fit into the same space as the RS232 serial-interface board it would be replacing, so the customer wouldn't have to redesign or retool all of the equipment, plastic (molded) cases, and mounting brackets.

Very Low Cost: It had to be inexpensive to manufacture, the target was $30 or less, to keep the customer's products inside their

Finch: Lowest Cost Web Server Components and Costs



Four integrated circuits plus a dozen or so discrete components form the hardware base of the Finch. The popular 18F65J15 microprocessor is the engine. Total component cost is about $15 US and this cost becomes less yearly as Microchip releases new versions of this circuit. Serial communications is the common way of passing data to the web server.

desired retail-price targets.

Fast Development: The solution had to be available off-the-shelf, or at least something that could be made from easily-available components, and preferably one that would continue to be available for several years into the future keeping the customer from having to change his design every few months.

Source Code Available: The customer knew how lack of source code could defeat the project, especially when bugs or new requirements appeared. We agreed the source code must be available.

This customer hadn't been able to find anything on the market that could meet all of these requirements. Cost and size seemed to be the biggest obstacles; most of the embedded web server devices on the market were either too expensive for his needs, or too large to fit into his existing equipment housings. A custom design seemed to be the only practical solution -- but was it possible to build an embedded web server from scratch without spending an inordinate amount of time and money on the project?

Fortunately, all of the building blocks were already available and all that was needed was to put them together in the right combination.

Finch Components: Easily Available

The key to the Finch architecture's small size and low cost is the combination of two inexpensive, yet powerful components from Microchip Technology Inc.:

1. The PIC18F65J15 Microprocessor, and the
2. The ENC28J60 Stand-Alone Ethernet Controller.

Together, these two devices form a small yet powerful embedded-processor core with an Ethernet interface, requiring only the addition of the appropriate software and a little external Flash or EEPROM memory. Now we have a fully-functional embedded web server suitable for any number of small-scale, single-function applications.

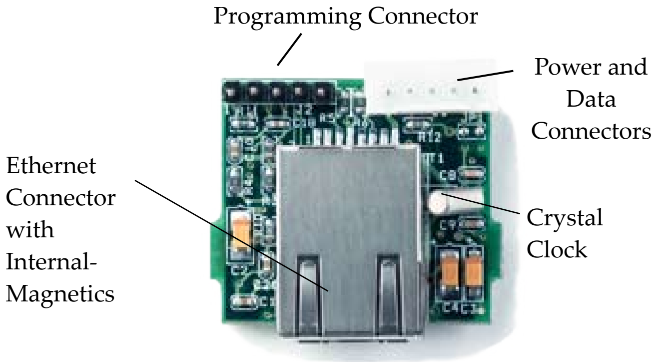
The Microprocessor (PIC18F65J15)

Based on Microchip's high-performance PIC18 processor core, the PIC18F65J15 microprocessor contains 32Kb of Flash program memory and 2Kb of RAM, plus multiple onboard I/O interfaces including two independent SPI ports and two USARTS for asynchronous serial communication.

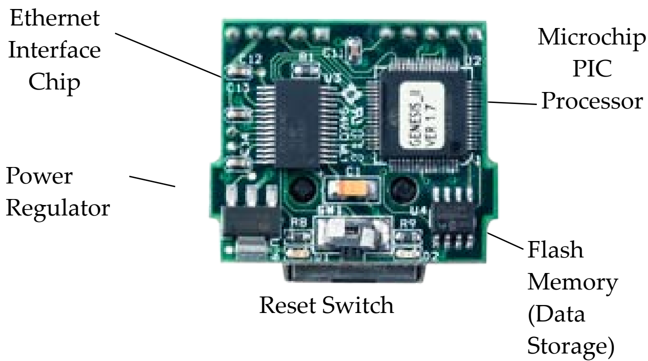
This single chip provides all of the computing power required to support the Ethernet interface, the web server, and the user's application. An internal 4x clock multiplier allows the CPU to run

Typical Design Using Finch Components

Top View



Bottom View



Using only a few integrated circuits, the Finch is based on a popular Microchip microprocessor. When cost must be kept to a minimum and a small physical size is important, the Finch is the ideal choice.

at speeds as high as 40MHz with an inexpensive 10MHz crystal. (The stock Finch architecture uses the 6.25MHz signal provided by the ENC28J60 to run at 25MHz, but this can easily be changed for user applications requiring faster clock speeds.)

Stand-Alone Ethernet Controller (ENC28J60)

Introduced in 2006 as a companion support chip for the PIC microprocessor family, the ENC28J60 from Microchip packs an amazing amount of power into a tiny package. This chip provides both an IEEE 802.3-compliant Medium Access Control (MAC) layer, accessible to the host controller via an industry-standard SPI serial interface, and a built-in Physical (PHY) layer capable of directly driving the appropriate pulse-transformer magnetics of the Ethernet connector without requiring any additional external drive circuitry.

It also provides a 6.25MHz clock output that can be used to drive the PIC microprocessor, helping to further reduce the system cost by eliminating the need for a second crystal to run the PIC.

EEPROM (25LC256)

The EEPROM communicates with the microprocessor via the SPI-bus serial port. This inexpensive 8-pin chip provides an additional 32Kb of data storage, giving the Finch plenty of space for storing user settings and web page data including simple graphics and page layouts.

While this information could be stored inside the PIC18F65J15's internal Flash space, an external EEPROM is generally preferable because the Flash is optimized for program storage rather than data storage. (This is a result of the Harvard memory architecture used by the PIC microprocessor.) Also, the internal Flash has a relatively low write endurance, compared to an external EEPROM, making it unsuitable for storing data which is frequently changed or overwritten.

Good I/O Capabilities

Even with all of the above, many of the PIC18F65J15 microprocessor's I/O capabilities remain available for other uses, making the Finch an excellent choice for small-scale, price-sensitive applications (such as environmental monitoring, simple relay controls, or simple sensor-data acquisition) that don't require highly complex calculations, multi-user access, or massive amounts of data processing, and thus don't need the added complexity and expense of a more powerful CPU or high-level operating system. With the appropriate code, any combination of the following is potentially available for your application:

- Up to 37 general-purpose digital I/O pins which can be connected to switches, relays, or other digital on/off devices
- Up two 12 analog inputs with 10-bit A/D resolution
- A second SSP port allows the user to add SPI or I2C-based devices to the system without needing to worry about servicing the Ethernet MAC/PHY or EEPROM, which are connected to an independent SPI bus.
- Two independent USART modules allow for asynchronous serial communication with existing data-acquisition systems, desktop PCs, terminals, etc.

Low Costs That Keep Getting Lower

Since only a single CPU is required to provide the Ethernet interface, the web server, and the user's acquisition-and-control application, the Finch makes it possible to construct simple Internet-enabled devices much more economically than other, more complex solutions which try to separate these functions across multiple microprocessor and controller chips.

Only a single 3.3VDC is required to run the entire system,

making the power-supply design much simpler and easier compared to other designs which require multiple I/O and CPU-core voltages.

Since the Finch core consists of only three chips, all of which are available in standard SOIC and QFP sized packages, a Finch-based design can easily be constructed on relatively inexpensive two- and four-layer boards. More complex designs typically use BGA (ball-grid array) parts, which almost always demand the use of more expensive six- or eight-layer boards with extremely tight manufacturing tolerances.

Power-over-Ethernet

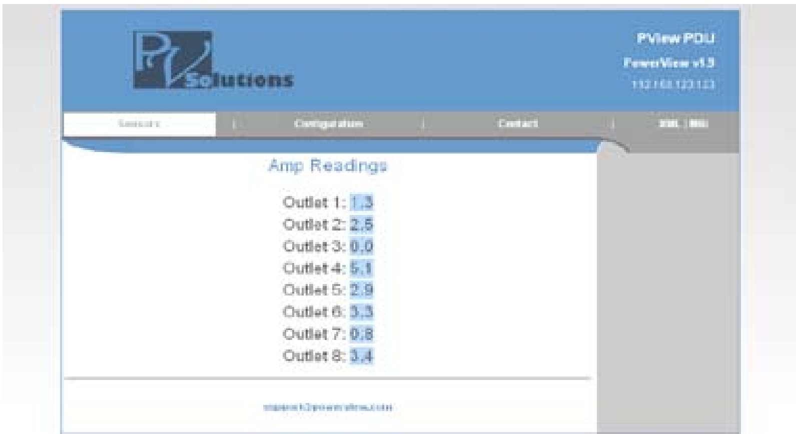
Since the Finch architecture has relatively low power requirements, Finch-based devices are a natural fit for power-over-Ethernet (PoE) applications. This capability can easily be added to the Finch simply by using an Ethernet jack with the appropriate magnetics for separating power and data, and the addition of a few readily available support components to derive the appropriate supply voltages from the PoE-supplied 48VDC line voltage.

If the Finch-based device is to provide connections for other external devices or add-ons, note that it is important to use a DC-DC converter with full isolation between input and output to avoid accidental cross-connection of positive- and negative-ground circuitry.

Easy Add-on to Existing Devices

The Finch web server allows easy web implementation of data from existing electronic devices. The user needs only to supply the Finch with space-delimited serial data. Many existing serial interfaces can supply this data without major modifications. Upon a web page request, the Finch uses the latest data received from the device and converts it to HTML and XML code. (See sample web page on next page.)

The Finch is completely self-contained. All the software



A Finch installed inside power strip produces this web page. the numbers show the amps drawn by each receptacle. The Finch can also measure voltage, power factor, and watts. All these values are computed using the RMS (Root Mean Square) method.

is factory shipped and resides in Flash memory. The web page software can easily be customized to show logos and other brand information.

The printed circuit board can be ordered in custom sizes upon request and properly sized sheet metal or plastic housings are also available on request.

Specifications

Power: 3.3 VDC, 150 ma (separate power supply required)

Connector: RJ-45

Ethernet: 10Mbps, full duplex

Protocols: HTML Web Page

XML: Meta tagged

SNMP: MIB stored in memory

Configuration: IP address, Gateway, Network mask

Dimensions: 1.7″ x 1.8″ x 0.6″

Indicators (LED): Link and Data lights on receptacle.

Data String Format Example

In the example below, the Finch is receiving data from a power meter with an eight channel output. Data is sent every second with each channel sending three integer values in the format shown below. The asterisk is replaced with the number of the channel [1-8]. See the illustration on the opposite page for the resulting web page display.

```
CT*[1-8]<space><space>[0-9][0-9].[0-9]A;
```

Other data formats can be used.

Data Collector (Console) Software

Where multiple Finches are used, the Geist Console, a software program, can collect, graph and log the data from a group of Finches. The resulting consolidated information can be viewed from a single web page. Operation is simple, the user only needs to key in the IP address and logging begins. Console documentation is available by request.

Gary Akins - Engineering Support

Owl: High Performance Web Server

$32 US in Components

The Owl Cube

Operating System

Flexible I/O Structure

ARM 7 Processor

Owl - Lots of Power, Many Features

Like its smaller cousin the Finch, the Owl architecture is designed to allow a single compact module to act both as a general micro-controller and as an Ethernet/web-server interface in a wide variety of embedded-processor applications. The Owl is powerful

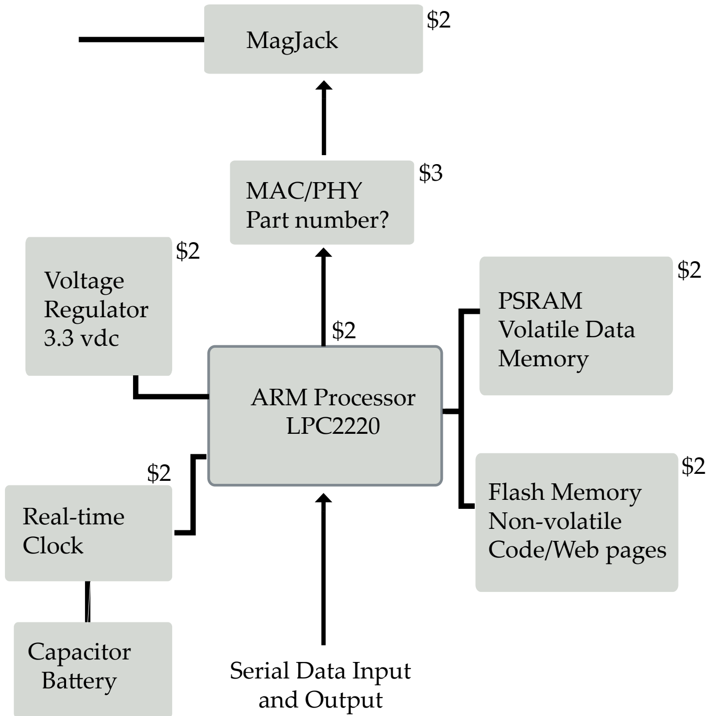
The Owl Cube: a complete, plug in, web server based on the Owl processor. This stacked, two printed circuit board assembly method reduces the area needed for the more expensive eight-layer printed circuit board. The assembly mounts to a printed circuit board with a miniature 18 pin connector.

enough to perform encryption plus other user-friendly functions such as graphing.

The Owl offers a much richer feature set than the Finch, thanks to a powerful 32-bit ARM 7 processor that makes it possible to support more complex internet protocols (including SSL encryption) and to perform more complex control tasks with larger amounts of data.

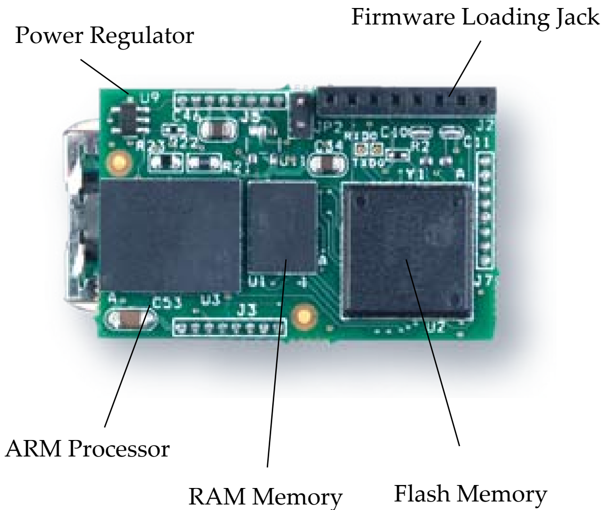
This power makes it a good choice for demanding applications such as multi-station environmental monitoring,

Owl - High Performance Web Server Components and Costs



The Owl's architecture accommdates small and large memory requirements. The ARM processor has a wide variety of I/O options (serial is the only type shown in the drawing).

Owl Plug-In Webserver Top View



This cube-shaped Owl web server has the principal integrated circuits mounted on the top circuit board. This footprint permits the Owl Cube to be added to existing electronic products without taking additional space.

building-wide security systems, point-of-sale terminals, scientific equipment, and industrial control systems.

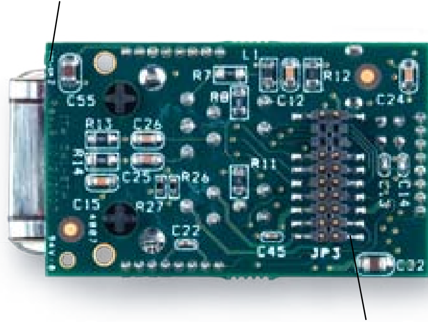
The stock Owl architecture offers the following features:

- ARM 7 microprocessor running at 70MHz
- 64Kb of static RAM (internal to the CPU[1])
- 4Mb of pseudostatic RAM[1]
- 16Mb of Flash memory for application code[1]
- Realtime Clock with independent backup power
- 16C550-compatible USART for serial communications with external devices

[1]*The actual amount of memory available for user applications will vary depending on which OWL O/S and device-support libraries are compiled into the application code.*

Owl Cube Plug-In Web Server Components - Bottom View

Ethernet Receptacle



18 Pin Jack (to other circuits)

The principal component on the bottom of the Owl Cube is the 18 pin connector which permits transfer of power and data to and from the Cube.

- 5 general-purpose, 5V-tolerant I/O pins
- High-speed SPI port[2]
- Integrated Ethernet jack with built-in isolation magnetics and optional Power-over-Ethernet support[3]
- JTAG port for programming and debugging / diagnostics
- Realtime operating system (RTOS) and TCP/IP network stack, plus support libraries
- single-supply (3.3VDC) operation

---

[2]*this requires dedicating some of the GPIO pins to the SPI signals, thereby reducing the number of GPIOs available for other uses. However, if the USART is not required, its pins may be reassigned as GPIO pins.*

[3]*implementing Power-over-Ethernet (PoE) support requires additional support circuitry in the form of a PoE Controller/Regulator. This is not included in the standard Owl design, but can be added to a customer's design, for a modest additional cost, if required.*

Owl: Big Processing Power in a Tiny Package

The core of the OWL architecture is a Philips ARM7 microprocessor, surrounded by a number of inexpensive yet powerful support components packed into a small space. External PSRAM and Flash memory chips allow for flexible memory configurations, while the Microchip ENC28J60 Stand-Alone Ethernet Controller (the same chip used in the Finch) provides an Ethernet interface to the outside world with a minimum of external components. An external realtime clock chip with its own backup-power source, keeps accurate time even during system shutdowns.

The ARM/Philips Microcomputer (ARM7TDMI-S)

The soul of a little machine. This sub-dime sized part costs about $2 US and executes instructions at 70 million times per second. The core is designed by the Advanced RISC Machine Consortium and is used by many integrated circuit manufacturers. This particular chip is made by Philips Semiconductor. If Philips stop making this model, there are other manufacturers to replace it without draconian software rewrites. We avoid proprietary chip designs.

The LPC2220 microcontroller is based on a 16/32-bit ARM7TDMI-S CPU core. Internally, it offers multiple 32-bit timers and external-event counters, an 8-channel 10-bit ADC, PWM channels, external interrupt pins, a Vectored Interrupt Controller (VIC) with configurable priorities and vector addresses, a fast I2C

(400 kbit/sec) interface, multiple power-saving modes, 5V-tolerant I/O pins, and an internal watchdog timer to reboot the CPU in case of a software crash or system lock-up.

For critical code-size applications, an alternative 16-bit Thumb mode can reduce code by more than 30% with minimal performance penalty. (Note that not all of these features may be available simultaneously; some combinations of features will depend on the hardware configuration required for a particular application.)

32Mbit PseudoStatic RAM (PSRAM)

This high-speed CMOS PSRAM combines the best features of static (SRAM) and dynamic (DRAM) memories. By combining a DRAM memory matrix with an internal controller and self-refresh circuitry we avoid the need for an external memory controller. A burst-mode Flash-style interface increases throughput and allows it to coexist on a Flash memory bus. The stock Owl design uses a 32Mbit PSRAM chip, organized as 2Mb x 16, but other configurations are possible depending on the user's application requirements and desired cost.

Flash Memory (128Mbit)

The stock Owl design offers plenty of space for both operating system and user applications with a 128Mbit page-mode Flash memory, organized as 8Mb x 16. For applications which don't require such a large amount of code space, cost-reductions are possible by using a smaller 64Mbit or 32Mbit part. Conversely, for more demanding applications, larger amounts of Flash or PSRAM memory are also possible.

Stand-Alone Ethernet Controller (ENC28J60)

Introduced in 2006 as a companion support chip for the PIC microcontroller family, Microchip has packed an amazing amount of power into a tiny package. This chip provides both an IEEE 802.3-compliant Medium Access Control (MAC) layer, accessible

to the host controller via an industry-standard SPI serial interface, and a built-in Physical (PHY) layer capable of directly driving the appropriate pulse-transformer magnetics without any additional external drive circuitry required.

The controller also provides a 6.25MHz clock output that can be used to drive the PIC microcontroller, helping to further reduce the system cost by eliminating the need for a second crystal to run the PIC.

Realtime Clock (DS1340U)

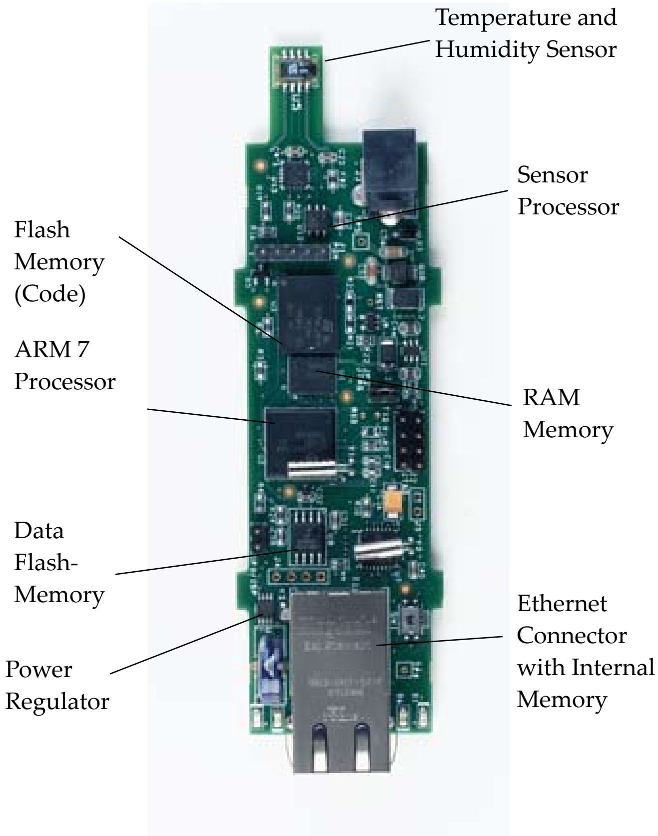
In order to keep power consumption to a minimum, the Owl architecture employs a separate, ultra-low-power, realtime clock chip rather than using the ARM 7 CPU's onboard realtime clock. Able to operate on as little as 100uA, with a typical accuracy of +/- 15ppm, this chip allows the Owl to maintain accurate time even while the rest of the system is inactive. In the stock Owl configuration, backup power is provided by an aerogel "super-capacitor" which can sustain the clock for approximately two weeks. If longer power-down times are required, a larger capacitor can be used; alternatively, a NiCd or lithium battery can also be employed for backup power with minor modifications to the software that controls the onboard charging circuitry.

Extensive I/O Capabilities

The LPC2220 offers a broad array of I/O capabilities, thanks to its sophisticated set of internal peripherals. The inclusion of a true realtime operating system (RTOS) in the Owl architecture makes it relatively easy to put these devices to work while still maintaining an acceptable degree of system responsiveness to the internet-based user interfaces. Among the available I/O features are:

- Up to 45 general-purpose, 5V-tolerant I/O pins which can be connected to switches, relays, or other digital on/off devices
- One 10-bit A/D converter with up to 8 available input channels

Owl Product Example: Climate Monitor (MicroGoose)



Owl components are small - here they are integrated into server room climate monitor called a MicroGoose. The monitor contains an on-board temperature and humidity sensor. Power is supplied by a wall plug transformer. The overall size of this printed circuit board is about 1.5″ x 3.″ All the components are present. The user simply plugs in the wall transformer and an Ethernet cable to begin monitoring the climate.

- Up to two 16C550-compatible USARTs for asynchronous serial communication with existing data-acquisition systems, desktop PCs, "dumb" serial terminals, etc.
- Hardware-driven PWM output with up to six output channels, suitable for motor-speed control, heating-element temperature regulation, and other such applications
- 400 kbit/sec I2C interface
- SPI port (clocked at 8.75MHz) for use with serial EEPROMs or other SPI-based devices
- JTAG port for in-system programming and diagnostics

Note that not all of these features can be available simultaneously because some of them share I/O pins with each other; however, with the appropriate programming and judicious selection of features, many combinations are possible to suit a wide variety of potential applications.

Compact, Inexpensive, and Easy-to-Use

The standard Owl device, including the Ethernet jack and capacitor backup for the RTC (realtime clock), occupies a space about the size of an ice cube.

Variations designed for specific applications can often be made to fit a customer-specified shape and size as required. If the objective is to replace an existing module with an Owl-based substitute, even the pinouts can be made to match an existing specification.

The Owl is built from easily-available commodity parts; the design contains no expensive custom silicon, and long lead-time parts have been avoided wherever possible. The Owl operates from a single 3.3V power supply, with no need for separate "core" and/or "I/O" voltages.

All of the external I/O lines are 5V-tolerant, simplifying the task of interfacing to existing TTL and CMOS-based circuit designs.

Realtime Clock (RTC)

The Owl system uses current time and date (a.k.a., "real time") in several ways:

- System event logging (such as tripped alarm conditions)
- Data point collection
- Graphing logged historical data
- Any network communication
- E-mails are time stamped
- Current local time, day and date are displayed on the user interface web pages

The Owl is equipped with a realtime clock (RTC) that is separate from the central processor so it can be sustained by a backup power source during any power interruptions to the unit. The Owl's backup power source maintains the RTC's date and time registers and keep the oscillator circuit powered for up to two weeks. When system power is restored, the accurate time and date are available immediately without having to be reset. If the power outage is too long for the backup power to sustain the RTC, this condition is detected on system startup and the invalid RTC timestamp is not used until it has been successfully reset - either manually (via a user interface web page) or automatically by the Owl software accessing a Simple Network Time Protocol (SNTP) server, assuming one is available.

Many SNTP servers are accessible on the Internet. For example, the Owl's default primary and secondary SNTP servers are the US Navy's servers:

```
tick.usno.navy.mil (192.5.41.40)
tock.usno.navy.mil (192.5.41.41)
```

Hundreds of well known SNTP servers are maintained by universities, government agencies, commercial companies, non-profit organizations, etc. If an Owl is on a LAN that does not

System Clock, set to GMT

Set Clock method: NTP Server

GMT to local, (+/-)hh:mm -05:00

Month	Day	Year (yy)	Hour (0-23)	Minute (0-59)	Second (0-59)
10	08	07	19	56	59

NTP primary server tick.usno.navy.mil
192.5.41.40

NTP secondary server tock.usno.navy.mil
192.5.41.41

Sync to NTP server period (seconds) 1800

Save Changes

Knowing the current time allows the embedded system's software to add accurate time stamps to functions like logging, graphing and e-mail alerts. The alternative is using an Internet-based time source but this method has drawbacks when the network connection is lost temporarily and the time is not available.

allow access to the Internet, a private SNTP server can be set up to serve the LAN. Software to have a PC serve this function is readily available.

Alternately, the time and date can be set manually by a user through a web page. This web page allows the user to do the following:

- Select between manual mode and automatic mode (i.e., accessing a SNTP server)
- Specify the time and date if manual mode is selected
- Specify primary and secondary SNTP servers to use if automatic mode is selected
- Specify the offset between local time and

Greenwich Mean Time (GMT; a.k.a., Coordinated Universal Time – UTC)

- Set the frequency with which the Owl will synchronize to a SNTP server

When the Owl is in automatic mode, it contacts an SNTP server on system startup and at subsequent time intervals set by the user, to synchronize the RTC with the time and date provided by the SNTP server. This frequency is typically daily or weekly. The actual elapsed time of the SNTP packet round trip transmission on the network is incorporated into the RTC synchronization software to obtain accuracy well within the one second resolution provided by the RTC.

The Owl's RTC chip is a Dallas Semiconductor DS1340 that interfaces to the central processor serially via an I^2C bidirectional bus. The backup power to the clock chip is provided by a super capacitor connected to the chip's secondary power supply pin.

The Owl system software configures the clock chip to control the trickle charging of the super capacitor. The DS1340 has a Trickle-Charger Register allowing the Owl software to enable a trickle charge and select resistors in the DS1340 to limit current level on the Secondary Power Supply pin, which controls the capacitor charge rate. On system power up, the Owl sets the DS1340 to charge the capacitor at a low current, slow rate. This protects the capacitor in the event that it was discharged to a very low level during power outage.

After a short time period, 90 seconds, when the capacitor is sufficiently charged to safely accept a higher current, the Owl software sets the Trickle-Charger Register to charge at a higher rate for ten minutes to rapidly bring the capacitor to a fully charged, or near fully charged, state. Then, the Owl software sets the Trickle-Charger Register to select a low current level to top it off and keep the capacitor in a fully charged state. If the power supply to the DS1340's Vcc pin falls below a given threshold, the DS1340

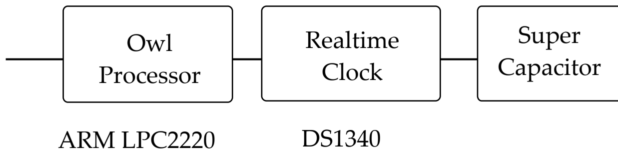
automatically switches from Vcc to the Secondary Power Supply pin connected to the super capacitor to maintain the data registers and keep the oscillator going.

Charlie Mayne - Software Developer

Backup Power: Battery vs. "Super-Capacitor"

Until recently, nearly all devices which needed to keep certain sections, such as memory or realtime clocks, continuously "live" and operational even when main power was removed from the system, have used some type of battery source.

Realtime Clock - Battery Powered



A clock is particularly useful in embedded applications. Logging, e-mails, and scheduled events depend on knowing what time it is. Network time servers can supply a time but this actual time can be lost if the network connection goes down.

In recent years, however, so-called "super-capacitors" have become increasingly viable for such applications; by using a carbon aerogel as their dielectric, super-capacitors (or "SuperCaps") can have plates and dielectric separators with astonishingly high surface areas for their size, allowing super-capacitors to achieve very high capacitances and hold sufficient amounts of charge to make them a viable substitute for batteries in low-current applications.

Which choice is best depends, of course, on the particular application. In the case of the Owl's realtime clock, the current draw of the clock circuitry is extremely low when in standby/power-down mode, so even a relatively small (0.2F) SuperCap can keep the clock running for several weeks. Of course, a lithium-cell battery could potentially run the chip for much longer than that, but a SuperCap is the better choice primarily for two reasons:

1. Easy Rechargeability: Unlike a NiCd cell (a common choice for such applications), SuperCaps can be recharged almost indefinitely, and do not exhibit any of the "memory effect" issues commonly associated with such cells that are repeatedly recharged without first being fully discharged first. This allows SuperCaps to be continuously trickle-charged to maintain a fully-charged state, while the system is running on main power.

2. Long-term Viability: Batteries tend to self-discharge over time, even when no current is being drawn from them, raising the possibility that if a device were to be in continuous operation for months or years, the battery could self-discharge to the point that it would no longer be able to run the clock when a power failure occurs. A lithium "coin cell", for example, can self-discharge to unusability in as little as 18 months.

While neither of these problems are insurmountable, they would have increased the complexity of the design. Since the consequences of eventually losing power to the realtime clock are relatively low (for most of the applications we had in mind, if the users' installations loses power for several weeks, he has far more serious problems to worry about than whether or not the clock was keeping the right time!) The SuperCap is definately the most obvious and best choice.

Gary Akins - Engineering Support

Web Data Transfer Methods

HTTP Data Transfer

Data Handshaking

Static HTML Pages

Dynamic Pages

HTTP Server

Web enabling your product allows users to interact with it from anywhere using an ordinary web browser. To make this a reality requires a web server small enough to embed inside your product. This chapter describes our small embedded web server.

Note that a web server is also referred to as an HTTP server because the information exchanged between client (the browser) and server is defined by the HyperText Transfer Protocol (HTTP). The full specification of HTTP can be found in RFC2616 but this chapter explains the aspects relevant to understanding the server implementation. HTTP is considered to an application layer protocol meaning it relies on other lower level protocols such as TCP, IP and Ethernet that are described elsewhere.

In this chapter we first describe the messages that are exchanged between the user's web browser and the embedded server. We will start with describing the simplest case which is a request for a pre-defined set of data and build towards complex pages containing dynamic data and forms that allow users to enter data. Building on this background, we describe the architecture of the embedded web server and explain how it works. The goal is to enable you to define your own web pages that display data generated by generated by your product and control its operation.

A Simple Request

When you type this

```
http://192.168.123.123/dyn_help.htm
```

into the address bar of your browser and press enter, the browser to establishes a TCP connection to the embedded server at IP address 192.168.123.123 on port 80. Port 80 is the default port where the web server listens for incoming requests.

The web browser and the embedded web server then exchange some information in the form of lines of text. This

1. Browser request and server response for a basic HTML page.

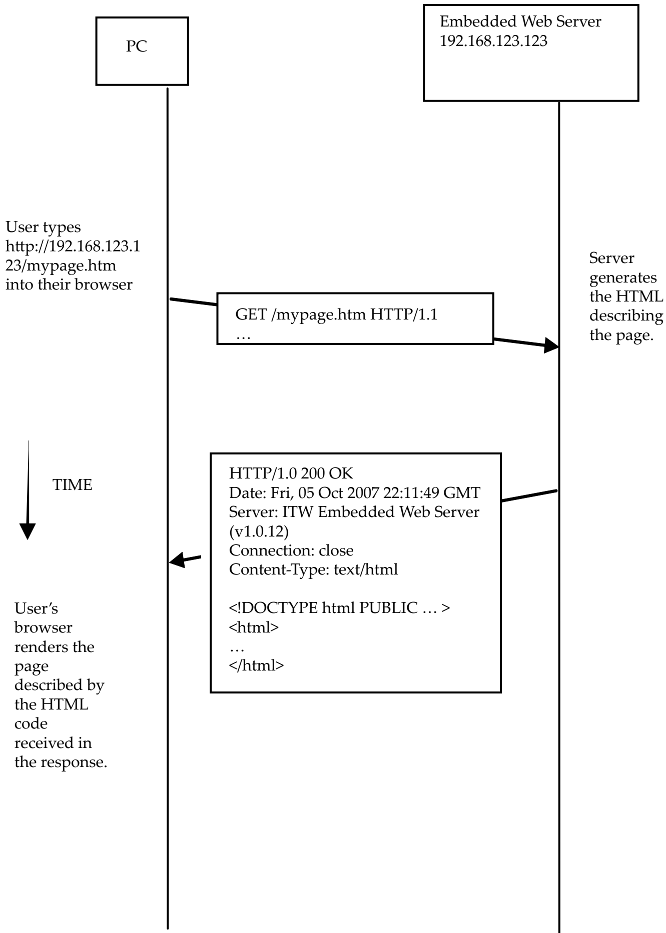


Figure : Browser – Server communication for first retrieval of a static page which is then cached by the browser. Subsequent request asks if page has been modified since the first retrieval, if no newer page exists in the server, the server returns code 304, and the browser uses the cached image.

communication between the web browser and the webserver can be captured and displayed by means of an HTTP debugging proxy such as Microsoft's Fiddler. Fiddler is a very useful tool for revealing how the browser and server interact with each other.

Figure 1 illustrates the interaction between the user's web browser and the embedded server when a basic web page is requested.

For the sake of brevity and clarity, some of the lines of text exchanged between the browser and the embedded server have been removed from Figure 1. These are indicated by the ellipses (…). The content of these omitted lines are not important for our understanding of the basic operation.

Each line of text ends with the carriage return and line feed characters. Lines may be of any length, but our web server rejects requests containing lines exceeding a limit set by the #define MAX_LINE_LENGTH. This prevents buffer overflow errors which are potential security vulnerabilities.

The general format of the information in both directions is similar and can be divided into three parts. The first line sent from the browser to the server is known as the Request Line, in the opposite direction the first line is known as the Response Line. The first line is followed by a series of lines each beginning with an identifier that ends with a colon (:). These lines are known as headers and they are used to convey certain information between browser and server. These will come into play later when we discuss more complicated interactions between browser and server. A blank line marks the end of the headers and, in our simple case, the end of the request. Requests using the POST method will have data following the blank line; this will be discussed in more detail later.

The Request Line has three fields; these define the method, the path and the revision of HTTP. The method field may be "GET", "HEAD" or "POST". The RFC defines some additional methods

but these are not commonly used and are not supported by our webserver.

The GET method is used to request a web page. It can also be used to submit data to the webserver. POST is an alternative method for submitting data to the server. Submitting data to the web server, for example when the user fills out a form, will be discussed in more detail later. HEAD is used when it is desired to retrieve only the header portion of the response.

The path field of the request line is of key importance in locating the page. The path is the portion of the URL ("the web address") from the first slash ('/') to the end. The path is used by the web server to identify which page is being requested and this determines how it goes about generating the content that will be returned. We will describe the more complicated ways the content can be generated later. For now we consider the simplest case is where the path corresponds to a file name in the flash file system. The content of the file will be the returned as the body of the response.

The first line of the response contains three fields: the name/version of the protocol being used; a numeric return code, and a human readable status string.

The code 200, "OK", means the request was processed successfully, therefore you don't normally see this in your web browser. On occasion you probably have seen code 404, "Not Found". Other return codes are used to cause the browser to take certain actions, and are explained in subsequent sections.

The response contains header lines between the first line of the response and a blank line.

In our simple scenario the Content-Type header provides information to the browser that determines how the data content should be rendered. The value text/html will result in the content being interpreted as HTML code and any HTML tags that, for example, define the appearance or layout of the text will take

effect. The value text/plain will result in the content being rendered simply as text; any HTML tags will not have their intended effect. GIF images may be identified by the value image/gif.

In later scenarios, the purpose of other headers sent by the server will be explained.

After the blank line comes the body of the response. In Figure 1 this is shown as an HTML document, but it could be plain text or the data defining an image.

Speeding Things Up

As was explained in the previous section, a request might reference a static page which is a page whose content does not change from one request to the next. For example, pages that are just files stored in the flash file system don't change unless we upload a new flash file system image. Static pages might be such things as a Cascading Style Sheet or images, such as a corporate logo or a background that are referenced by multiple HTML pages.

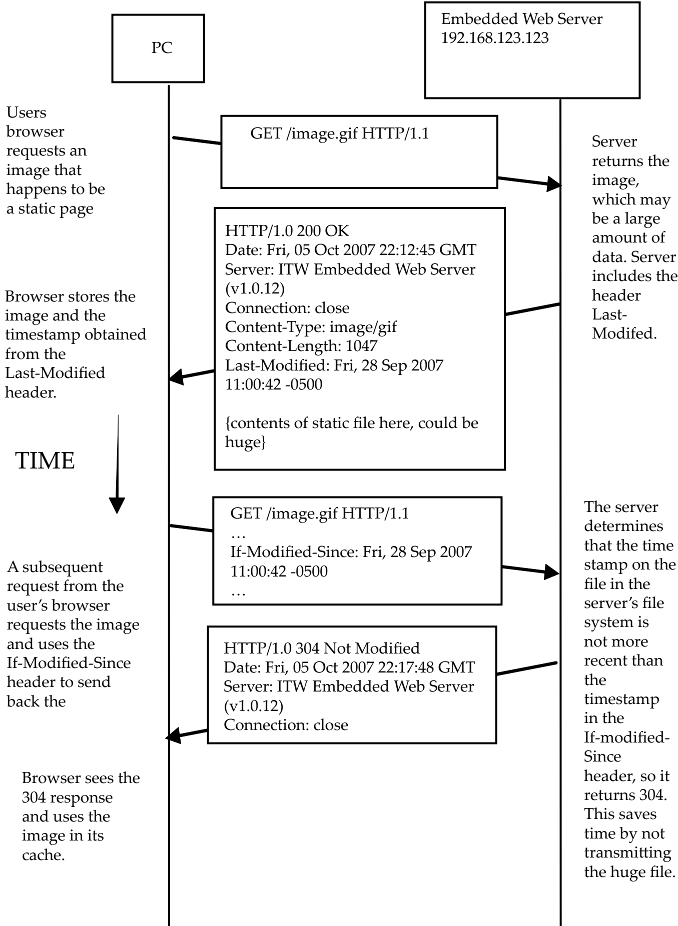
For static pages the server's response includes the Content-Length header which provides the browser with the number of bytes returned in the body of the response. This is easily provided for static pages because it is the size of the corresponding file in the flash file system.

We can gain a significant performance improvement by caching static pages in the browser. Figure 2 illustrates how this is done.

At the top of Figure 2, the web browser is requesting the page for the first time and the server responds by transmitting the entire page. This is essentially the same we discussed previously, except that because the server knows this is a static page, it includes the header Last-Modified. This header contains a timestamp that the browser will store along with the page in its cache.

The next time the browser needs the page, it checks its cache for a copy and uses the header If-Modified-Since to send the stored

2. Browser request and server response for a static page.



Browser request and server response for a static page which is then cached by the browser. Subsequent requests ask if the browser has a version of the page that is newer than that stored in the browser cache.

timestamp back to the server. When the server gets the request, it compares the time stamp it received to the timestamp on the file it currently has. If the timestamp on the file is not more recent than that it received from the browser, it knows that the browser is already in possession of an up-to-date copy of the page. It then sends back the response 304, "Not Modified".

This response is very short just consisting of the status line and header lines with no body. This is much more efficient than resending the entire file.

Configuring Your Device

Almost every web-enabled device has some user configurable settings; these can be presented to your user as a web form displaying the current settings and allowing the user to submit new settings by clicking a button in the form. There are two possible methods that can be used to deliver the information to the server. You decide which method to use when you write the HTML code that defines the form.

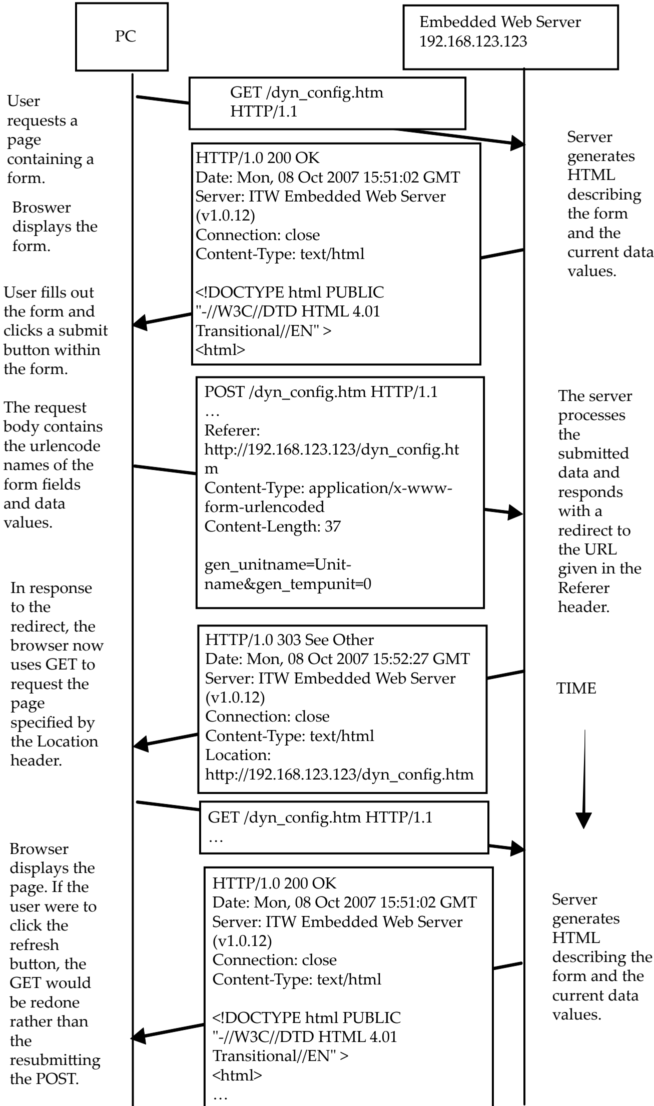
GET is one method that can be used. When the user clicks the submit button, a request is sent to the web server just as in Figure 1. The URL used is the form page URL with a question mark and the encoded form data appended to it. The appended data looks like this:

```
name1=value1&name2=value2& …..  &nameN=valueN
```

The names correspond to names assigned in the HTML code to particular controls on the page. The values are what the user entered. The names and the values are "URL encoded" by the browser which converts certain reserved characters to a form that can be transmitted without ambiguity. For example a '+' is converted a '%' followed by the two digit hexadecimal ASCII value of the character. Any spaces are converted to '+'.

The result of submitting data using GET is that the first line of the request will contain the submitted names and values as part of the path field.

3. Browser request and server response to POSTing data



Browser request and server response to POSTing data to a web based form showing POST-redirect-GET.

POST is the other method that can be used to submit form data. POST submits the name-values pairs in the body of the request (following the blank line terminating the headers). This is preferred for large amounts of data as the long string of name-value pairs does not appear in the browser address bar as it does if GET is used.

There is an issue that arises when POST is used. The issue is that if the user clicks their browser refresh button after submitting the form, they are unwittingly sending the POST request a second time. Many browsers warn the user by popping up a message box asking if they are sure they want to do this. This is annoying. The way around this is to use a strategy known as POST-redirect-GET that is shown in Figure 3. This involves some new headers and response codes.

The top of the diagram shows the user requesting and receiving the web form. This occurs as we described earlier.

Next, after filling the form data, the user clicks the submit button on the form. This results in the browser sending a POST request to the server. The data that was entered is sent in the request body. Note that the browser includes the header Referer (yes, this is the spelling that is specified by the RFC and is necessary for it to work) which provides the URL of the form.

The server responds with a code 303, “See Other” and it includes the header Location which contains the URL that was provided by the Referer line in POST request.

When the browser sees the code 303, it uses GET to load the page at the URL provided. Since this is the URL of the form, what the user sees is the same form. The result is that if they click the refresh button, they are reloading the form via the last GET rather than resubmitting the POST. The annoying warning pop-up will not appear.

Another thing to keep in mind about forms is that they are an example of what we call a dynamic page. A dynamic page

contains variables whose values are substituted in at the time the server sends the HTML code to the browser. The relevance of this for forms is that this mechanism allows us to display the current values of the settings in the form. After the user submits changes, the POST-redirect-GET scheme puts a fresh copy of the form with the current settings inserted in it. This provides some feedback to the user that the submitted changes were accepted. If they were not accepted the fresh copy of the form can contain error messages pointing out which fields need correction.

More About the Response Body

For a static page the response body consists of the contents of the flash file system file corresponding to the URL.

Dynamic web pages contain content, for example a temperature reading, that may change each time the user loads the page into their browser. Such pages are not cached by the browser so that the information presented is always up to date.

For a dynamic page the webserver creates the HTML code in the response body by executing a write_content function corresponding to the URL. The write_content function is C code that extends and customizes the webserver for your application. Writing this C code directly is tedious and error prone, so we have created a system that allows developing each dynamic web page as a file containing HTML code. These files are then all processed by our pagecompiler.pl script, which creates the required C source code for the write_content functions for each page.

Our pagecompiler also supports special tags we defined which cause substitution of data values to take place when function write_content function is executed. This enables pages to display values that are generated dynamically inside your product.

Webserver Architecture

Now that we have explained the information exchanged between the user's web browser and the web server embedded in

your product, the necessary background is in place to delve into how the webserver is structured and how it works.

At the top level, the code module http_test.c contains a continuously running task that accepts incoming TCP socket connections and passes the socket to the function that implements the generic processing of the webserver. That function is called ws_process_request and is located in webserver.c.

Note that http_test could be structured to implement a multi-threaded web server that handles multiple requests simultaneously. To do so, http_test would have to create a new OS task for each connection accepted and the socket connection would be handled off to that task. The new task would be responsible for calling ws_process_request and passing it the socket. The code in ws_process_request and all related functions are implemented in a thread safe manner.

The function ws_process_request reads the various lines of the HTTP request that were described in a previous section and as it does so fills in a data structure called request. The request data structure keeps track of the method, the path portion of the URL, the names and values of any data the user entered in the fields of a form, among other things. This data can be efficiently communicated to other functions by passing the address of the request structure (a pointer) to the other functions.

The request data structure and other buffers used to process the request are automatic variables allocated on the stack when function ws_process_request is called. This provides for a thread safe design. Any necessary processing of the data in these variables is done in place to reduce memory usage and enhance performance by eliminating unnecessary copying of data. For example, data submitted in a form is enters the webserver in encoded form where all the name value pairs are concatenated into a long string in the buffer. The decoding process null terminates each individual string in the buffer and sets up pointer variables to the individual strings.

The actual mechanics generating the response to a request for a particular webpage is handled by C modules we call page handlers. Page handlers consist of a standard set of functions and a data structure consisting of pointers to those functions. The webserver can support multiple page handlers, each responsible for generating the response to request for a particular type of page.

The webserver identifies the appropriate page handler for the requested page by stepping through its set of page handlers and calling the function is_page_handler within each handler. The function is_page_handler examines the path portion of the URL and returns true if the handler is the one assigned to respond to that request. This process continues until the webserver receives an affirmative (true) response or there are no more handlers to try. If after searching the entire set of handlers none of them have identified themselves as capable of handling the request, a code 404, "Not Found" is returned to the browser.

Another function that the handler must provide is write_mime_headers, whose purpose to to return the part of the response header that is particular to the type of request the handler is designed to handle. For example, requests for static pages generate responses that include the headers Content-Type, Content-Length and Last-Modified. Since the path for a static page corresponds to a file in the flash file system of the server, the header information is readily obtained from the file system and included in the response.

For dynamic pages the portion of the response header generated by write_mime_headers is typically just Content-Type: text/html.

Various other functions provided by a page handler are used by the webserver to determine what kind of processing needs to be done to generate the response. Among these additional functions are is_static which returns 1 if the page is a static page, i.e. one that can be cached and reused by the browser. The function redirected returns 1 if the page contains a form and the request used the POST

method. In such a case the POST-redirect-GET response sequence is used. See Figure 3. The function parse_postdata returns 1 if the request method was a POST and the page is one that contained a form, in such a case the webserver must decode and process the form response data contained in the body of the request.

The key function contained in all page handler modules is write_content. The write_content function in a handler for static pages simply opens the file in the flash file system identified by the path portion of the URL, and writes the data in the file to the TCP socket.

In the case of a dynamic page, write_content function contains statements that write out the HTML code that defines the web page that the user will see. Typically, the HTML code is broken into pieces output by different tcp_sock_write statements. The write_content function may contain code that algorithmically generates the HTML code. In the next section, we will explain how to create the write content functions for your own web pages.

Defining a New Web Page

Creating the write_content function directly in C language is tedious and error prone because the relationships between the pieces of HTML code tend to get lost in the structure of the C-code. This is problematic on a number of levels but it presents particular difficultly with ensuring that the various HTML tags are all matched with their proper closing tags.

To avoid these difficulties, we have created a system that allows you to define each web page as a separate file containing HTML code, possibly with some additional markup tags we will describe shortly. The names of these files should end with a .wc extension. The HTML code can reference other static pages such as a style sheet or an image by using standard HTML link or img tags that include the file name where the static page is found in the flash file system. Details on how to use HTML and CSS to define a web page are provided in another chapter.

By running the following command:

```
perl pagecompiler.pl *.wc
```

a C-source file will be automatically generated from each .wc file. Note that you must process all .wc files at once because some may have dependencies on others. Each C-source file contains one function that will send the HTML stream for a web page when called by the write_content function of the webserver. The path portion of the request identifies the specific page the user requested and is used in write_content to determine which of the functions to call.

In addition to the .c files created for each web page, the pagecompiler also creates the a file dyn_pages.h that declares all the generated functions defined in all the .c files, and a file dyn_pages.c that includes all the .c files. This makes it possible for the makefile used to build the system to only name dyn_pages.c and for the parts of the code that call the generated functions to just include dyn_pages.h. This eliminates the need to change the makefile when adding or removing web pages.

Dynamic Content

But standard HTML is not enough to support creating dynamic webpages, pages that have content that is not known until the time the user accesses the webpage. Examples of such content would be the current value of a sensor reading or the current setting of a user configurable option. Such values must be obtained from other parts of the product's software, by calls to functions outside the webserver code.

To support dynamic content, we have created a set of tags that extend HTML and define a standard programming interface to the rest of code. For example, if you want the title of the page to be obtained from elsewhere in your code, you use the <?= … ?> tag with a C expression inside

```
<title><?= getPageTitle() ?></title>
```

The pagecompiler translates into, among other things, a function call to a function that must have a declaration like this

```
int getPageTitle(char *buf, size_t buflen, [,…])
```

The function must return less than zero on error, zero or greater on success. The buf and buflen arguments are mandatory. The notion [,…] indicates that such a function may or may not have additional arguments. Such functions constitute part of the interface to the rest of the software making up your product. The declaration of these functions should be included in the file dyn_globals.h to ensure that dynamically generated functions have access to them.

Injecting arbitrary text strings into the outgoing HTML code introduces an issue of safety. Say the string includes the character ‘<’, if this is not properly encoded, the browser will interpret this as the beginning of an HTML tag resulting in the browser rendering the page in some unintended way that may depend on what character(s) follow the ‘<’.

To avoid this problem, the <?= … ?> tag automatically encodes the emitted string, preventing you from forgetting to do this and accidentally introducing this vulnerability into your system. In the event that you really do need to output the string without encoding, we provide the <?=raw ?> tag.

Another tag, <?msg KEYNAME ?>, provides support for obtaining strings from a centralized message catalog. The pagecompiler substitutes a function call for these tags that will, at run time, retrieve and insert into the HTML stream the string associated with the value of KEYNAME. This makes it possible to store all the strings your application displays to the users in a file in the flash file system. This makes it very easy to create a version of your product to be sold under a different brand or into a market

where a different language is spoken just by changing out the message catalog file. The message catalog file format is described in the chapter on the message catalog.

Finally, another tag we provide is <?include OTHERPAGE ?> where other page is the name of another .wc file containing some HTML. OTHERPAGE should not include the .wc suffix. The file could, for example, define a standard header or footer, that can appear on multiple pages just by using the <?include OTHERPAGE ?> tag in each of the other pages.

David Karoly - Electrical Engineer and Software Developer

Web GUI Techniques

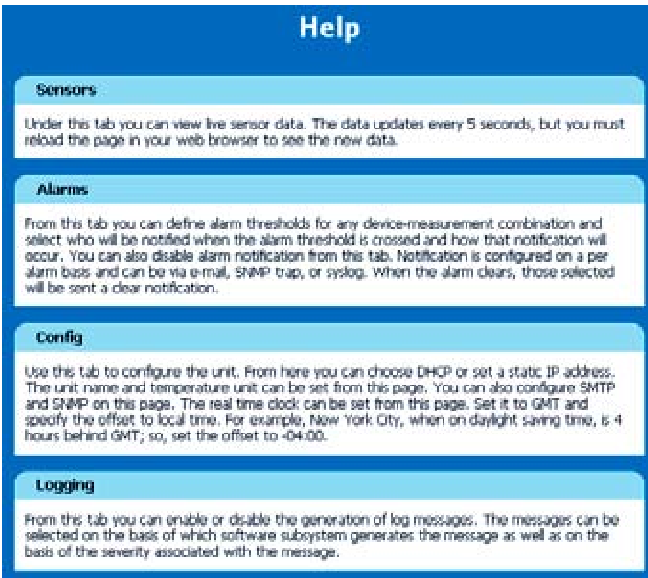
Lightweight, Fast Web Pages

Using Cascade Style Sheets

Examples of Styles

Web Page GUI Design

The GUI (Graphical User Interface) provides the tools by which the user interacts whith your device. You can have a great product but a poorly designed interface can ruin it. The interface shouldn't be an obstacle someone has to overcome to use your device. Hopefully, in the following chapters we can provide some hints that will help you make a better GUI.



Shows consistent subject headings and enclosures.

Consistency

An important part of designing an effective GUI is to ensure consistency. What does this really mean? It means choose an approach and stick with it. For instance if you have a navigation bar, keep it in the same place on every page. If there is a submit button make sure it's it looks the same on every page. The user doesn't expect these sorts of items to change. The expectation is they will be the same on every screen. In other words interfaces are expected to be consistent.

Responsiveness

A reasonable user knows not to expect an embedded device to respond as quickly as a workstation, but they don't expect to have to wait too long either. If an action will take some time keep the user informed with a countdown timer, but don't leave the user

Group A			0.0 Arms	
Outlet	Name	Status	Amps	URL
☐ 1	Server R2809	On	0.0 Arms	
☐ 2	Server C2724	On	0.0 Arms	
☐ 3	Server C0348	On (Off in 2 sec)	0.0 Arms	
☐ 4	Server X289	On	0.0 Arms	
☐ 5	Server X304	On (Off in 4 sec)	0.0 Arms	
☐ 6	Server Mail	On	0.0 Arms	
☐ 7	Server Demo	On	0.0 Arms	
☐ 8	Switch	On (Off in 7 sec)	0.0 Arms	

Action: None [Execute]

Shows an example of response feedback.

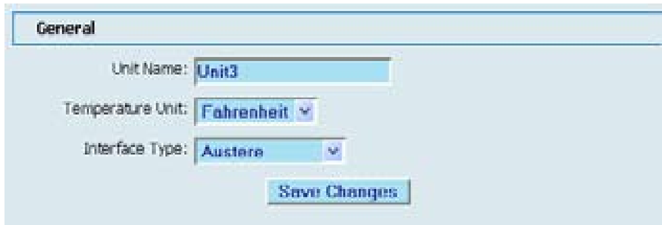
guessing. Better yet restructure some code or optimize it to make the device more responsive.

For example, we had a device that could turn on a light. When the user clicked on the control it took several seconds to turn the light on. The user expected it to come on quickly. They were left wondering what the unit was doing.

Did the unit die? Did it just ignore my request? This wasn't acceptable. After looking through the code we found that we were only checking for an on/off change every 10-15 seconds. We had the system check more frequently and were able to get the delay, between button click and the light change, down to just a few seconds. The lesson here is a product that is slow to respond seems inferior, even if it's a good product.

Familiarity

Consider the sort of interfaces the user is familiar with. Is there a piece of software they use on a daily basis? What sort of



While the styling is different, standard input elements are used.

interface does it have? Study it to get some ideas on how to create a better interface for that type of user.

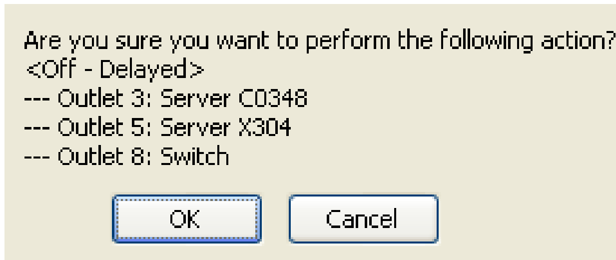
Our devices are web-enabled. We assume that the customer is comfortable navigating web pages, so our interface looks like a normal web page. It has a header, footer, body and navigation bar (nav-bar) just like a typical web page. The user knows how to use this interface without instruction, because he is already familiar with it.

This way the user doesn't have to fight through a boring manual or have to call technical support. The user has expectations about interfaces based on what they are familiar with. If you know these expectations the user will interact with your device smoothly and will be happy he doesn't have to take the time to learn a new interface.

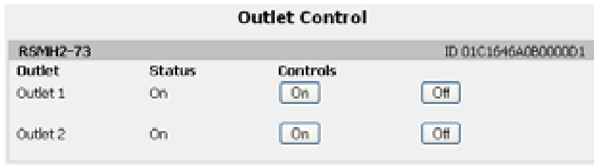
Feedback

Interaction between the user and the device is a two-way street. If the user performs an action he expects some sort of feedback, especially in the case of an error. A good error message informs the user what can be done to remedy the problem. This means don't just give a message like "Error 134." Such a message is useless, and forces the user to look up the error code or call technical support. Even when not dealing with errors it's best to keep the user informed about what the device is doing. For

instance if the user selects an option to turn on a relay in 30 seconds, provide a countdown. This way the user isn't left guessing what the device is doing, just sitting there for 30 seconds. The goal of feedback is to help the user navigate the interface and better



Shows feedback to prevent errors.



A simple interface to turn on/off a power receptacle.

understand what is going on.

Simplicity

The key to a successful GUI is to distill it down to the basics. A simple interface is easier to use and takes less time to understand. If a user has to trudge through the manual to figure something out, it's probably too complicated and will likely lead to more errors for the user and more bugs for the developers to clean up.

Find out as much as possible about the features required by your endusers. You know who your audience right? Don't try to

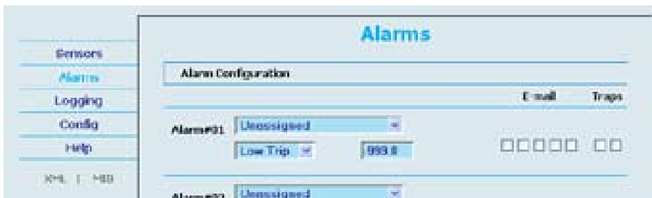
develop in a vacuum, guessing at what people might use. Go ask your target market and find out their necessary features. Get rid of the rest. This will simplify the design and keep you from spending time adding features no one wants or will use. Once the users get your interface they'll give you feedback on how to make it better. That's the time to add extra features. This helps you get your product to market quicker and ultimately should save you money.

Attractiveness

This can be a difficult issue to handle. People have different opinions about what looks good. Again a good place to start is to find out which websites or interfaces your users like. Consider the color schemes and basic layout of these sites. If the site uses HTML or HTML and CSS, you can look at the page source code to find out the colors used and how the site is put together.

Maintenance

It's just about impossible to predict what the requirements will be for future products or which new features we might add to current ones. About the only thing that can be counted on is change. We have to consider this when creating a GUI. One area in



With a left navigation bar a new tab can be added without effecting the text on the right.

particular is the nav-bar.

In earlier products we used a nav-bar at the top. This was fine for a while, but over time we kept adding new items to this

bar. Eventually it got too crowded and some of the items had to be moved into the header. Now, we put the nav-bar on the left side of the screen. This way it will grow vertically instead of horizontally. Users seem to tolerate scrolling down instead of sideways.

Another important part of this design was to make each button in this bar a `div` element. To add a new button we just add a new `div`. If the the nav-bar was a series of images, or worse one single image, then every change would require the creation of new images. Because of situations like the following, we have to consider the how difficult it may be to add or subtract elements from the design. Some hints:

1. If text is used in an image, then a new image must be made when a new item is added to the menu
2. If CSS is used it can help isolate changes to a single document
3. If an image is used for buttons then a background template can speed up the process

HTML

We commonly write HTML by hand. There are many programs a person could use to design web pages without having to mess with HTML, but you never know what sort of HTML it will create. In an embedded environment, memory is at a premium, so we can not afford to use bloated HTML code created by another program. We can also ensure the HTML is clean and structured the way we want.

We start with an editor that can properly color HTML. This helps us spot mistakes like a missing quote on an attribute. A good practice we use when working with an editor is to indent the portion of text inside a tag. This helps us make up tags to ensure they are closed and helps make the structure of the document more apparent.

Another invaluable tool is a web page validator. We use the

web-based one provided by the W3C. This helps catch a complete array of bugs, like the dreaded missing `close` tag. By using the right tools we can produce compact and readable HTML files.

When creating an HTML file it is important to test it with many different browsers. There are a variety of known bugs in the different browsers. Many of the bugs already have a solution that can be found by searching the Internet.

Some guidelines for writing HTML:

1. Indent the text inside a tag.

2. Validate your HTML often to catch mistakes early.

3. Test your HTML files with many browsers to find compatibility problems.

Stylin' with CSS

HTML provides reasonable control over the appearance of a web page. However, CSS (Cascading Style Sheet) provides much greater control. We use CSS to globally set the appearance of things like links, font type, and background colors. In the following sections we give a brief introduction to CSS.

There are several ways to style with CSS. An attribute "style" can be added to just about any tag. This allows control of the appearance of just one tag at a time. We try to use this only when one item needs to be styled.

If we want to change the styling for many instances of a tag then we use one of the other methods. For this we use a stylesheet. These can be internal or external. An internal stylesheet is placed in the head section of the HTML file, but we don't use this method because internal stylesheets can't be shared between HTML files. This is one of the great strengths of CSS, because it creates consistent styles across many pages. For this reason we use external stylesheets. To create an external stylesheet you just need to create a text file with CSS commands and save it with a .css extension. Tell the HTML page to use this stylesheet by adding the following

command in the head section:

```
<link rel="stylesheet" type="text/css" href="file_name.
css">
```

CSS Syntax

Now that you know how the include style commands in a document, let's actually look at the syntax of CSS.

The basic syntax for CSS is:

```
selector { property: value }
```

Let's look at each part of this individually just to get an idea of what it would look like. The following command sets the background color of all `div` tags in the page to green:

```
div { background-color: green; }
```

If you are applying the style attribute to an HTML tag then you only use the property: value part. For example, let's say you want to set the background color of just one `div`. The line of code looks like:

```
<div style="background-color: green;"> some_material_
in_here </div>
```

Now let's look at the component parts of the CSS syntax.

Selector

The first part is the `selector`. As the name implies this tells the browser which items you will be setting the attributes of. There are several types of selectors. In the example above the selector chooses the `div` tag. Most of the other tags in HTML can be used in the same way. Two other common ways of selecting are by id and by class. An HTML tag can have an id or class attribute. If id is used this needs to be unique in the document. For instance to style a certain h1 tag use the following command:

```
<h1 id="maintitle">Tiny Web Servers</h1>
```

This makes a unique item with the id of "maintitle." To select this in CSS put an '#' before the id as the selector. This is shown below:

```
#maintitle
{
  font-size: 20px;
  color: green;
}
```

Make sure that only one tag in the entire HTML file uses "maintitle" for an id. This works for styling single items, but often you want to style all items of a given type. CSS provides a class selector. In HTML add the attribute class to a tag. For example:

```
<h1 class="original"> Title </h1>
<p class="original"> Some text </p>
```

As you can see class can be applied to multiple tags and those tags don't have to be the same type. To style these tags you put a '.' in front of the name, like in the following example.

```
.original
{
  color: green;
  background-color: blue;
}
```

When several HTML elements share style attributes you can group them together. For example, if you wanted to set the color of the text inside a `div` and `p` element the code is:

```
div, p
{{
  color: red;
}
```

There are many other ways to use selectors to refine the selection of elements to style. Below are several other examples:

```
p.original
{
  color: blue;
}
```

This selects elements of `p` with a class of "original".

```
div p
{
  text-align: right;
}
```

Descendants are elements contained inside a parent element. In this case this selector looks for `p` element descendants of `div`. In other words `p` elements contained within a `div` element.

The above types of selectors are supported well by most browsers. There are some selectors that let you select with even finer control, but some browsers don't support them. There are many good tutorials and references on the Internet if you are interested in learning about the other selectors.

Now let's look at the property portion and value portion of CSS syntax.

Property/Value

Property/value pairs describe the style that will be applied to an HTML element. There are many properties that you can set for an element. The following is a small set of them.

```
background-color - ex. background-color: green;
background-image - ex. background-image: url('bg_im-
age.gif');
color - ex. color: #ff0000;
text-align - ex. text-align: left;
font-size - ex. font-size: 12px;
font-weight - ex. font-weight: bold;
border - ex. border: 5px;
margin - ex. margin: 10px;
padding - ex. padding: 4px;
width - ex. width: 45%;
height - ex. height: 105px;
```

The value part of a property/value pair can be one of several

units. For colors one of the following forms can be used:

```
color_name - ex. red
rgb(x, x, x) - ex. rgb( 255, 0, 255 )
#rrggbb - ex. #ff00ff
```

For measurements some of the common units are:

```
% - percentage of space available, grows with increase
in screen size
em - equal to the current font size, used to adapt to
user defined font size
pt - point (1/72 of an inch), often used to specify a
font size
px – pixel (one dot on the screen), often used in fixed
width webpages
```

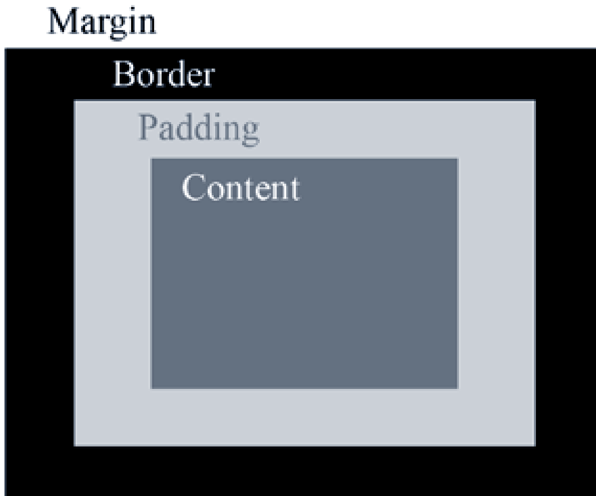
As shown in some of the previous style examples, more than one property/value pair can be used when writing a style. Each property/value needs to be separated by a ';'. Technically the last property/value does not need to end by a semi-colon, but it's normally a good idea to include it anyway. Often styles are changed while developing a web page, so ending every property/value with a semi-colon helps protect against errors.

CSS Box Model

An important part of styling in CSS is understanding the CSS box model. In CSS each element is viewed as a box. This box has 4 areas. The inner most layer is the content area. This is where text and images are placed. Surrounding the content area is the padding area. Padding is used to separate the content area from the border. The border surrounds the padding area. The last area is the space around the outside of the box. This is called the margin. This space is used to provide space between boxes. The combination of the sizes of the content area, padding and borders gives the final size of the box. For example suppose the box is defined as follows:

```
#header
{
  width: 750px;
```

```
  padding: 5px;
  border: 1px;
  margin: 10px;
}
```



Sample of a css box

Applying this style to an element, the resulting box would has a content area 750 pixels wide. Around this is a padding area of 5 pixels on each side. The border encloses the padding adding one pixel to each side. The final box has a width of 762 (750 content + (2 * 5 padding) + (2 * 1 border)) pixels. The margin around the box pushes it away from other boxes by 10 pixels.

There is a known bug in Internet Explorer where it uses it own box model instead of the CSS box model. In this model if you set the width of a box it refers to the whole box including the padding and border. In the CSS model this width only refers to the content area. This difference causes boxes in IE to be smaller than those that use the CSS box model. This difference is supposed to be fixed in IE version 7. Many work-arounds exist. One work around is to

enclose a `div` within a `div` then apply a width to the outer box and the padding and border values to the inner box.

This has been a brief tour of CSS. There are many great tutorials on the web. We recommend the CSS tutorial at the W3 Schools (http://www.w3schools.com)

CSS Positioning

In the past web page layout was usually done with tables. This was for good reason. Tables are supported correctly in all browsers. Conceptually tables make sense. There are some arguments against using tables though. One is that they tie layout and content together. Some developers believe they have better control putting content in HTML and creating the layout in a CSS file. An upside to separating them is that the layout of the HTML elements can be significantly altered just by editing the CSS file. The current trend favors using CSS to control layout and only putting content in an HTML file. The decision involves weighing the benefits of separate layout versus inconsistencies in how browsers render CSS. As support for CSS improves this should become less of an issue.

Instead of using tables the element most commonly used is `div`. A `div` is just a container for others things like text, images or another `div`. This is the main building block. The key to layout with CSS is positioning these containers. Other elements can also be positioned like span. We'll go through five ways to position a block. Before we go into these the term normal flow needs to be explained. Normal flow is the layout (position) of HTML elements if you don't use any CSS positioning. This is important because some elements get removed from the normal flow during positioning. If they are moved then the elements around it are positioned like as if they never existed. Here are the four position types. We'll cover the fifth way to position in the next section.

1. **static:** This is the default position for the element (normal flow). In this case CSS is not positioning the element. All position

offsets will be ignored (we'll get to these later).

The text "test" is styled with a static position as shown. The HTML file used to create this image follows.

The test box is styled with `position:static`.

```
<html>
  <head>
    <title>Position Tests</title>
    <style type=”text/css”>
      #test
      {
        background-color: #aaa;
        position: static;
        top: .75em;
        left: .5em;
      }

      #content
      {
        border: 1px solid black;
      }

      body
      {
        font-size: 2em;
        padding: 1em;
      }

    </style>
  </head>

  <body>
    <div id=”content”>
      This is a <span id=”test”>test</span> of CSS po-
sitioning. This is a test of CSS positioning. This is
a test of CSS positioning.
    </div>
  </body>
</html>
```

2. relative: The element is placed in the normal flow and then offset by some amount. Since the element is first placed in the normal flow, the other elements around it are positioned as if the

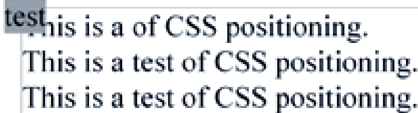
element was still there. This leaves a gap after the element is moved as shown. The change made in the test style code is also given.

The effect of `position:relative`.

```
#test
{
  background-color: #aaa;
  position: relative;
  top: .75em;
  left: .5em;
}
#test
{
  background-color: #aaa;
  position: absolute;
  top: .75em;
  left: .5em;
}
```

3. **absolute:** With this type the element is never placed in the normal flow, so there is no gap. The element is positioned a certain distance from the origin. This origin is the top-left corner of the container holding the element, that has a position other than static. If all of the parent containers are static then the origin is the top-left corner of the browser area.

This figure shows what happens when the position of "test" is



The "test" box changes position with absolute.

changed to absolute. It is in a very different position from relative. In this case the origin was top-left corner of the browser area as shown when the offset is reduced to 0. This can be easily seen next.

test
This is a of CSS positioning.
This is a test of CSS positioning.
This is a test of CSS positioning.

The "test" box moves to the origin when the offset is 0.

The browser ignores the top-left corner of the content area, because it has a static position. To get it to use the "content" area as the origin, the content must have a position other than static. Next we see the results of setting this to relative.

tests is a of CSS positioning.
This is a test of CSS positioning.
This is a test of CSS positioning.

The origin for "test" has changed to the content area.

```
#test
{
  background-color: #aaa;
  position: absolute;
  top: 0em;
  left: 0em;
}
#content
{
  border: 1px solid black;
  position: relative;
}
```

4. **fixed**: Like absolute, with fixed positioning the element is removed from normal flow. Then its place is based on an offset with the origin at the top-left corner of the browser area. It ignores all parent containers regardless of what position type they use. Also this element stays fixed at this location even if the user scrolls the page. One thing to note is not all browsers support this type of positioning.

The next page shows what happens when the position on the "test" box is set to fixed. Notice it ignores its parent container even though it has a non-static position. The full HTML text for this example follows the figure.

test
This is a of CSS positioning.
This is a test of CSS positioning.
This is a test of CSS positioning.

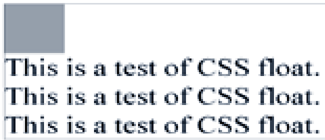
Fixed positioning uses the browser area as its origin.

```
<html>
  <head>
    <title>Position Tests</title>
    <style type=”text/css”>
      #test
      {
        background-color: #aaa;
        position: fixed;
        top: 0em;
        left: 0em;
      }

      #content
      {
        border: 1px solid black;
        position: relative;
      }

      body
      {
        font-size: 2em;
        padding: 1em;
      }

    </style>
  </head>

  <body>
    <div id=”content”>
      This is a <span id=”test”>test</span> of CSS po-
sitioning. This is a test of CSS positioning. This is
a test of CSS positioning.
    </div>
  </body>
</html>
```

Float

The position property is used some in CSS layouts, but the more common way to layout elements is to use `float`. In the past support for floats was poor, but now most browsers have good

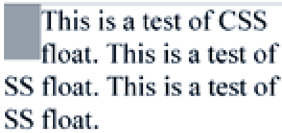
support. To `float` an element use the property `float` with right, left, or none for the value. In the following example `float:none` is used of the style "box." This is the normal flow for this element, as shown here. The HTML we will use for this example follows.



Shows the normal flow for "box."

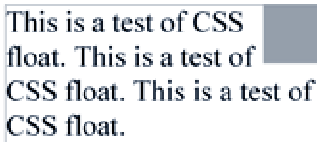
```
<html>
  <head>
    <title>Float Tests</title>
    <style type="text/css">
      #box
      {
        background-color: #aaa;
        float: none;
        width: 2em;
        height: 2em;
      }

      #content
      {
        border: 1px solid black;
        position: relative;
      }

      body
      {
        font-size: 2em;
        padding: 1em;
      }

    </style>
  </head>

  <body>
    <div id="content">
      <div id="box"></div>
               This is a test of CSS float. This is a
test of CSS float. This is a test of CSS float.
    </div>
  </body>
</html>
```

Now let's change from `float:none` to `float:left` or `float:right`. The HTML follows for `float:left` and HTML for `float:right` is obvious.



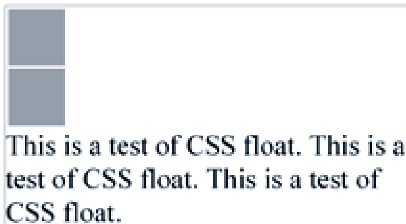
Shows the "box" `float:left`.

```
#box
{
  background-color: #aaa;
  float: left;
  width: 2em;
  height: 2em;
}
```



Shows the "box" `float:right`.

Notice that along with `float` there is a width property. A width is required when floating an element. What `float` does is to remove the element (now with a given width) and aligns it flush with either the left or right side of the parent container or the edge of another `float` element. A `float` of none is the default for elements, so no `float` is done. The content after the floated element flows around the floated element. Let's look at an example where two elements will eventually be floated. We'll start with the elements in normal flow, as shown here. The HTML for this

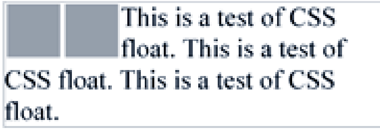


Two of the "box" class in normal flow.

example follows.

```
<html>
  <head>
    <title>Float Tests</title>
    <style type="text/css">
      .box
      {
        background-color: #aaa;
        float: none;
        width: 2em;
        height: 2em;
        margin: .1em;
      }

      #content
      {
        border: 1px solid black;
        position: relative;
      }

      body
      {
        font-size: 2em;
        padding: .25em;
      }

    </style>
  </head>

  <body>
    <div id="content">
      <div class="box"></div>
      <div class="box"></div>
      This is a test of CSS float. This is a test of
CSS float. This is a test of CSS float.
    </div>
  </body>

</html>
```

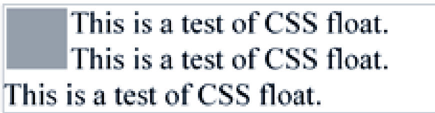
Now lets `float` the two boxes to the left. This allows the text to flow around as in the previous example. Notice here though the boxes line up horizontally. See the figure on the following page. The rule is that a floated element goes as far to one direction as possible until it meets the border or another `float`. (Shown on an earlier page.) The code change for two boxes `float:left` is given following the figure.



Two "box" divs float:left.

```
.box
{
  background-color: #aaa;
  float: left;
  width: 2em;
  height: 2em;
  margin: .1em;
}
```

But if we want to stop an element from flowing around a floated one, the property to apply is `clear`. The values for `clear` are `left`, `right`, `both`, and `none`. An element with the `clear` property goes below the elements on the side indicated. We'll start out again with elements in the normal flow. The HTML used to



Nothing is cleared with clear:none.

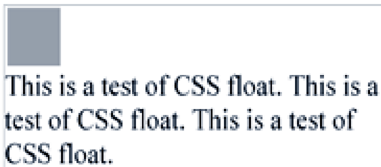
create this example follows the figure.

```
<html>
  <head>
    <title>Float Tests</title>
    <style type="text/css">
      .box
      {
        background-color: #aaa;
        float: left;
        width: 2em;
        height: 2em;
        margin: .1em;
      }

      #content
      {
        border: 1px solid black;
```

```
      position: relative;
    }

    body
    {
      font-size: 2em;
      padding: .25em;
    }

    #text
    {
      clear: none;
    }

  </style>
</head>

<body>
  <div id="content">
    <div class="box"></div>
    <div id="text">
      This is a test of CSS float. This is a test of
CSS float. This is a test of CSS float.
    </div>
  </div>
</body>
</html>
```

Now let's make the change below and use `clear:left`. This will give the placement that follows.

```
#text
```



Result of using `clear:left`.

```
{
  clear: left;
}
```

We've covered only some of the issues involved with using

`float` for positioning. Searching the web will provide more details about CSS positioning.

Some hints:

A `div` with a class of clear that applies only `clear:both` is useful to make a parent container big enough to contain a floated element if it is the last element in the parent.

```
.clear
{
  clear: both;
}

<div>
  This is a test of a “clear” div.
  <div id=”box”></div>
  <div class=”clear”></div>
</div>
```

It can be difficult to create columns with an equal height. There are several “tricks” to get this effect. One is called faux columns. Start by enclosing the two columns you want to be the same size within a parent `div`. Add a background image to the parent `div` that repeats only in the y-axis. The image to repeat is a single line the width of the parent `div` with colors the same width as the children columns.

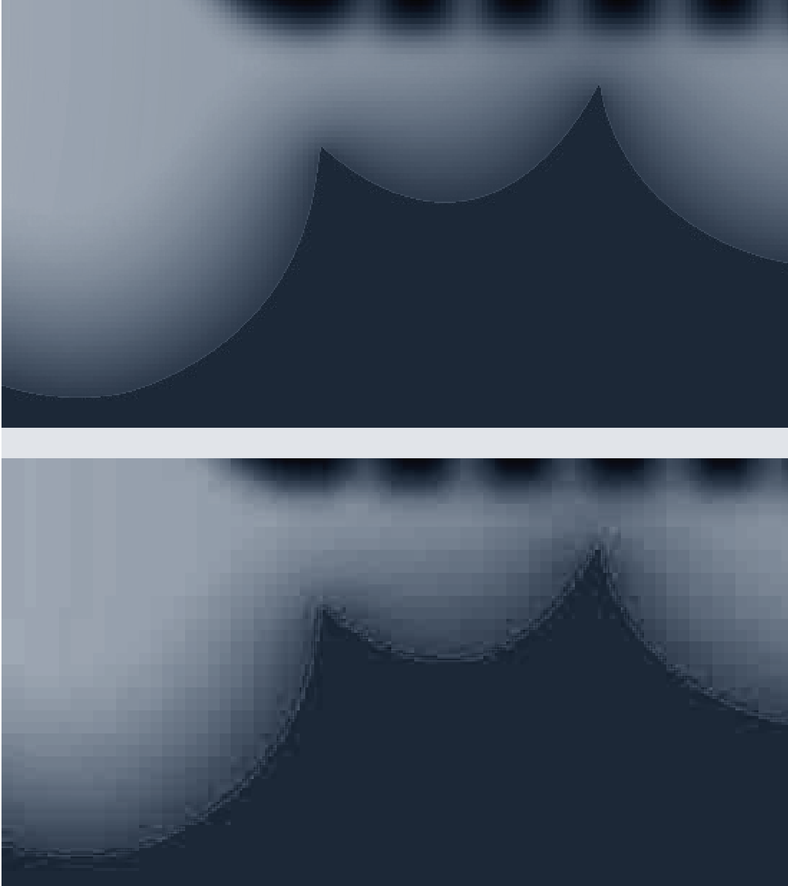
Images

There are many image formats available, but the two most common types used on the web are gifs and jpgs. Format png is used some, but currently browser support for this image format is inconsistent. When browser support is corrected for png it may become one of the predominant image formats.

JPG (JPEG)

We use this format for full-color images. It does a good job taking an image and compressing it to a reasonable file size. The downside is that compressing the image degrades it. You can see this in the figure on the following page. There is a trade-off here,

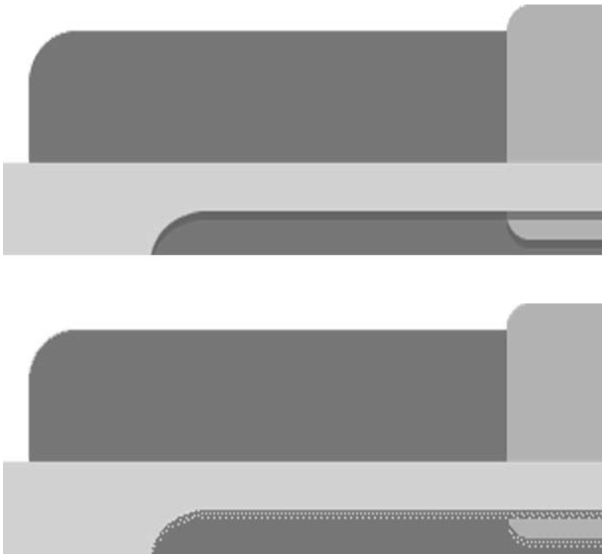
the more compression the smaller the image size, but also the more degraded the image. Some image editors like Adobe Photoshop have a preview function that allows you to see how an image looks at different compression levels and gives the size of the final image. This really helps in finding the right ratio between image quality and image size. Again, remember that every time the image is saved the quality degrades, because the image is recompressed.



Close-up of a JPG showing the difference between low compression (top) and high compression (bottom).

GIF

GIF is the format used for limited color images and ones where we need transparency. Another use for gif is when we need sharp edges. With jpgs edges can be blurred. Normally we use gifs for the image portions of an interface. Gifs are reasonably small in size. The compression scheme used does not degrade the image. The fewer colors used the smaller the image. The trade-off is how few colors can you use before the image starts to look bad. This is shown in the figure below.



Comparison showing how limiting the number of colors can degrade an image. The top uses 16 colors, while the bottom uses only 4.

Compare the shadow areas in both images. These are the sorts of places where gifs start to degrade as the number of colors is reduced. Photoshop has a useful tool for saving images as gifs. With it the final image can be previewed while decreasing the

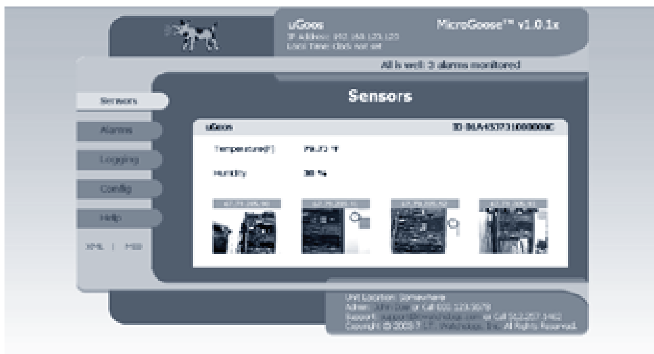
number of colors used. The tool also allows you to select the most important colors in the image, so they will be saved when reducing the number of colors.

Types of GUIs

We've identified three types of GUIs. We call them Contemporary, Midrange, and Austere. They each serve a specific purpose. The majority of our product line uses the Midrange GUI. Our new products have all three types allowing the user to select the one that best suits his needs.

Contemporary

This GUI is based on the common layout of a web page which appears in the example here. By creating an interface that takes



Example of a Contemporary style web page with rounded corners.

advantage of a fixed width layout we have the ability to place elements in predetermined locations. Regardless of the screen size the layout turns out the same on the user's screen. Images can be used to create large portions of the interface. These images fit together properly because of the fixed width. The downside is this requires more bandwidth. On a unit with a slow web server this may lead to a poorly performing interface.

The Contemporary layout typically consists of a box centered on the screen with a width approximately 750 pixels. The top portion has a header with a logo and some important information for the user.

The next area contains buttons that allow you to navigate through the interface. Sometimes this goes across the top with the buttons placed horizontally. Our preferred design is to put the buttons vertically down the left side of the interface. We did this to allow more room if the number of navigation buttons increases.

Top navigation bars (nav-bar) limit how many buttons can be placed on a line. We also take advantage of the fixed layout to use images to create the interface. The keep the interface small and to load quicker the images are gifs using the least amount of colors possible. With the nav-bar on the left the content is placed on the right. The layout is finished with a footer giving support information.

Midrange

What we call midrange is characterized by a layout width that expands to fill the full width of the browser window. This is sometimes called a liquid layout, and was more common a few years ago. It is still used but it seems the trend is toward more fixed layouts. This interface has few images, so it loads quickly, making the interface more responsive. Faster response time is a major factor in why we continue to use this interface. In our version of Midrange the header has the same information as the header of Contemporary including a logo, unit info, and version.

Following the header is the nav-bar. Originally we used a top nav-bar but over time we added more buttons to the point where it became crowded and we had to move some of the pieces into the header. This is part of the reason why we went to a side nav-bar in our current design. To reduce bandwidth the layout is created without images as shown on the following page.

Example of a Midrange web page.

Austere

We put this interface together because of customer requests. It is designed to be very minimal, mostly text, and it was created for users who just want the data withou a full figured interface.The header contains the same information as before.

Again, we used a left nav-bar to allow room for growth and change. The content is on the right, web page followed by the footer at the bottom. The placement of the elements is nearly identical to that of the Contemporary with new styles added. We use a CSS style sheet and CSS positioning to change between the different types of interface GUIs.

Examples: Contemporary Owl Web GUI

The Owl GUI uses stylesheets to transform the interface from Contemporary, Midrange and to Austere. It has a second Mid-

Example of an Austere style web page.

range GUI that looks like Contemporary, but with the interface images removed. This can be seen on the next page.

Finch GUI

This interface is of the Midrange GUI style. Since the Finch has limited processing power we had to use a light-weight GUI. The

Example of a "lite" web page. This is a Finch web page.

logo is the only image that is loaded from the webserver. Even with the limited power of the Finch, the interface is quite responsive.

Graphing

Knowing the current temperature or humidity value is useful. But what was the temperature like last night? Did the AC come on? Many of our customers had this concern. It was obvious we needed to have graphs. Sure the customer could write a program or script to hit the unit every minute and save the data to a log. Then later ran it through a program to get graphs. This takes time and expertise on the customers side, and what if they want to view it over the web? Putting these graphs on the web for the customer is a hassle. Thus began the task of adding graphing to our web server.

To create a graph first we need data. The unit needs to take readings at some interval and save the points to be later used in graphing. This data logging portion can be tricky. Storage space in an embedded product is scarce. Sometimes there is a trade-off between accuracy and memory usage. Maybe certain measurements only need to use a byte, while others might need two bytes. The logging system can be simplified by using the same

Finch web page with extra images.

PView PDU
PowerView XL v1.19
192.168.123.123

PV Solutions

Sensors | Configuration | XML | MIB

Total Amps

Phase A	0.03
Phase B	0.02
Phase C	0.04
Neutral	0.05

Phase AC - 208 Volts

Outlet	Name	Amps	Friendly Name
18	AC-4	0.02	*Outlet18, top*
17	AC-3	0.02	*Outlet17*
16	AC-2	0.02	*Outlet16*
15	AC-1	0.02	*Outlet15*

Phase A - 120 Volts

Outlet	Name	Amps	Friendly Name
14	A-2	0.02	*Outlet14*
13	A-1	0.02	*Outlet13*

Phase AB - 208 Volts

Outlet	Name	Amps	Friendly Name
12	AB-4	0.02	*Outlet12*
11	AB-3	0.02	*Outlet11*
10	AB-2	0.02	*Outlet10*
9	AB-1	0.02	*Outlet9*

Midrange Finch web page featuring few graphics with dominant text.

number of bytes for each reading. But do we use one byte per reading to save memory, at the expense of losing the accuracy of a two byte reading? Or do we pad out the one byte readings to two bytes and waste memory? Of course a more complicated system can be used to store the one bytes as one and the two bytes as two. This is just one of several concerns.

Another concern is the logging frequency. What happens if the user changes it? Do you delete the old data and start over? Maybe you could open a new “file” and start logging into that location. You could choose to forgo this problem by fixing the logging interval. Some people might complain, but that might be better than fighting bugs in a more complicated system.

Yet another issue is timestamping the data. You may have a bunch of data saved, but you need to know when it's obsolete. A common timestamp is something like the number of seconds since a certain point in time. Or maybe the number of clock cycles since the device started. Using seconds is ok, if the device always knows what time it is. The device either needs a realtime clock with backup power or a way of getting the time from the network.

The device could connect to an NTP server but what happens if it can't connect to NTP? This is the reason why we moved to an onboard, realtime clock. Once set, the clock keeps track of the time even if the unit is off. The trade-off here is the hardware design and cost issues versus the time trying to fix the problem in software.

Once we have time-stamped data stored in some form in memory, one last issue is to put a header on the data to identify it. What device is this data logged for? What is the format of these bytes of data. To accomplish this, a header id's what format is used for the data and for which device. Now you can graph.

Building the Graph

Now we are ready to graph! We start by figuring out the x and y axes. The x-axis will be a time range. If we graph everything then this range would go from the oldest timestamp to the newest. This involves going through each header in memory and determining this range. The y-axis consists of the range of data values. The min and max values of the sensor are fixed of course, but if the data typically stays in a small range, graphing the complete range loses a lot of detail. Therefore, we use auto-ranging.

We look though the data for each sensor we are going to graph and find the overall min and max of the data. We use this for the range of the y-axis. Since the physical size of the graph is fixed we can now determine how to divide the x and y axes into steps or ticks. Then we can draw the axes with labels.

Drawing the data is the next step. We use line graphs to show

trends in the data and to be able to fit graphs for many devices on one graph. This invites comparison between readings. Maybe you'll notice that the sound level increases when the light level is high. Suppose the temperature drops then also. This might be evidence that someone came into the equipment area, turned on the lights and turned down the AC. Another good reason for using a line graph is that it is straightforward.

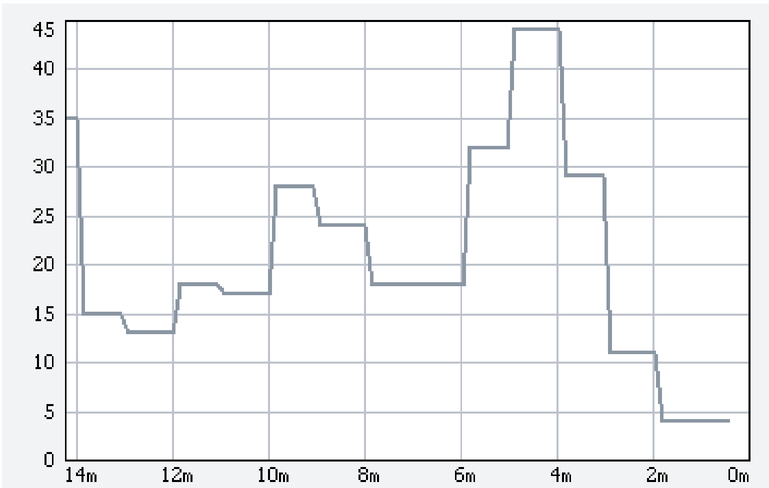
To graph, pick a series of points between the start of the time period to graph and the end. The number of points you pick will be determined by how large the graph is and how detailed you want it. Then lines are drawn connecting the points and a line graph is displayed. Do this for all of the sensors to be graphed and you're done.

One thing to note is artifacting that could occur when selecting the points to graph. Although one way is to pick points at a fixed intervals across the time span, a problem is that spikes in the data may be missed. The graph may appear to change abruptly between subsequent renderings. Another method is to average points across a span to create a data point. The downside to this is that now averages and not actual values are graphed, it also takes longer and ties up more resources.

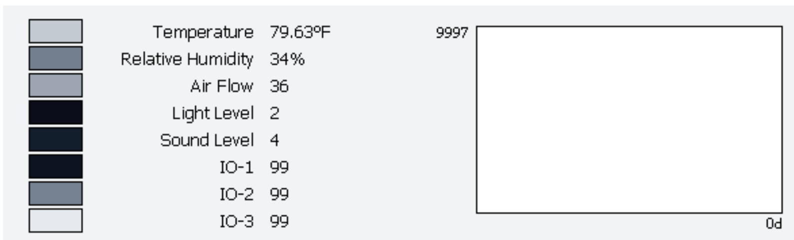
One last method is to look through the data for spikes or other areas of interest and graph them. This method would probably require the most resources of the three methods to implement. These costs and the limited resources of an embedded system have to be taken into consideration.

We use the first method of regular time intervals because it is fast to calculate and it does a good job showing basic trends. Any spikes of interest trigger an alarm condition. If a customer wants to view the data in more detail, he can download the full data log.

The last piece of the graph is a legend. Without it a viewer won't be able to identify which data relates to which sensor. The graphs on the following pages demonstrate the topics in this

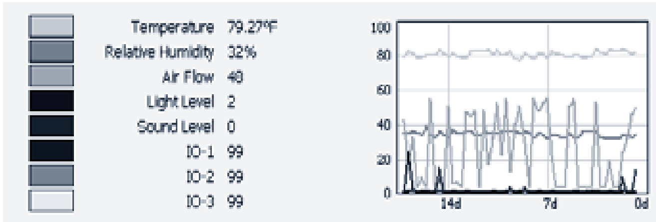


A graph example showing the y-axis scaled to fit data.



A typical graph before figuring out the axis ranges.

section. The y-axis is set for the range of data, so that it gives enough detail to make comparisons. The x-axis shows the time range of the data, over 14 days. The lines are color-coded and tied to a legend that identifies the sensor.



A graph with data including a legend.

This section shows the importance of graphing to see trends in the data and make comparisons. We've demonstrated many of the steps in the process and have shown where problems can occur.

Ron McCormack - Software Developer

Comparing the Two Servers

Hardware Comparison

Software Comparison

Physical Size

Which One to Use?

Before we compare the two systems, let's look at what the two products have in common:

Software Source Code: All the source code is on-hand at Geist Technologies.

Programming Language: All software is written in the "C" programming language.

Microprocessors: We use Philips (ARM) and Microchip processors. Both are readily available, low-priced and well supported. Specialized, non-commodidty integrated circuits are avoided.

Size: The smallest is the Finch which can be reduced to a few tiny integrated circuits. The Owl is slightly larger.

Ease of Assembly: The Finch can be assembled by hand or by basic "pick and place" automatic circuit board assembly machines. The Owl, however, requires BGA (Ball Grid Array) assembly which must be machine assembled with late-model assembly equipment. The cost of the BGA Owl assembly is slightly higher than the Finch processor.

Costs: You can look up the current costs of the components through DigiKey (catalog component house) or almost any component distributor. Owl components cost over twice what the Finch does. The primary reason for the Owl's extra cost is the more sophisticated ARM processor and the dual external memories. The Finch will likely be the lowest cost circuit for at least the next year.

Additionally, the printed circuit boards are more expensive for the Owl because the highly dense circuit connections on the Philips ARM processor require an eight layer board layout which can cost more than twice the four-layer Finch board. This can add a lot of cost to a product.

Our solution to the higher cost eight-layer board was to make the Owl Cube. This design reduced the eight-layer board to postage

Comparison Table Between Owl & Finch Hardware

	Owl	Finch
Processor	ARM 7	Microchip PIC
CPU Clock Speed	70 MHz	25 MHz
Code Flash	None	32 KB or 64 KB
CPU Clock Speed	70 MHz	25 MHz
CPU Register Size	32 bits	8 bits
External Code Flash	Up to 16 MB	None
Internal RAM	64 KB	2 KB
External RAM	Up to 4 MB	None
External Data Flash	Up to 2 MB	32 KB
MAC / PHY	Microchip PHY	Microchip PHY
Outboard sensors	Yes, via a PIC	No
Reset Button	Yes	Yes
General-purpose pins	5	0
Cost of Components	$30	$15
Current Consumption	150mA	120mA
Remote Firmware Updates	X	X
Ethernet Speed	10-base-T	10-base-T
RS-232 Serial I/O	115000	9600
SPI (Serial Peripheral I/F)	X	X
GPIO	Up to 45 pins	Up to 37 pins
Real-Time Clock (Battery)	X	
I2C Serial Interface	X	X
Debugger Interface	JTAG	Microchip ICE
Hardware Watchdog Timer	X	X

stamp size. The main printed circuit board could be an inexpensive four-layer design and the Owl Cube simply soldered in with a low-cost receptacle.

Growth Versus Cost

The Finch can be thought of as a minimalistic approach to a web server. The absence of SSL (Secure Socket Layer) encryption probably eliminates all but the simplest control applications and uses whereas the data placed on the web does not require any kind of security.

On the other hand, the lowest cost of the components is certainly attractive in applications where price must be kept to an absolute minimum. High volume applications usually require a low cost to manufacture and this is where the Finch has a strong appeal.

Many applications require complex graphing, multiple alarms, and logging functions and this requires the processor power of the Owl's ARM processor.

The Owl has outboard memories which means you can select the amount you need for the application. The Owl's operating system gives you the ability to prioritize which tasks need quicker service than others. Also, if the application needs complex algorithms or formula calculations, these can be easily accommodated in the Owl.

If the initial application is viewed as a starting point with more applications to be added continually, then the Owl is the definite choice.

Using Other ARM Processors

In applications where requiring specialized operations, such as digital signal processing (DSP) or other high processor load, the Owl's ARM 7 can be replaced with a member of the ARM family that contains that enhanced capability. In most circumstances this can be accommodated with minimal software changes.

Software Comparison Table

	Owl	Finch
Operating System	FreeRTOS	None
Flash File System	X	X
OEM Message Catalog	X	
Device Manager/Driver	X	
Dynamic Device Detection	X	
Error Handling	X	
Logging	X	
Graphing	X	
Alarms	X	
SYSLOG	X	
Friendly Names	X	X
Non-Volatile Configuration	X	X
Email		
SMTP e-mail	X	
POP-AUTH	X	
Plain AUTH	X	
MD5-AUTH	X	
Reporting		
XML	X	X
Excel Logging	X	X
Network Monitoring		
SNMP v1	X	X
SNMP Get	X	X
SNMP Set		
SNMP Trap	X	
Development Tools		
Dev. Toolchain	ARM GCC/ GDB	

Software ComparisionTable, Continued

	Owl	Finch
Network		
SSL	X	
ARP	X	X
DNS	X	
Ping	X	X
UDP	X	X
TCP	X	X
FTP		X
DHCP	X	
IPv4	X	X
IPv6		
TCP Retransmission	X	
Out-of-Order Segments	X	
SNTP (Time Server)	X	
Web Server		
Webserver	X	X
Webserver Threads	2	1
GET	X	X
POST	X	
HEAD	X	
If-Modified-Since	X	
Basic AUTH (Password)	X	
Dynamic Web Pages	X	

The tables compare the differences between the Finch's minimalist approach and the Owl's full-featured approach. The Finch uses no operating system - the programs run endlessly in a loop. The Owl, however has a sophisticated operating system capable of running tasks in time-slice coordination. The Finch is built for minimum cost with a bare-bones software approach. When more flexibility or more features are needed, the Owl is the best choice.

SNMP

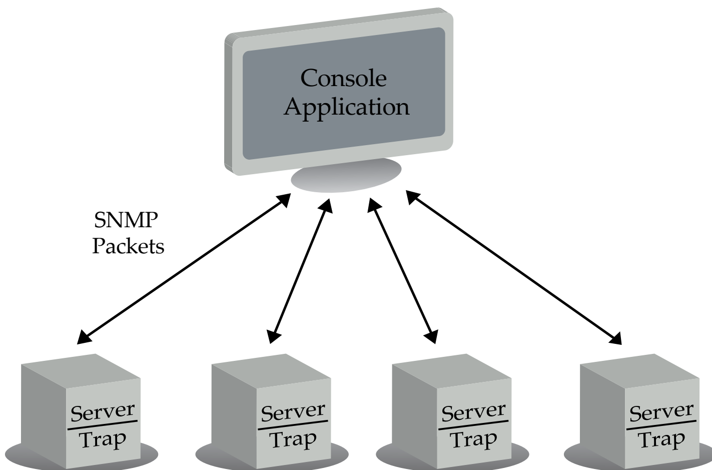
Communicating with Network

Monitoring Systems

Why SNMP is Useful

The Simple Network Management Protocol (SNMP) is a client/server method allowing a remote user to view management information held in a networked device. The SNMP method was defined by RFC 1157.

A Console Monitoring Program with Multiple SNMP Devices



Remote embedded processor devices are shown minitored by a central console program. When alarm levels are exceeded, the remote units send an SNMP "trap" which alerts the console to an out-of-limit condition. SNMP is a commmmon means of remote monitoring.

For example, an SNMP application (client) on the remote user's system uses the protocol to converse with an SNMP agent (server) on the device to retrieve the management information. The SNMP agent provides information about the

device's current state, such as the device's network interfaces or device specific data (e.g. temperature).

The provided information is stored in a format specified by the Management Information Base (MIB). The MIB is organized in a hierarchical tree structure with SNMP objects represented as leaves on the tree. The Object Identifier uniquely distinguishes each variable. The MIB was defined by RFC 1155, 1212, 1213. The MIB specifies the SNMP objects that one can manage and their format.

SNMP objects are referred to as Object Identifiers (OIDs) which are unique and paired with a value (e.g. 1.3.6.1.4.1.9999.1 might have the value 77). The digits in the OID number represent levels in the hierarchical tree. The MIB maps the complicated OID to a human-understandable token (e.g. TemperatureValue7).

SNMP is highly complicated to implement. OIDs are defined as byte strings in multiple length data packets along with their values. Multiple OID/values can be sent in a data packet.

SNMP Operations

An SNMP application reads and/or writes values from the SNMP devices. The four SNMP operations are:

• Get — request the values of one or more SNMP objects

• Get next — gets the next object in a table one row at a time

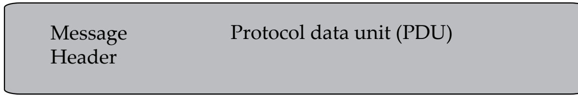
• Set — sets the value of an SNMP object

• Trap — sends a packet about an event to the SNMP application

The device's SNMP agent listens on Universal Data Port (UDP) port 161 for SNMP application requests. Trap messages are sent to UDP port 162. Traps are sent by the device's SNMP agent to notify an SNMP application of a significant event.

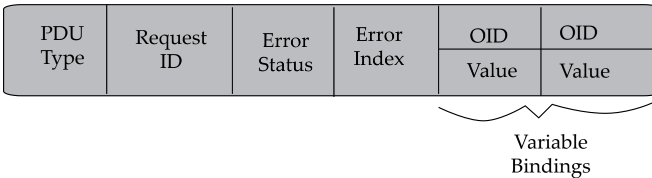
When an SNMP application wants to know the value of a MIB

SNMP Version 1 Packet Format

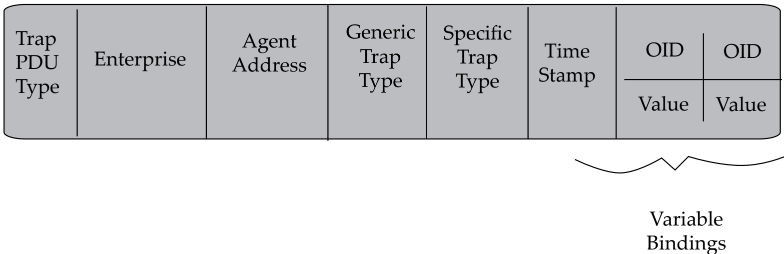


Get, Get Next, Set, Response PDU

Protocol data unit (PDU)



Trap PDU



token (or OID) of a device, it issues a UDP get request to port 161 containing the OID to the IP of the device. The device then replies with a get response ("get response" is the fifth type of message) containing a status and the OID pair requested.

When an SNMP application wants to know the value of MIB tokens (or OIDs) of a device, it issues a UDP get request to port 161 containing the OIDs to the IP of the device. The device then replies with a "Get" response containing a status and the OID pairs

requested.

To access a device's "private" data, the OID refers to the private section of the MIB hierarchy and a manufacturer's code follows creating an extensible tree for the "custom" OID/value pairs.

Regarding traps, there are seven types of generic traps: cold start, warm start, link down, link up, authentication failure, EGP neighbor loss, and enterprise specific. An enterprise specific trap is sent with a unique specific trap identifier. The trap can include OID pairs to further signify the trap event. This makes it easier to identify the trap with different SNMP applications.

Steven Gettel - software developer

Typical Owl or Finch MIB - Partial Listing

(Current MIB can be downloaded from actual unit)

OID name	OID	access	type
General System Information OID's			
sysDescr	1.3.6.1.2.1.1.1	read only	DisplayString
sysObjectID	1.3.6.1.2.1.1.2	read only	OBJECT IDENTIFIER
sysUpTime	1.3.6.1.2.1.1.3	read only	TimeTicks
sysContact	1.3.6.1.2.1.1.4	read only	DisplayString
sysName	1.3.6.1.2.1.1.5	read only	DisplayString
sysLocation	1.3.6.1.2.1.1.6	read only	DisplayString
sysServices	1.3.6.1.2.1.1.7	read only	INTEGER(1..100)

OID name	OID	access	type
General Device Information OID's			
deviceInfo	1.3.6.1.4.1.17373.2.1		other
productTitle	1.3.6.1.4.1.17373.2.1.1	read only	DisplayString
productVersion	1.3.6.1.4.1.17373.2.1.2	read only	DisplayString
productFriendlyName	1.3.6.1.4.1.17373.2.1.3	read only	DisplayString
productMacAddress	1.3.6.1.4.1.17373.2.1.4	read only	DisplayString
productUrl	1.3.6.1.4.1.17373.2.1.5	read only	DisplayString
alarmTripType	1.3.6.1.4.1.17373.2.1.6	read only	INTEGER(0..9)

Climate Sensor OID's

OID name	OID	access	type
climateTable	1.3.6.1.4.1.17373.2.2	no access	other
climateEntry	1.3.6.1.4.1.17373.2.2.1	no access	other
climateIndex	1.3.6.1.4.1.17373.2.2.1.1	no access	INTEGER(1..1)
climateSerial	1.3.6.1.4.1.17373.2.2.1.2	read only	DisplayString
climateName	1.3.6.1.4.1.17373.2.2.1.3	read only	DisplayString
climateAvail	1.3.6.1.4.1.17373.2.2.1.4	read only	TruthValue
climateTempC	1.3.6.1.4.1.17373.2.2.1.5	read only	INTEGER(-50..100)
climateHumidity	1.3.6.1.4.1.17373.2.2.1.6	read only	INTEGER(0..100)
climateAirflow	1.3.6.1.4.1.17373.2.2.1.7	read only	INTEGER(0..100)
climateLight	1.3.6.1.4.1.17373.2.2.1.8	read only	INTEGER(0..100)
climateSound	1.3.6.1.4.1.17373.2.2.1.9	read only	INTEGER(0..100)
climateIO1	1.3.6.1.4.1.17373.2.2.1.10	read only	INTEGER(0..100)
climateIO2	1.3.6.1.4.1.17373.2.2.1.11	read only	INTEGER(0..100)
climateIO3	1.3.6.1.4.1.17373.2.2.1.12	read only	INTEGER(0..100)

Power Monitor OID's

OID name	OID	access	type
powerMonitorTable	1.3.6.1.4.1.17373.2.3	no access	other
powerMonitorEntry	1.3.6.1.4.1.17373.2.3.1	no access	other
powMonIndex	1.3.6.1.4.1.17373.2.3.1.1	no access	INTEGER(1..100)
powMonSerial	1.3.6.1.4.1.17373.2.3.1.2	read only	DisplayString
powMonName	1.3.6.1.4.1.17373.2.3.1.3	read only	DisplayString
powMonAvail	1.3.6.1.4.1.17373.2.3.1.4	read only	TruthValue
powMonKWattHrs	1.3.6.1.4.1.17373.2.3.1.5	read only	Unsigned32
powMonVolts	1.3.6.1.4.1.17373.2.3.1.6	read only	Unsigned32
powMonVoltMax	1.3.6.1.4.1.17373.2.3.1.7	read only	Unsigned32
powMonVoltMin	1.3.6.1.4.1.17373.2.3.1.8	read only	Unsigned32

powMonVoltPk	1.3.6.1.4.1.17373.2.3.1.9	read only	Unsigned32
powMonAmpsX10	1.3.6.1.4.1.17373.2.3.1.10	read only	Unsigned32
powMonRealPow	1.3.6.1.4.1.17373.2.3.1.11	read only	Unsigned32
powMonAppPow	1.3.6.1.4.1.17373.2.3.1.12	read only	Unsigned32
powMonPwrFact	1.3.6.1.4.1.17373.2.3.1.13	read only	INTEGER(0..100)
powMonOutlet1	1.3.6.1.4.1.17373.2.3.1.14	read only	INTEGER(0..100)
powMonOutlet2	1.3.6.1.4.1.17373.2.3.1.15	read only	INTEGER(0..100)

External Temp Sensor OID’s

tempSensorTable	1.3.6.1.4.1.17373.2.4	no access	other
tempSensorEntry	1.3.6.1.4.1.17373.2.4.1	no access	other
tempSensorIndex	1.3.6.1.4.1.17373.2.4.1.1	no access	INTEGER(1..100)
tempSensorSerial	1.3.6.1.4.1.17373.2.4.1.2	read only	DisplayString
tempSensorName	1.3.6.1.4.1.17373.2.4.1.3	read only	DisplayString
tempSensorAvail	1.3.6.1.4.1.17373.2.4.1.4	read only	TruthValue
tempSensorTempC	1.3.6.1.4.1.17373.2.4.1.5	read only	INTEGER(-50..100)

External Temp/Airflow/Humidity OID’s (RTAF or RTAFH)

airFlowSensorTable	1.3.6.1.4.1.17373.2.5	no access	other
airFlowSensorEntry	1.3.6.1.4.1.17373.2.5.1	no access	other
airFlowSensorIndex	1.3.6.1.4.1.17373.2.5.1.1	no access	INTEGER(1..100)
airFlowSensorSerial	1.3.6.1.4.1.17373.2.5.1.2	read only	DisplayString
airFlowSensorName	1.3.6.1.4.1.17373.2.5.1.3	read only	DisplayString
airFlowSensorAvail	1.3.6.1.4.1.17373.2.5.1.4	read only	TruthValue
airFlowSensorFlow	1.3.6.1.4.1.17373.2.5.1.5	read only	INTEGER(0..100)
airFlowSensorTempC	1.3.6.1.4.1.17373.2.5.1.6	read only	INTEGER(-50..100)
airFlowSensorHumidity	1.3.6.1.4.1.17373.2.5.1.7	read only	INTEGER(0..100)

Power Only OID’s (RSP)

powerOnlyTable	1.3.6.1.4.1.17373.2.6	no access	other
powerOnlyEntry	1.3.6.1.4.1.17373.2.6.1	no access	other
powerIndex	1.3.6.1.4.1.17373.2.6.1.1	no access	INTEGER(1..100)
powerSerial	1.3.6.1.4.1.17373.2.6.1.2	read only	DisplayString
powerName	1.3.6.1.4.1.17373.2.6.1.3	read only	DisplayString
powerAvail	1.3.6.1.4.1.17373.2.6.1.4	read only	TruthValue
powerVolts	1.3.6.1.4.1.17373.2.6.1.5	read only	Unsigned32
powerAmps	1.3.6.1.4.1.17373.2.6.1.6	read only	Unsigned32
powerRealPow	1.3.6.1.4.1.17373.2.6.1.7	read only	Unsigned32
powerAppPow	1.3.6.1.4.1.17373.2.6.1.8	read only	Unsigned32
powerPwrFactor	1.3.6.1.4.1.17373.2.6.1.9	read only	INTEGER(0..100)

External Door Sensor OID’s

doorSensorTable	1.3.6.1.4.1.17373.2.7	no access	other
doorSensorEntry	1.3.6.1.4.1.17373.2.7.1	no access	other
doorSensorIndex	1.3.6.1.4.1.17373.2.7.1.1	no access	INTEGER(1..100)
doorSensorSerial	1.3.6.1.4.1.17373.2.7.1.2	read only	DisplayString
doorSensorName	1.3.6.1.4.1.17373.2.7.1.3	read only	DisplayString
doorSensorAvail	1.3.6.1.4.1.17373.2.7.1.4	read only	TruthValue
doorSensorStatus	1.3.6.1.4.1.17373.2.7.1.5	read only	INTEGER(0..100)

Finch Software Specifications

Code Functions

Protocols

User Defined Features

Finch Software Specifications

v1.1 July 18, 2007

This specification gives a list of the software features contained in the basic Finch configuration. All device access is through a non-encrypted web page. The device firmware can be upgraded by the end user.

This Finch can be be re-branded with a reseller's logo and text. The text "(OEM)" means that it is possible to change this value by updating files in the Flash File System. This means the values are permanent from the end user's point of view, but they are changeable without changing the firmware.

Web User Interface

Header

- Device type (eg,"MicroGoose") (™)
  - Version number of the firmware (h4)
  - Company logo (OEM)
  - Device friendly name
  - Device current IP address

Footer

- Name of the company with copyright and home page link (OEM)
- Technical support contact info: email, phone (OEM)

Topbar

- Menu of all available pages
- Current page is clearly marked in the menu
- Link to view sensor data in XML format
- Link to download the MIB for the device

Live Sensor Readings Page

- Current sensor values
- Values are displayed with proper units of measurement
- Values are highlighted in red if they are in an alarmed state

Device Configuration Page

- Device-wide friendly name

Configuration Page

- Network Configuration
  - Manual
    - User must set IP address, net mask, and gateway
- SNMP
  - SNMP community string for GET
  - SNMP community string for TRAP

SNMP port to listen on for GET
One IP Addresses to send TRAP on alarm
Alarm Configuration
The device has a set maximum number of possible alarms
Each alarm has the following properties:
Threshold type (high trip only)
Threshold value (sensor value for tripping alarm)
Action (SNMP only)

Firmware

Upload new firmware

XML Page

Format identical to that of standard products
Must work with console
Must work with the Excel logger

Layout

Uses goose-like layout

FTP

Upload changes to Flash File System

Web Server

Single-threaded server, just like WxGoose

SNMP

Only protocol v1.0 will be supported
Server settings are configurable as defined on the configuration screen
MIB is included in the flash file system so it can be downloaded by the user without CD's or access to another web site
The server handles GET and GET-NEXT
RFC OID values must be present but most can be faked for now. Settings that must be there in v1.0 are:
Manufacturer (OEM)
Support contact info (OEM)
IP address info
Our OID values will include at least:
Device type
Device version number
Device friendly name

Sensors

Power monitoring via CT's
"current sensor values" means "current as of the last successful sensor sweep."
Done by the CT read interrupt. Other threads can access only the last-known-good values in a thread-safe manner

Alarms

Alarms are configured from a web page (described elsewhere)
Each alarm has high-trip only
GNMP only

When an alarm is triggered:
Trap is sent; var-binds:
Current sensor value

Firmware Upgrades

Firmware is updatable by the enduser
Upgrade system detects invalid firmware – bad headers, bad checksums, too large, etc.
If the firmware isn't fully uploaded or is invalid, the device continues with the existing firmware, even after a reboot

Flash I/O Manager

The manager controls read and write access to the flash chips.

Settings Manager

Manages loading and saving system-wide settings, both user preferences and internal state that we need to preserve across reboot
Provides defaults when no settings are present
Ability to "Reset to defaults"

Hard Reset

There is a user-accessible reset button
To reset the device, hold the button.
The "reset" action is specifically:
IP address, net-mask, SNMP settings, channel names, alarm values, gateway can all be reset to factory defaults

Demo Mode

Web page displays current mode: demo vs. normal

Owl Software Specifications

Code Functions

Protocols

User Defined Features

Owl Software Specifications

v1.1 July 18, 2007

This specification give a list of the software features contained in the basic Owl configuration. All device access is through a non-encrypted web page. The device firmware can be upgraded by the enduser.

This Owl can be be re-branded with a reseller's logo and text. The text "(OEM)" means that it is possible to change this value by updating files in the Flash File System. This means the values are permanent from the enduser's point of view, but they are changeable without changing the firmware.

Web User Interface

Header

Device type (eg,"MicroGoose") (™)

Version number of the firmware (h4)
Company logo (OEM)
Device friendly name
Device current IP address
General status indicator:
Bold, green text when everything is fine (h5)
Bold, red text when at least one alarm is triggered

Footer

Name of the company with copyright and home page link (OEM)
Technical support contact info: email, phone (OEM)
Device location
Device administrator name, phone, email

Sidebar

Menu of all available pages
Current page is clearly marked in the menu
Link to view sensor data in XML format
Link to download the MIB for the device

Live sensor readings Page

Current sensor values
Entire page refreshes automatically every 60 seconds
Values are displayed with proper units of measurement
Values are highlighted in red if they are in an alarmed state

Device Configuration Page

Device-wide friendly name
Temperature unit can be changed

Device Administration
- Full Name (optional)
- Email Address (optional)
- Phone Number (optional)
- Device Location (optional)

Alarm Configuration Page

The device has a set maximum number of possible alarms

Each alarm has the following properties:
- Device/sensor to monitor
- Threshold type (high or low)
- Threshold value (sensor value for tripping the alarm)
- Action (email, SNMP)

The user cannot configure the ordering of alarms but can create/destroy alarms at will up to the maximum number

See the Alarms section below for internals

Configuration Page

Reset to defaults button

Network Configuration
- Manual
  - User must set IP address, net mask, and gateway
- Automatic
  - User can specify DHCP

Domain name to IP resolution
- Fields to enter IP:port for two DNS servers

Realtime Clock
- Fields for user to set local time
- Field for user to set offset from GMT
- Manual setting or sync to SNTP server
- Two SNTP sever addresses: primary, secondary.
- User set synchronization internal

Video Cameras
- Up to four cameras configurable
- Drop-down list of natively-supported camera models, plus one option for "Custom"
- Text box for entering the IP address and port of a supported camera or a standard URL for the "Custom" field
- Individual cameras can be disabled

Email
- IP Address of SMTP server
- Port number of SMTP server (default 25)
- "From" address
- Up to 5 "To" addresses
- IP Address of POP3 server
- Port number of POP3 server (default 110)
- Username for email

Password for email
Button for “send test email” (saves first)
SNMP
SNMP community string for GET
SNMP community string for TRAP
SNMP port to listen on for GET
SNMP port to send TRAP
Two IP Addresses to send TRAP on alarm
Ability to disable SNMP server completely
Button for “send test trap” (saves first)

HTTP

HTTP port number can be set
Flash File System
Form allowing file system uploads

Firmware

Form allowing new firmware uploads

Help Page

A simple help page explaining how things work
No screen shots used
Static content only; served up from flash file system

XML Page

Format identical to that of the goose
Must work with console
Must work with the Excel logger

Layout

Uses contemporary layout

Web Server

Listening port is configurable
Single-threaded server, just like WxGoose
Entire flash file system is always served up by the web server, so OEM’s can potentially insert entire additional files to be served up
Supports “If-Modified-Since” caching for static content

Security

Supports HTTPS/SSL Cypher Suites shown below:
SSL-RSA-RC4-128-MD5
SSL-RSA-RC4-128-SHA
SSL-RSA-3DES-EDE-CBC-SHA
TLS-RSA-AES-128-CBC-SHA
TLS-RSA-AES-256-CBC-SHA
User accounts: admin, view only, control.
Per page read/write privileges assignable to user accounts

SNMP

Only protocol v1.0 will be supported
Server settings are configurable as defined on the configuration screen
MIB is included in the flash file system so it can be downloaded by the

user without CD's or access to another web site
A single thread is used for the SNMP server
The server handles GET and GET-NEXT
Traps are sent by the alarms thread
RFC OID values must be present but most can be faked for now. Settings that must be there in v1.0 are:
Manufacturer (OEM)
Support contact info (OEM)
IP address info
Our OID values will include at least:
Device type
Device version number
Device friendly name
Device location
Device administrator information
Current temperature, C
Current temperature, F
Current humidity

Sensors

Temperature, humidity, on-board
"current sensor values" means "current as of the last successful sensor sweep." This sweep happens once every 5 seconds or less often if this activity is pre-empted by a large amount of network activity. PIC delivers T and H in our usual way
Each time the values are successfully updated, the time-of-last-update is modified
The sensor-sweeper is in the device manager thread. Other threads can access only the last-known-good values in a thread-safe manner
Dallas 1-Wire protocol is supported for remote sensors

Alarms

Alarms are configured from a web page (described elsewhere)
Each alarm has high-trip or low-trip threshold
Sending emails and SNMP traps can be separately enabled

Syslogd

These alerts are always enabled
These messages won't actually be sent unless the syslogd subsystem is configured by the user
Only one message per alarm set/clear
Format includes friendly names, current values, threshold values, and whether this is "set" or "clear" alarm
When an alarm is triggered:
Email is sent
Trap is sent; var-binds:
Base device ID

Base device friendly name
Remote device ID (same as base for internal sensors)
Remote device friendly name (same as base for internals)
Current sensor value
Syslogd message is sent
When an alarm is cleared:
Email is sent
Syslogd message is sent
Hysteresis so alarms are not repeatedly triggered and cleared if sensor value vacillates over a threshold value
Email subject lines are brief so can be viewed easily on cell phones
Email content is as brief as possible for easy viewing on cell phones
Alert messages generally use friendly names wherever possible
Alert messages include unique device ID wherever possible
Alarms states are evaluated at the end of every sensor sweep as part of the sensor sweep thread.
If alerts are necessary, a message is placed in a queue and a separate thread handles sending the alert.(e.g. e-mail/SMNP trap)

syslogd

The internal logging facility can send logging messages to the serial debugging port

Firmware Upgrades

Firmware must be updatable in the field
Upgrade system must detect invalid firmware – bad headers, bad checksums, too large, etc. The user must not be able to kill the device by uploading a file
If the firmware wasn't fully uploaded or is invalid, the device must continue with the existing firmware, even after a reboot

Flash I/O Manager

The manager controls read and write access to the flash chips.
Provides "continuous buffer" API to the rest of the code, so no other subsystem needs to understand anything about sectors or other flash-chip-specific information
Auto CRC for all read and write operations, so other subsystems can assume that a non-error operation has been double-checked.
Automatically ignores trivial writes (a sector-write where no bytes changed) increasing both speed and flash chip lifetime. Other subsystems can therefore

Settings Manager

Manages loading and saving system-wide settings, both user preferences and internal state that we need to preserve across reboot
Provides defaults when no settings are present
Ability to "Reset to defaults"
All settings accessible using get/set API; no setting is accessed directly

by other subsystems

Setting access is thread-safe, and this constraint is managed completely by the settings manager so other threads can call any API without worrying about threads.

Error messages are provided in the case that setting a variable was illegal

These error messages can be propagated up to the user in situations like the web-based user interface

Error messages include things like "Text too long for this variable" or "IP address required" or other formatting issues.

System Upgrade Management

Systems get upgraded to new versions; settings must always be preserved in that case. Sometimes previous settings will affect the default values during the upgrade.

Existing settings must stay where they are in memory so that system down-grades will still preserve those settings that existed at that time. This may mean writing a setting in two places if we needed to change how the setting was represented.

All system upgrade paths will be supported, but not all downgrades must be. However we must specifically document when we're not supporting downgrading through a certain version so customers are aware.

OEM Settings

Items marked (OEM) in this spec can be altered using the "flash file system." This means these items can be updated at authorized sites without changing the firmware.

These settings will be in a plain-text file with one setting per line in "key=value" format. All data will be pulled from this file.

End users are not able to update these items, so for them these things will appear to just be built into the unit.

New flash file systems can be uploaded using a secret web page.

Secret because there are no links to it and we only tell OEM's about it

The page contains a form that allows the user to upload a new flash file system image

The upload mechanism makes a small attempt to validate the input – checking that the header is correct – so that if the file is obviously not really a valid flash file system it will be rejected and the existing file system will remain

If the file system image appears correct, but later in the file there is an error or if the end user cancels the operation, the device might be left without a flash file system at all. The operator must re-upload a valid file system to remedy

This "secret page" must operate completely independently from the flash file system since it must be operational even

without a file system. This means a custom page handler just for the page without styles, logos, and so on.

Hard Reset

There will be a user-accessible reset button (paperclip or easy-push?)
To reset the device, hold the button for 5 seconds.
The "reset" action is specifically:
IP address, net-mask, gateway reset to factory defaults
Web server port reset to 80

Demo Mode

By use of the user/admin password user functions can be set. In the WeatherGoose this was called "Demo Mode."

Graphing

Typical number of variables: 12 typical
Number of colors: 23
Scaling: automatic
Selection of graphed values:
Based on logged data
View trends in sensor data
Plot multiple sensors (data) per graph

Logs

Store sensor data in non-volatile memory
Store up to days/weeks/months of data depending on number of sensors and user configuration
Download data values in CVS format for analysis by other programs

Real Time Clock

Separate RTC chip interfaced to LPC processor using I2C bus.
RTC chip has on-board backup power source (super capacitor) so that accurate time is kept over a period of days without an external power source.
Functions to provide time data strings formatted for display in web page (local time) and per RFC 2616 for inclusion in HTTP response headers (GMT).
Local time/date and offset from GMT are entered through the Config web page.

Parts to be further specified:

FTP
Telnet
BOOTP
SNMP "SET"
daily affirmation
special WAP page

MicroGoose - A Sample Product

Low-Cost Computer Room Climate Monitor

Owl Processor

Power-over-Ethernet

Temperature & Humidity Sensors

Built in Six Weeks

Climate Monitor for Server Rooms with PoE or Cabinets

Worried about hot spots in blade server cabinets or remote computer rooms? Place these tiny MicroGoose monitors where you need them. Get e-mail and SNMP alerts with escalations. Add an optional web camera and see what's going on. A bare-bones version of the best-selling WeatherGoose IT climate monitor, this small monitor fits tight budgets and spaces with PoE and video options.

Sensors are exposed outside the metal case to minimize heating by circuitry.

About the size of a candy bar, the MicroGoose contains temperature and humidity sensors in a metal case. The exposed circuit board contains the temperature and humidity sensor.

Web Accessed, Self-Contained

Temperature, humidity sensors internal

Web accessed (internal web server)

Power-over-Ethernet optional

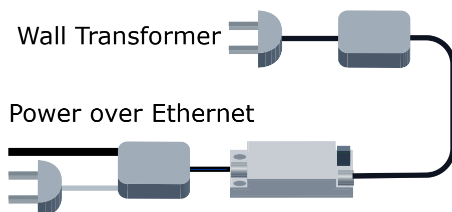
E-mail alarms and escalations

Simple installation, built-in brackets

SNMP, XML pre-installed

Optional video camera

Power-over-Ethernet Option



The MicroGoose can be powered with an external wall transformer power supply or Power-over-Ethernet (PoE). The PoE option requires an extra internal circuit and must be specified at time of ordering.

Typical Applications

The small size and low cost of the unit and the use of a web interface make the MicroGoose useful for:

Blade server cabinets - where hot spots need monitoring

Small server rooms - to detect air conditioning failures early

Data centers - to know temperature and humidity in problem areas.

Note that the MicroGoose can be ordered with the Power-over-Ethernet option.

Internal Sensors

Temperature: -40°F to 140°F, +/-0.5°C

Humidity: 5%- 100%, +/-5%

Video Camera

Up to four cameras supported

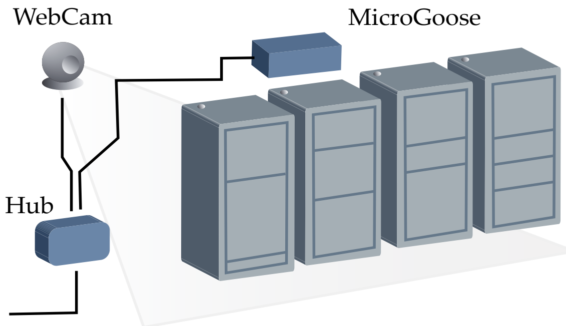
Axis 205, 206 or D-Link 950 (optional)

Specifications

Physical: 4″L x 1.5″H x 1.5″W, 0.5 pounds

Power: 6VDC (supplied wall transformer)

Ethernet: 10 Mbps, RJ-45 receptacle



In this drawing, a MicroGoose and an optional webcam monitor a server room with four cabinets. The IT manager knows the temperature and humidity plus gets e-mails when the web cam detects motion.

Standards: FCC Part 15, 802.3fc (PoE)

Control:

Reset push-button: restores factory settings and factory IP address (192.168.123.123)

Software Features

HTTP - web access

Alarms - high, low values, multiple addresses

SMTP/POP3 - e-mail alerts, POP password

SNMP - MIB with Gets, Traps and Clears

Paging - e-mail to pager proxy

XML - all values exposed and meta-tagged

Console - multiple MiniGoose viewer available, with log aggregation, and thumbnail camera views (optional).

Internal Board Heating

The electronics generate a small of amount of heat which can heat the temperature sensor. Bench tests show a typical internal heating amount of 3°F in still air. The temperature and humidity sensors are mounted externally on a thin fiberglass board to

The simple home page shows the temperature, humidity and thumbnails of up to four video cameras. The cameras can be configured to send an e-mail upon detection of motion.

minimize this internal heating error.

If the MicroGoose is exposed to a 25 cfm airflow, which is typical in computer rooms and cabinets, the internal heat is dispersed by the airflow reducing the heating offset error to less than 2°F.

The heating error is additive over a wide range. If the actual room temperature is 70°F the MicroGoose shows 72°F, typically. At an actual room temperature of 90°F, the MicroGoose indicates 92°F. The actual amount of heating error depends on the amount of airflow around the unit airflow.

To restore the factory set address and defaults press the small internal reset switch on the side of the metal case of the MicroGoose for two seconds.

Model Number:

WxGoos-5 MicroGoose (includes a wall-mounted power supply)

WxGoos-5P Same as above plus an internal Power-over-Ethernet adaptor.

Data Input and Output Methods

Analog Data Input

Analog Data Output

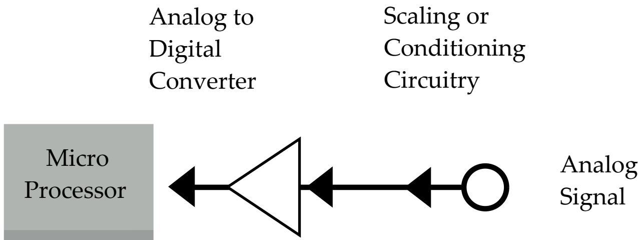
Digital Data Input

Digital Data Output

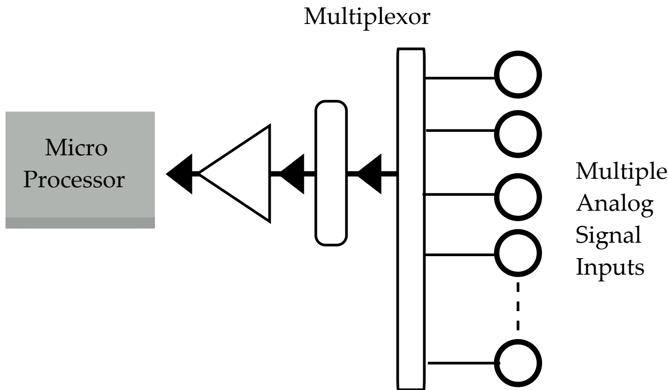
Analog Data Inputs (Typical)

0 - 5 VDC
0 - 10 VDC
4 - 20 ma
0 - 240 VAC

Analog Data Inputs - Single source



Analog Data Inputs - Multiple Sources



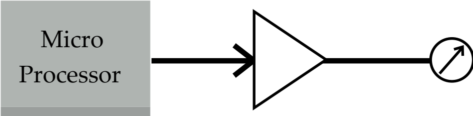
Either single or multiple analog inputs can be used. If multiple inputs are required, a multiplexor circuit can share one analog to digital converter among dozens of input signals.

Analog Data Output - Basic and Amplified

0 - 5 VDC
0 - 10 VDC
4 - 20 ma
0 - 240 VAC

Analog Data Output - Basic voltage output

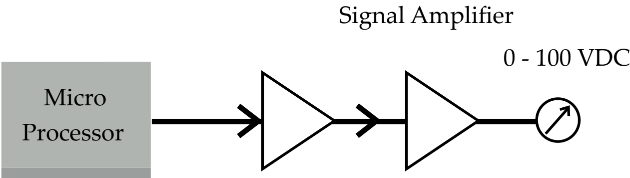
Digital to Analog Output:
0 - 5 VDC typical



Boosted Analog Output

Signal Amplifier

0 - 100 VDC



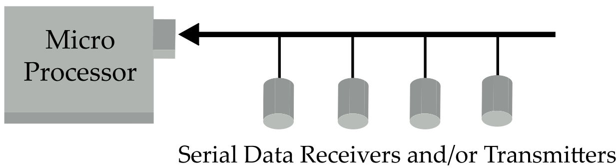
Voltage or current output can also be created in a variety of formats. The native output of the embedded processor is either 3 or 5 volts. If this is not the voltage needed or you need a current-based signal, additional circuits are needed. The output can drive meters, lamps or other devices.

Serial Data Inputs (Typical)

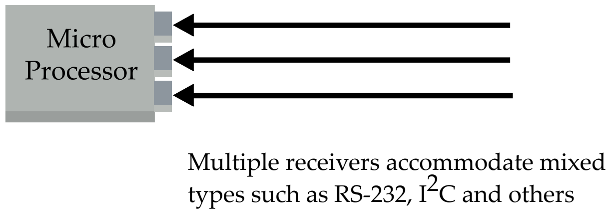
RS-232
RS-485
I^2C
1-Wire
Modbus

Single Serial Data Inputs (Typical)

Multiple Data Inputs - RS-485



Mixed Data Inputs - Multiple data formats



Many devices generate serial data which can be used by a variety of circuits. Serial data is usually easy to convert into web pages since the data formats normally conform to widely used standards.

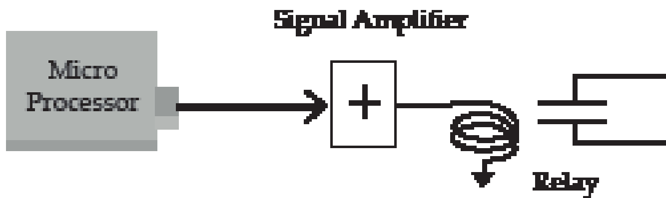
Digital Data Output (Control)– Low and High Level

TTL
0 - 5 VDC (on or off)
Relay control
Motor control

Logic Level Output – Basic voltage output

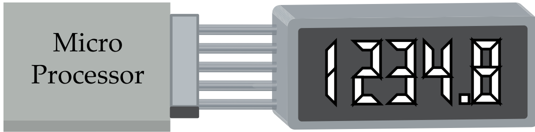
Native Digital Output
0 – 5 VDC typical

Relay Output

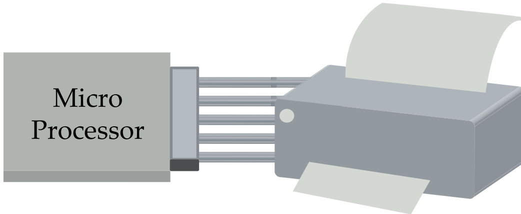


Many devices have an on-off command. A typical use of this is the control of a relay which, in turn, controls a remote device such as a motor or light. A web enabled device could control one or hundreds of relays. A relay can be specifically sized to handle the amount of control current needed.

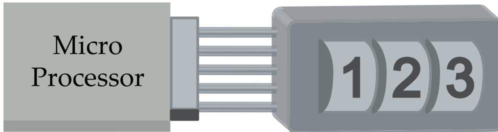
Digital Data Displays - LED displays and Printers



Parallel Interface LED or LCD display



Parallel Interface Printer



Parallel Interface Thumbwheel Value Setter

Some devices require a parallel format of data. Typical of these devices are parallel printers, numeric displays, and some types of industrial controllers. Typical devices use eight data lines plus one for control and error checking. Many legacy devices use parallel data transmission and reception.