

The GNATs Developer's Manual

The **G**orgia **T**ech **N**etwork for **A**utonomous **T**asks

Tucker Balch, Victor Bigio, Eric R. Dodson, Arya Irani,
Keith J. O'Hara, Daniel B. Walker
Georgia Institute of Technology
<http://www.cc.gatech.edu/~borg/gnats/>

May 1, 2005

Abstract

We are interested in the application of low-cost, pervasively distributed network nodes to support cooperative multirobot tasks. In this work we consider a heterogeneous system composed of small, embedded, immobile sensor-less communication nodes and larger mobile robots equipped with sensors and manipulators. The embedded network serves as a pervasive communication and computation fabric, while the mobile robots provide sensing and actuation. In our work the embedded nodes provide only modest computation and communication for the team.

The GNATs are small, immobile, sensor-less communication nodes. This document introduces robotics researchers and other technically inclined people to the GNATs. The following sections describe various aspects of the GNATs operation and communication, and the programming environment. This document, in conjunction with the software supplied (API and sample applications), will assist the reader in developing applications for the GNATs.

Contents

1	Components and Functional Overview	4
2	Controlling the Hardware	5
2.0.1	PIC PortA - Gnat hardware version 2	7
2.0.2	PIC PortB - Gnat hardware version 2	7
2.0.3	PIC PortA - Gnat hardware version 1	7
2.0.4	PIC PortB - Gnat hardware version 1	8
2.1	Controlling Power	8
2.2	D/A Converter Signals and Programming	8
2.3	Setting Power Levels	9
2.4	Controlling Outputs	9
2.4.1	LEDs	10
2.4.2	Infrared Emitters	10
2.5	Inputs	11
2.5.1	Infrared Receivers	11
2.5.2	Button Input	12
2.5.3	Battery Voltage Measurement	12
3	GNAT-to-GNAT Communication	12
3.1	Receiver Characteristics	13
3.2	Emitter Power Levels	13
3.3	Protocol	15
3.3.1	Data transmission	15
3.3.2	Data Receiving	17
3.4	Range	17
3.4.1	Power Settings	17
3.4.2	Orientation	18
4	Programming, API and Sample Applications	20
4.1	Programming Language(s)	20
4.2	Programming Information	20
4.2.1	Initializing the system	20
4.2.2	Data Structures	21
4.2.3	Transmitting Data	21
4.2.4	Receiving Data	21
4.2.5	Still To Do	22
4.3	GNAT API and Library	23
4.3.1	GnatBaseAPIConfig.h	24
4.3.2	GnatBaseAPI.h	24
4.3.3	GnatCommAPI.h	28
4.3.4	GnatLib.h and GnatLib.c	33
4.4	Sample Applications	36
4.4.1	sample.c	36
4.4.2	SendReceive_v2.c	36

4.4.3	PathPlan_v2.c	36
4.5	Programming Tips for the PIC Microprocessor	37
4.5.1	Compiler Case Sensitivity	37
4.5.2	Program Counter, Function Call Stack, and Lack of Function Pointers	37
4.5.3	CCS C function bugs	38
4.5.4	Unused Functions	38
4.5.5	Interrupt Servicing and saving Global Interrupt state	38
5	GNAT Development Environment	40
5.1	Software and Hardware Required	40
5.2	Setting Up the Development Environment	41
5.3	Create a new Project	42
5.4	Compiling and Downloading a Project	43
A	Signals Organized by Function	44
A.1	Gnat hardware version 2	44
A.2	Gnat hardware version 1	45
B	Built-in commands	46
C	sample.c	47
D	GnatBaseAPIConfig.h	49
E	Data EEPROM Memory Map	54
F	IR Power Test and Power Model	55
G	GNAT Hardware Version 2 Circuit Diagram	58
H	GNAT Hardware Version 1 Circuit Diagram	60
I	List of Parts	61

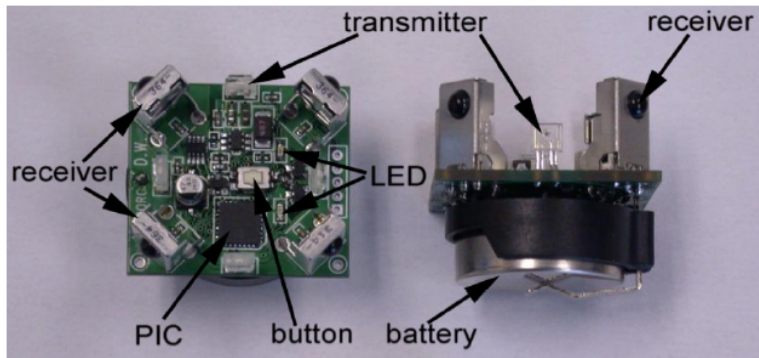


Figure 1: A photo of two GNATs with the components labelled.

1 Components and Functional Overview

A GNAT is a compact device composed of a circuit board with the following components (as well as the necessary support circuitry). See the photos of a GNAT (Figure 1). The diagram of a GNAT (Figure 2) additionally details the numbering scheme used for the components.

Qty	Item
1	PIC 16F87 Processor
1	Battery Holder
1	Digital-to-Analog Converter
1	Power Regulator
4	Infrared Emitters
4	Infrared Receivers
2	LEDs (Green and Red)
1	Button
	Programming connector

The relationship between the various components and control signalling is shown in the GNAT Functional Diagram ((Figure 3), and described here:

- The PIC16F87 is a low power general purpose microprocessor.
- The Battery powers all of the circuits of the board.
- The output of the D/A Converter (DAC) is used to control the Power Regulator, which provides power to the IR Emitters (IROUT1 – IROUT4) and the LEDs (LOUT1 – LOUT2).
- \overline{TCNTRL} turns on/off power to the D/A Converter and the Power Regulator.

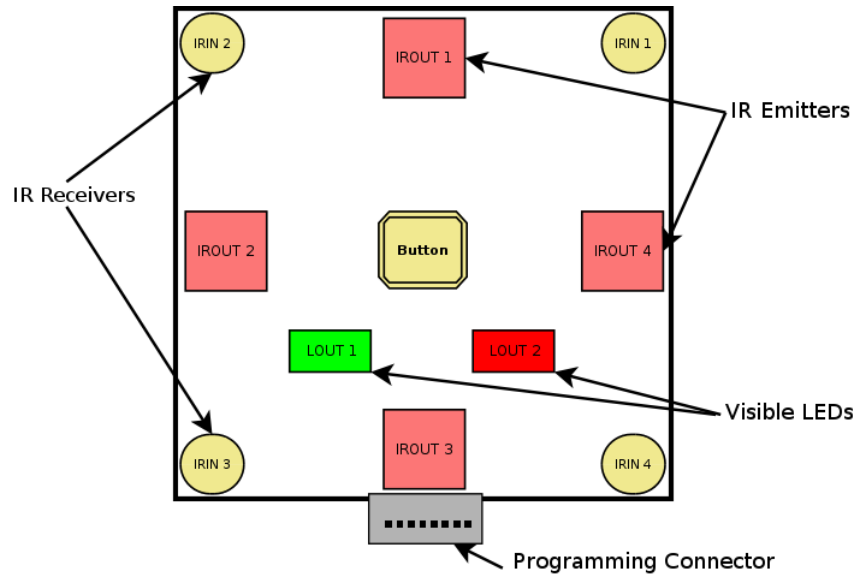


Figure 2: A diagram of the GNAT with the components numbered.

- The LEDs (LOUT1 – LOUT2) emit visible light (green and red). Power is determined by the setting of the DAC. Each LED (on/off) is individually controlled by the processor.
- The IR Emitters (IROUT1 – IROUT4) emit infrared light. Power is determined by the setting of the DAC. Each emitter (on/off) is individually controlled by the processor.
- \overline{RCNTRL} turns on/off power to the IR Receivers
- The IR Receivers ($\overline{IRIN1}$ – $\overline{IRIN4}$) signal the presence of Infrared at the base frequency (40kHz).
- The Button (\overline{BUTT}) provides input to the processor

2 Controlling the Hardware

At the heart of the system is a Microchip PIC 16F87 Microcontroller. Output from the 16F87 is used to control:

- Power on/off to the DAC, the Power Regulator, and the IR Receivers
- Power level for the IR Emitters and the LEDs via setting the DAC
- Individually turning on/off the IR Emitters and the LEDs

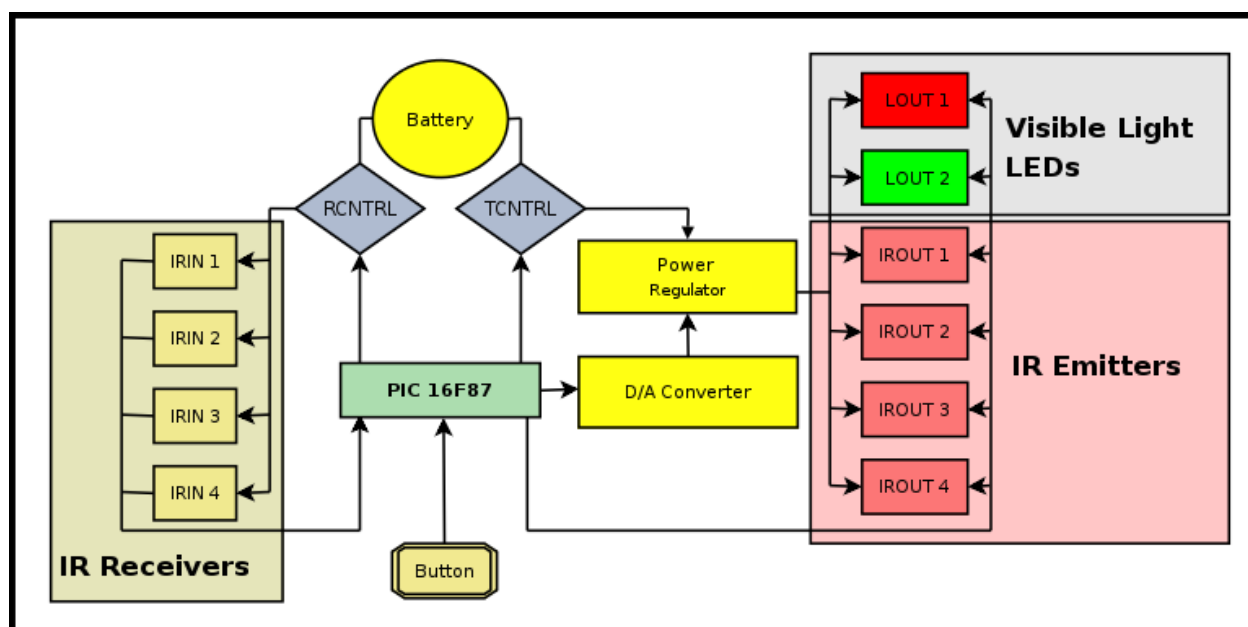


Figure 3: Functional Diagram of the GNAT Board

The following signals are provided as input to the 16F87:

- The presence of modulated infrared
- The button being pressed/released
- Voltage monitoring information

The data sheet for the PIC 16F87 Microcontroller can be found at:
<http://ww1.microchip.com/downloads/en/DeviceDoc/30487b.pdf>

2.0.1 PIC PortA - Gnat hardware version 2

Pin	Signal	In/Out	Description
0	DADIN	Output	D/A Data In
1	\overline{TCNTRL}	Output	D/A Converter On/Off
2	LOUT1	Output	LED Out 1 - Red
3	LOUT2	Output	LED Out 1 - Green
4	DACLK	Output	D/A Clock Input
5	\overline{BUTT}	Input	Button Input
6	\overline{RCNTRL}	Output	IR Receivers On/Off
7	\overline{DACS}	Output	D/A Sync

2.0.2 PIC PortB - Gnat hardware version 2

Pin	Signal	In/Out	Description
0	IROUT4	Output	IR Emitter 4
1	IROUT1	Output	IR Emitter 1
2	IROUT2	Output	IR Emitter 2
3	IROUT3	Output	IR Emitter 3
4	$\overline{IRIN2}$	Input	IR Receiver 2
5	$\overline{IRIN1}$	Input	IR Receiver 1
6	$\overline{IRIN4}$	Input	IR Receiver 4
7	$\overline{IRIN3}$	Input	IR Receiver 3

2.0.3 PIC PortA - Gnat hardware version 1

Pin	Signal	In/Out	Description
0	DADIN	Output	D/A Data In
1	IROUT4	Output	IR Emitter #4
2	LOUT1	Output	LED Out 1 - Red
3	LOUT2	Output	LED Out 1 - Green
4	DACLK	Output	D/A Clock Input
5	\overline{BUTT}	Input	Button Input
6	\overline{DACS}	Output	D/A Sync
7	\overline{RCNTRL}	Output	IR Receivers On/Off

2.0.4 PIC PortB - Gnat hardware version 1

Pin	Signal	In/Out	Description
0	\overline{TCNTRL}	Output	D/A Converter On/Off
1	IROUT1	Output	IR Emitter 1
2	IROUT2	Output	IR Emitter 2
3	IROUT3	Output	IR Emitter 3
4	$\overline{IRIN4}$	Input	IR Receiver 4
5	$\overline{IRIN3}$	Input	IR Receiver 3
6	$\overline{IRIN1}$	Input	IR Receiver 1
7	$\overline{IRIN2}$	Input	IR Receiver 2

2.1 Controlling Power

The GNAT is designed to allow for very efficient use of power, and can be configured so that portions of the circuitry can be turned off for lower power usage.

As shown in the functional diagram (Figure 3), \overline{TCNTRL} is used to turn on/off power to the D/A Converter (DAC) and to the Power Regulator (which supplies power to the IR Emitters and the LEDs. When \overline{TCNTRL} is brought low power is provided to the D/A converter and the Power Regulator. \overline{RCNTRL} is used to turn on/off power to the IR Receivers. When \overline{RCNTRL} is brought low power is provided to the IR Receivers.

If you need to minimize power usage, whenever possible, you should turn power off to the receivers (\overline{RCNTRL} high) and to the DAC and Power Regulator (\overline{TCNTRL} high). You can also use some of the features of the PIC 16F87 to reduce the power usage such as processor speed scaling and the ability to put the processor in a sleep mode.

Signals and pins for Gnat hardware version 2:

Signal	Port	Pin and Description	Processor Pin
\overline{TCNTRL}	A1	Power to D/A Converter and Power Regulator	24
\overline{RCNTRL}	A6	Power to IR Receivers	20

Signals and pins for Gnat hardware version 1:

Signal	Port	Pin and Description	Processor Pin
\overline{TCNTRL}	B0	Power to D/A Converter and Power Regulator	7
\overline{RCNTRL}	A7	Power to IR Receivers	21

2.2 D/A Converter Signals and Programming

The GNAT includes a Texas Instruments DAC7513, an onboard low-power, rail-to-rail output, 12-bit serial input Digital-To-Analog Converter (DAC). Three (3) signals from the PIC 16F87 are used to program the DAC:

- \overline{DACS} - D/A Sync - a level triggered control input (active LOW), the frame synchronization signal for the input data
- DACLK - Serial Clock Input
- DADIN - Serial Data Input.

Signals and pins for Gnat hardware version 2:

Signal	Port	Pin and Description	Processor Pin
\overline{DACS}	A7	D/A Sync	21
DACLK	A4	D/A Clock Input	28
DADIN	A0	D/A Data In	23

Signals and pins for Gnat hardware version 1:

Signal	Port	Pin and Description	Processor Pin
\overline{DACS}	A6	D/A Sync	20
DACLK	A4	D/A Clock Input	28
DADIN	A0	D/A Data In	23

When \overline{DACS} goes low, it enables the input shift register and data. Data, DADIN, is clocked into the 16-bit input shift register on the falling edge of the serial clock input, DACLK. The DAC is updated following the 16th clock cycle unless \overline{DACS} is taken high before this edge in which case the rising edge of DACS acts as an interrupt and the write sequence is ignored.

See the documentation at <http://focus.ti.com/lit/ds/symlink/dac7513.pdf> for more information.

2.3 Setting Power Levels

The output of the DAC is used as input to the Power Regulator, which in turn controls the current to the output devices (IR emitters and LEDs). Different power settings must be set for each of the LEDs (red and green) and for the emitters. The same power level will be used for all of the emitters that are turned on.

Use the GNAT IR Power Chart in Appendix: GNAT Power Plot to select the DAC value corresponding to the current level desired.

Note: the lower the value in the DAC, the higher the power setting.

2.4 Controlling Outputs

Power to the output devices is provided by the Power Regulator, which is in turn controlled by the D/A Converter. It is important to follow some simple

steps in order to prevent damage to the GNAT's circuitry. Whenever generating an output:

1. Make sure all outputs are turned off
2. Set appropriate power level in the DAC, and let the output settle
3. Turn on the specific output signal(s)
4. Delay for the period to leave signal on
5. Turn off the specific output signal(s)

NOTE: Make sure the outputs are all turned off before changing the power settings. Set the power level before turning on an output. Display only one LED at any time. Do not display IR and LEDs simultaneously. Be sure to turn off an output after turning it on.

2.4.1 LEDs

The GNAT has two LEDs, red and green, which can be individually controlled. Each of them has different power requirements, so they should not be turned on simultaneously.

- Red should range from 0x0800 (dim) to 0x0500 (bright)
- Green should range from 0x0600 (dim) to 0x0300 (bright)

Signal	Port and Pin	Description	Processor Pin
LOUT1	A2	LED Out 1 - Red	26
LOUT2	A3	LED Out 2 - Green	27

2.4.2 Infrared Emitters

The GNAT contains four (4) IR Emitters (Panasonic LNA2603F). The emitters are powered by the output of the Power Regulator (at the power level determined by setting the DAC) by setting the corresponding pin(s) (IROUT1 - IROUT4) high on the emitter(s).

Note that the emitters act like a light bulb: when power is applied the emitter emits infrared light and when power is turned off the emitter stops emitting. The emitter does not strobe at any frequency on its own. See the discussion on GNAT-to-GNAT communication below for more information on emitter characteristics.

Signals and pins for Gnat hardware version 2:

Signal	Port and Pin	Description	Processor Pin
IROUT1	B1	IR Emitter 1	8
IROUT2	B2	IR Emitter 2	9
IROUT3	B3	IR Emitter 3	10
IROUT4	B0	IR Emitter 4	7

Signals and pins for Gnat hardware version 1:

Signal	Port and Pin	Description	Processor Pin
IROUT1	B1	IR Emitter 1	8
IROUT2	B2	IR Emitter 2	9
IROUT3	B3	IR Emitter 3	10
IROUT4	A1	IR Emitter 4	24

See the specifications of the LNA2603F at <http://www.semicon.panasonic.co.jp/ds/eng/SHC00027AED.pdf> for more detailed information on the IR Emitters.

2.5 Inputs

The GNAT provides inputs to the PIC 16F87 from the individual receivers and from the Button.

2.5.1 Infrared Receivers

The GNAT contains four (4) Infrared Receivers (Sharp GP1UD28YK 40kHz). Power is provided to the four receivers when \overline{RCNTRL} is brought low. When infrared is absent, the receiver is in a high signal. When the receiver detects infrared modulated at 40kHz, the receiver's signal is brought low. See the discussion on GNAT-to-GNAT communication below for more information on receiver characteristics.

Signals and pins for Gnat hardware version 2:

Signal	Port and Pin	Description	Processor Pin
$\overline{IRIN1}$	B5	IR Receiver 1	13
$\overline{IRIN2}$	B4	IR Receiver 2	12
$\overline{IRIN3}$	B7	IR Receiver 3	16
$\overline{IRIN4}$	B6	IR Receiver 4	15

Signals and pins for Gnat hardware version 1:

Signal	Port and Pin	Description	Processor Pin
$\overline{IRIN1}$	B6	IR Receiver 1	15
$\overline{IRIN2}$	B7	IR Receiver 2	16
$\overline{IRIN3}$	B5	IR Receiver 3	13
$\overline{IRIN4}$	B4	IR Receiver 4	12

See the specifications on the IR Receiver for more detailed information: <http://smaecom1.sharpsec.com/eprise/main/FldrSharp/Techpub/productfocus/publications/opto/infrared/wac\protect\unhbox\voidb@x\kern.06em\vbox{\hrulewidth.>

3em}rmtc/tec\protect\unhbox\voidb@x\kern.06em\vbox{\hrulewidth.3em}datasheet\
 protect\unhbox\voidb@x\kern.06em\vbox{\hrulewidth.3em}gp1ud26xk27\
 protect\unhbox\voidb@x\kern.06em\vbox{\hrulewidth.3em}28\protect\unhbox\
 voidb@x\kern.06em\vbox{\hrulewidth.3em}yk\protect\unhbox\voidb@x\kern.
 06em\vbox{\hrulewidth.3em}series.pdf

2.5.2 Button Input

When the button is pressed the button signal goes low, and stays low until the button is released.

Signal	Port and Pin	Description	Processor Pin
<i>BUTT</i>	A5	Button Input	1

Note: The button can be used in one of two modes, which is determined at the time of programming (#fuses):

- Hardware reset to the processor
- Input to the processor program

See the information on MCLR, PortA pin 5 in the PIC 16F87 Data Sheet: <http://ww1.microchip.com/downloads/en/DeviceDoc/30487b.pdf>

2.5.3 Battery Voltage Measurement

The GNAT provides the ability to measure the voltage level of the battery by using the PIC 16F87's Dual Analog Comparator module. This feature has not yet been implemented.

3 GNAT-to-GNAT Communication

GNATs communicate via transmission (and receipt) of modulated infrared. The receivers are designed to detect the presence of infrared modulated at a base frequency of 40kHz. The IR Emitters generate infrared light when turned on. Modulated infrared is produced when the emitter(s) are turned on and off at the appropriate rate (40kHz requires 12us on and 13us off).

The receivers detect the presence of modulated infrared, at the base frequency (40kHz), and signal that information to the PIC 16F87. In order to transmit/receive data reliably, there must be a known relationship between the output of an emitter and the output of the receiver, with respect to:

Pulse Width: the width of the detected pulse

Period: the time from the start of one bit to the next

3.1 Receiver Characteristics

In an ideal world the pulses generated by an emitter would correspond directly to the information provided by the receiver, both in terms of pulse width and period. This appears to be the case only at very close range and/or at high power settings. Actually, we observed:

1. Receiver pulses may not be the same length as the emitter's output, and are usually shorter
2. Receiver's period can be uneven from pulse to pulse

To get a better understanding of the receiver, we programmed a GNAT to continuously transmit modulated infrared followed by silence. We took measurements using 3 different settings:

Modulated IR (μs)	Silence(μs)
250	2000
400	2000
800	2000

As the distance between the GNATs increased, we observed that the pulses got shorter, and we saw an increase in 'jitter', or a fluctuation of the period from one pulse to the next. At about 5 - 6 meters we started to see a dropping of pulses (i.e. some pulses were not seen at all). See the graph (Figure 4) for a characterization of the pulse width vs. the distance between GNATS.

The duty cycle of the signal also affects the reliable reception of data. The duty cycle is defined as the ratio between the presence of modulated infrared vs. the total time of a data block. The receiver appears to produce more accurate results with a lower duty cycle (the datasheet recommends $< 40\%$).

It also appears that the receivers have a gain setting, which is adjusted according to the presence of both visible and infrared light. A header is recommended in order to allow the receiver(s) to adjust their gain.

Some light fixtures generate infrared modulated at the base frequency, and will affect the receipt of data. Infrared light, like visible light, will reflect off of surrounding surfaces. The GNAT seems more susceptible to noise when no infrared is being transmitted.

3.2 Emitter Power Levels

In order to get actual current values we disconnected an emitter and measured the current flowing from the GNAT circuit. We used 3 different emitters (since the characteristics vary from emitter to emitter), and took readings when the output started (low) and when the output levelled off (high), and then averaged the 3 low values and the 3 high values. The reason there are two values is because, as the emitter heats up, the value of the current increases. Since we

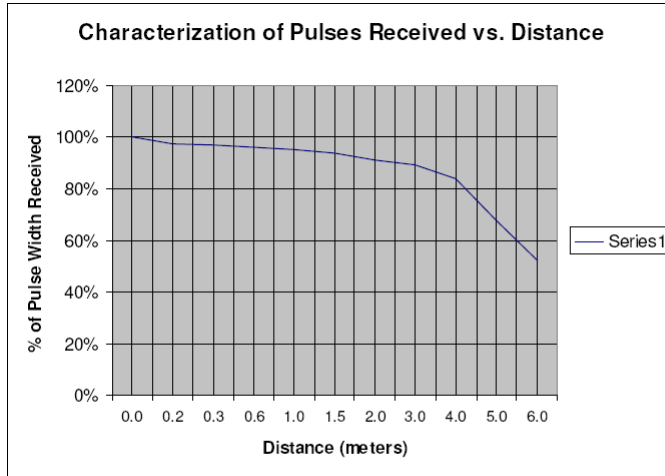


Figure 4: A Characterization of Pulse vs. Distance.

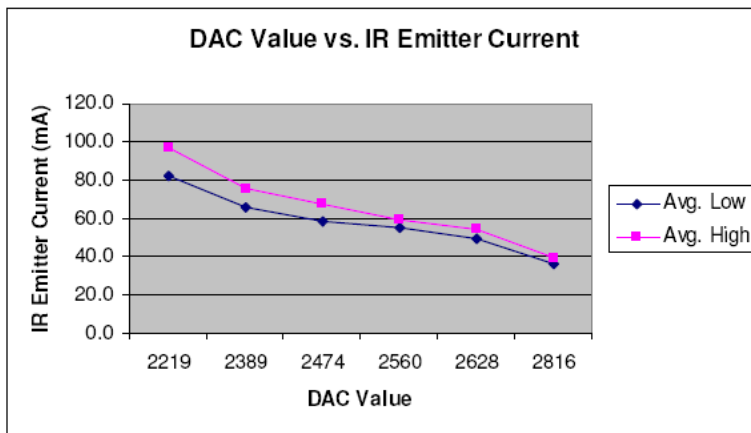


Figure 5: DAC Value vs. IR Emitter Current

are pulsing the output at a very high rate (40kHz), the low values may be more accurate, but the high values were more accurately measured.

See the plot of DAC values vs. the current flowing through the IR Emitter (Figure 5).

3.3 Protocol

We experimented with various protocols, including:

Pulse Width Modulation: the value of a bit ('1' or '0') depends on the width of the pulse, and the period varies.

Manchester: guarantees a rising or falling edge in the middle of the pulse, using a fixed period

Pulse Counting: the number of pulses within a fixed period determines the value of each bit

Pulse Counting appears to give us the best results. Even though it depends on a fixed period, it is relatively immune to a varying pulse width, since we are not looking for information based on the width of the pulse.

3.3.1 Data transmission

Infrared is emitted at 40kHz by turning IR emitters on and off (modulated) using a $25\mu s$ cycle and is controlled through the software using the following timing:

IR emitter on for $12\mu s$

IR emitter off for $13\mu s$

At present all emitters are used for transmitting data. Transmission of a data packet starts with a header, followed by all of the data bits, and ends with a stop bit. The header, data bits and the stop bit all have a fixed period of $2400\mu s$

The timing we are currently using is presented in the following table:

Name	No. of Cycles
EMIT_CYCLES_H	0x30
EMIT_CYCLES_1	0x18
EMIT_CYCLES_0	0x18
REST_CYCLES_H	0x30
REST_CYCLES_1	0x18
REST_CYCLES_0	0x48

The header consists of:

Output	Name	Calculation	Time
Modulated IR	EMIT_CYCLES_H	48 cycles * 25 μ s/cycle	1200 μ s
Silence	REST_CYCLES_H	48 cycles * 25 μ s/cycle	1200 μ s

A zero (0) consists of:

Output	Name	Calculation	Time
Modulated IR	EMIT_CYCLES_0	24 cycles * 25 μ s/cycle	600 μ s
Silence	REST_CYCLES_0	72 cycles * 25 μ s/cycle	1800 μ s

A one (1) consists of:

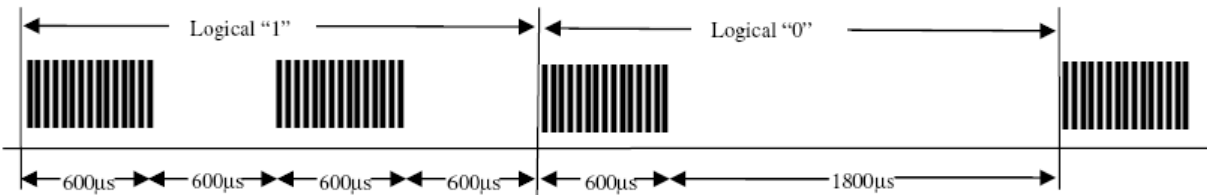
Output	Name	Calculation	Time
Modulated IR	EMIT_CYCLES_1	24 cycles * 25 μ s/cycle	600 μ s
Silence	REST_CYCLES_1	24 cycles * 25 μ s/cycle	600 μ s
Modulated IR	EMIT_CYCLES_1	24 cycles * 25 μ s/cycle	600 μ s
Silence	REST_CYCLES_1	24 cycles * 25 μ s/cycle	600 μ s

The stop bit (same as a zero (0)) consists of:

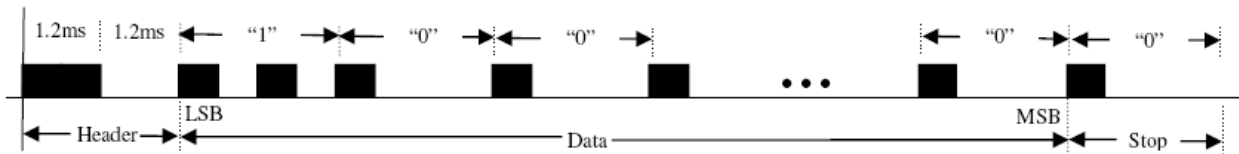
Output	Name	Calculation	Time
Modulated IR	EMIT_CYCLES_0	24 cycles * 25 μ s/cycle	600 μ s
Silence	REST_CYCLES_0	72 cycles * 25 μ s/cycle	1800 μ s

Each period is 2,400 μ s long (96cycles * 25 μ s/cycle). The header is used to alert the gnat that IR is being transmitted and to set the gain on the receiver(s).

Modulation of Infrared



Data Transmission Protocol



3.3.2 Data Receiving

An interrupt is generated on the receiving gnat whenever IR is detected on any of the receivers. The receiving gnat determines which receiver(s) detected IR, and uses these to capture the data being received.

The receiving gnat uses a timer to count the time that IR is being received (a pulse) and also keeps track of the total time for a bit. In general, the software tries to assure that:

- Each pulse that is detected does not exceed a maximum allowed time. There is no minimum limit for the length of a pulse.
- There should be either one or two pulses within the period of a bit.
- The period for each bit (either header, data, or stop) must fall within an acceptable range. This is measured as the time:
 - from the start of the header to the start of the first bit
 - from the start of a data bit to the start of the next data bit
 - from the start of the last bit to the start of the stop bit

The timing we are currently using is the following:

Name	Time
ON_TIME_MAX_H	0x0708
TOTAL_TIME_H	0x0960
ON_TIME_MAX	0x0320
TOTAL_TIME	0x0960

The criteria for valid data bits is detailed in the following table:

Bit	Pulses	Max. Pulse Duration	Period
Header	one	ON_TIME_MAX_H	TOTAL_TIME_H +- 30%
Zero (0)	one	ON_TIME_MAX	TOTAL_TIME +- 30%
One (1)	two	ON_TIME_MAX	TOTAL_TIME +- 30%
Stop bit	one	ON_TIME_MAX	TOTAL_TIME +- 30%

3.4 Range

Though there are many factors that influence the range of transmission, we have done extensive testing under a variety of conditions. The data that follows was gathered using the Pulse Counting protocol and timing described above, and transmitting using all of the emitters simultaneously.

3.4.1 Power Settings

The power setting of the emitter(s) of the sending GNAT will strongly influence the range of transmission. Here are some sample values of DAC settings vs.

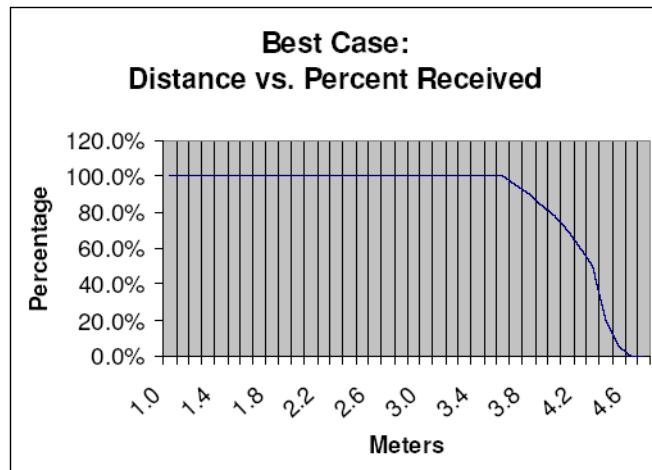
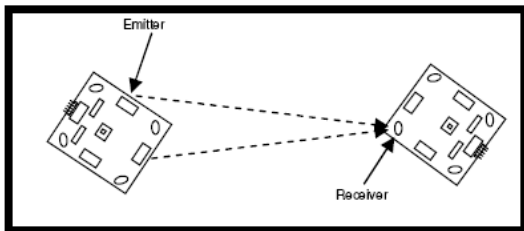
distance (assuming an ideal orientation and average lighting conditions):

DAC Value	Distance
0x0800	3+ meters
0x0A00	2.5+ meters
0x0C00	2+ meters
0x0D00	30+ inches
0x0D80	15+ inches
0x0DB0	11+ inches
0x0E00	5+ inches

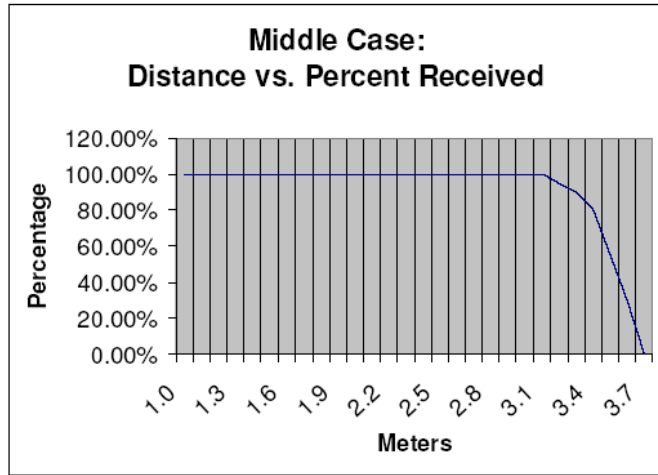
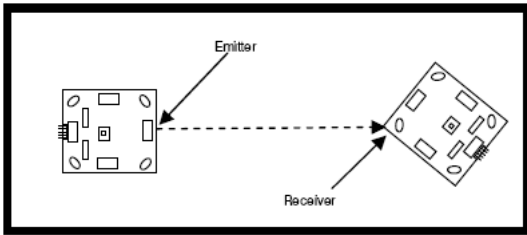
3.4.2 Orientation

The orientation of the sending GNAT to the receiving GNATS will also affect the range. The best orientation for the sender is when 2 emitters are lined up with the receiving GNAT. The best orientation for the receiving GNAT is when a receiver is directly facing the sending GNAT.

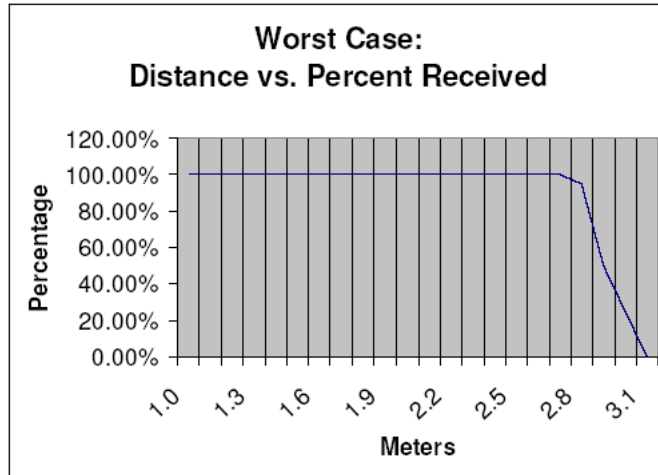
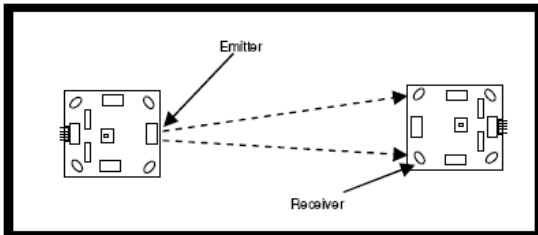
At a power setting of 0x0800, with normal light settings, we achieved the following results displayed in the following figures.



Best Orientation: Two emitters angled to face-on receiver.



Middle Orientation: One emitter face-on to one receiver.



Worst Orientation: One emitter face-on to two angled receivers.

4 Programming, API and Sample Applications

The best way to learn to program any system is to review and/or modify existing applications. Following we provide programming tips and a description of the GNAT API and sample applications.

4.1 Programming Language(s)

Our initial applications for the GNAT were written in assembly code. This provided us with some important insights into programming the PIC 16F87, controlling the GNAT's circuitry, and timing requirements. But, if you've ever programmed extensively in assembly code, you know how cumbersome any development or debugging can be. Once we started working with the 'C' compiler for the PIC, we subsequently converted most of our code to 'C', and have continued using it for new applications.

The compiler provides a variety of built-in commands that makes development for the PIC relatively straightforward. It includes built-in commands (see the Appendix) that facilitate access to some critical features:

- PIC mode of operation (#fuses)
- PIC processor control (speed, port directionality)
- Access to the special registers (PortA, PortB)
- Setup and access to timers
- Interrupt handling

4.2 Programming Information

4.2.1 Initializing the system

The `init()` API function performs the following steps to properly initialize the GNAT circuits and the run-time environment:

- Set port directionality
- Turn off the D/A Converter and Receivers
- Set the oscillator speed (to 4MHz, which results in a 1 MHz instruction speed)
- Turn on the D/A Converter and allow time for the DAC to settle
- Turn on the receivers
- Setup the timers and interrupts

See the full description `init()` in the GNAT Base API section below.

4.2.2 Data Structures

For IR communication the application needs to define an application level data structure to hold the data being received and the data to be sent. Here is a sample of a structure and initializing the send and receive buffers:

```
typedef struct {
    uint16 generation;
    uint8 hop_count;
    uint8 flags;
} app_payload_t;

app_payload_t recv_payload_buf;
app_payload_t send_payload_buf;
```

The size of the payload packet must be multiples of 8 bits (i.e. multiple bytes) and no more than 20 bytes. We have reliably sent (and received) packets of up to 12 bytes total.

4.2.3 Transmitting Data

When preparing to send data, the application may first call `wait_for_channel()` to make sure the channel is clear (so packets have a smaller probability of collision). If the channel has been clear for some number of cycles then this function returns TRUE. See `wait_for_channel()` in the GNAT API.

To send data just store your data in `send_payload_buf`, set the value for transmitter and `IROUT_PWR`, then call:

```
send_capsule (&send_payload_buf, transmitter, IROUT_PWR, sizeof(app_payload_t));
```

See the description of the API for more information.

4.2.4 Receiving Data

An interrupt is generated when there is a change in PortB pins 4 – 7, the IR Emitters. The interrupt handler (`#int_rb`) is called immediately. Interrupts are disabled while data is being received. After a packet is successfully received:

1. The data is initially captured into the API defined global variable `recv_pkt_buf`, and if the receive queue is enabled, the data portion of the packet (the data capsule) will be copied into the next available queue position.
2. The API defined global variable `recv_error` is set to `APPLICATION_MESSAGE`. WARNING: Do not access this variable directly if the application will be used within the GNAT simulator. The value of `recv_error` may be obtained by calling `get_recv_status()`. The number of capsules in the queue may be obtained by calling `get_recv_queue_status()`.

3. The data capsule is removed from the queue (or directly from `recv_pkt_buf`) by calling `get_capsule`:

```
get_capsule (&recv_payload_buf, sizeof(app_payload_t));
```

See the description of the API for more information.

Since receiving data is interrupt-based, it can happen at any time, unless interrupts are disabled. Your code should be written in such a way as to take this into account. The only indication will be that the condition:

```
(get_recv_status() == APPLICATION_MESSAGE)
```

is TRUE. In our applications we maintain `recv_error = NO_MESSAGE`, except when data has been received.

If the receive queue is being used, it is up to the application to ensure that it calls `get_recv_status()` and `get_recv_queue_status()` often enough that the queue is not filled. If the queue is not being used, there is an even higher probability in a densely populated area for `recv_pkt_buf` to be overwritten if not polled often enough.

With the PIC running at 4Mhz a message with a 1 byte capsule takes 57 ms to send, and if the receive queue is being used it takes an additional 80 μ s to save into the next queue slot.

4.2.5 Still To Do

There are several features and functions we are considering for future implementation:

- The self-programming feature provided by the PIC 16F87 would allow us to program the GNAT using the transmission from another GNAT. We are currently testing a GNAT Bootloader. Contact an author for details.
- We can reduce the power consumption of the GNAT by varying the speed of the processor (processor speed scaling), turning power off to the receivers and output devices (DAC and Power Regulator), and by using the sleep facility provided by the PIC 16F87.
- By varying the emitter power we could use only enough to communicate with the nearest GNATs. This might also provide an indication of distance.
- Reliable communications will require the implementation of a communication/contention/routing protocol that is more robust than the random back-off scheme we're currently using. Although, we have successfully experimented with reliable communications on the GNATs, this code is not yet mature enough to be included in this release of the Communications API.

4.3 GNAT API and Library

The API and library we've developed will take care of the basic functions:

1. Initialize a GNAT
2. Set power level
3. Display an LED
4. Transmit and receive a data packet
5. Log data to EEPROM
6. Save and restore interrupt status

The API is broken into various files. We use '#include' to integrate them into the application (instead of linking) due to limitations of the 'C' compiler. Here's a message from CCS:

```
The compiler only allows one compilation unit at this time. Our compiler
object format is not compatible with Microchips because we have a more
flexible RAM allocation scheme. CCS is working on a separate linker that
should be done soon.
```

```
You can use multiple source files in your program you just have to be
sure they are all #INCLUDEd somewhere. See EX_LCDKB.C for an example
with multiple files with device specific drivers in their own file.
```

```
=====
CCS PIC C Compiler home: www.ccsinfo.com/picc
Software download page: www.ccsinfo.com/download.shtml
Latest Compiler news: www.ccsinfo.com/news.html
```

```
If you would like to be notified when the next compiler
version comes out go to www.ccsinfo.com/cgi-bin/email.cgi
```

The GNAT API contains the following files:

GnatBaseAPIConfig.h - The configuration file for the Base GNAT API.
It also controls inclusion of the IR communication file: GnatCommAPI.h.

GnatBaseAPI.h - The base API for PIC parameters and pin definitions
as well as routines required to initialize the PIC processor and the gnat
circuit board, to set the output power levels, and LED display routines.

GnatCommAPI.h - Parameters, variables, and routines used for IR com-
munication.

GnatLib.h and GnatLib.c - Various routines (for LED displays, logging to
EEPROM, checking the battery level, etc.) that may be helpful.

4.3.1 GnatBaseAPIConfig.h

There are now 2 versions of the Gnat hardware platform. The newest version includes 2 offchip *i²c* devices, a thermal sensor and a 32K EEPROM, in addition to several PIC pin out changes. To ensure that the proper pin outs are used and that the code controlling the *i²c* devices are compiled in, be sure the following defines are correct for hardware being used (just leave the top line commented out for version 2 of the hardware, or uncomment the top line for the older version):

```
//#define GNAT_V1
#ifndef GNAT_V1
#define GNAT_V2
#endif
```

Since the PIC Microprocessor has very limited program EEPROM (4K), it is necessary to make flexibility decisions at compile time whenever possible. This file is used to control what functionality is compiled into the GNAT program. For example, to set the CPU speed at compile time to 4 MHz simply uncomment the line:

```
//#define USE_CPU_SPEED SPEED_4_MHZ
```

Other functionality controls what systems of the PIC are initialized (port direction, timer setup, enabling interrupts, etc.), using buffered data EEPROM writes, setting up the comparator, disabling interrupts during LED displays, and communication protocols. The file may be viewed in Appendix D.

To enable the IR aware and Communication routines, uncomment this line:

```
//#define USE_IR_COMMUNICATION
```

Future work will involve compiling in the bootloader, scheduler and byte code interpreter.

4.3.2 GnatBaseAPI.h

Includes basic PIC parameters and pin definitions

- Includes the 16F87.h file for you.
- PIC Fuses - to set PIC 16F87 operational parameters
- Instructions to the compiler/assembler
- 'C' type definitions - uint8, uint16, uint32
- #byte statements - define the address of select registers (PortA, PortB, OSCCON)
- #define statements - define the location of key signals for 'C' and assembly

- enum CPU_SPEED - the cpu speed selections

It also includes the basic routines required to initialize the PIC processor and the gnat circuit board, to set the output power levels, and basic LED display routines. These include:

```

init()
set_cpu_speed()
set_output_voltage()
turn_on_led()
turn_off_led()
get_interrupts()
restore_interrupts()
get_global_interrupt()
restore_global_interrupt()
NOP()
gnat_erase_program_eeprom()
gnat_write_program_eeprom()
button_down()
wait_for_button_release()
a timer1 interrupt handler
get_ticks()

```

The full function prototypes and descriptions follow.

void init (void)

- Initializes the GNAT circuit board and the PIC 16F87 ports, pins, oscillator frequency, timers, and interrupts.
- See the GnatBaseAPIConfig.h file (mentioned in the previous section) for a list of tunable parameters.

void set_cpu_speed (CPU_SPEED speed)

- Manually sets the CPU oscillator frequency. Only use if having the CPU switch speeds during execution is desired. Otherwise, set the speed at compile time using the GnatBaseAPIConfig.h file.
- Inputs:
 - CPU_SPEED speed: oscillator frequency - selected from the CPU_SPEED enumerated type.

void set_output_voltage (uint16 voltage)

- Inputs voltage into the D/A Converter and sets output power levels
- Inputs:

- int16 voltage: value to set the D/A Converter - (**NOTE: the lower the value, the higher the power level**)

void turn_on_led (uint8 led, uint16 brightness)

- Turns on an LED at a specified brightness.
- Inputs:
 - int led: selected LED
 - int16 brightness: power level - (**NOTE: the lower the value, the higher the power level**)

void turn_off_led (uint8 led)

- Turns off an LED
- Inputs:
 - int led: which led

get_interrupts (void)

- A macro that returns the value of the entire INTCON special function register. INTCON enables interrupts for peripherals, timers, and port b. It also contains the global interrupt enable bit (bit 7).
- This can be used to save the current state of interrupts before turning off interrupts to perform some interruptible operation. Use restore_interrupts() to restore.
- Returns:
 - The 8 bit value of INTCON.

void restore_interrupts (uint8 value)

- Restores a previously saved interrupt status value to the entire INTCON special function register.
- Use to restore the interrupt state after performing some interruptible operation.
- Input:
 - uint8 value: unsigned 8 bit value to be loaded into INTCON.

get_global_interrupt (void)

- A macro that returns the current value of the global interrupt enable bit (bit 7 of INTCON).

- Returns:
 - The value of the global interrupt enable bit (INTCON bit 7).

void restore_global_interrupt (uint8 value)

- Restores a previously saved value of the global interrupt enable bit (bit 7 of INTCON).
- Input:
 - uint8 value: unsigned 1 bit value to be loaded into INTCON.

void NOP (void)

- A C function wrapper for NOP asm instruction.

void gnat_erase_program_eeprom (uint16 address)

- Used to erase a 32 byte region of program EEPROM.
- This is a replacement function for the CCS C function “erase_program_eeprom()” that contains a bug. We will remove this function from the API when a bug fix is provided.
- Before writing to program EEPROM an entire 32 byte region must first be erased even if less memory needs to be written. See the C Compiler Reference Manual from CCS for further details.
- Input:
 - uint16 address: 14 bit address held in this unsigned 16 bit variable used as a starting point to begin the erase process.

uint8 gnat_write_program_eeprom (uint16 address, uint16 *instruction)

- Used to write instructions to program EEPROM.
- Instructions must be written 4 at a time. The PIC has a hardware buffer that holds 4 instructions. The actual write to program EEPROM is triggered by the 4th write.
- Before writing to program EEPROM an entire 32 byte region must first be erased even if less memory needs to be written. See the C Compiler Reference Manual from CCS for further details.
- Input:
 - uint16 address: 14 bit address used as a starting point to begin the erase process.

- uint16 *instruction: pointer to a value to be interpreted as a PIC instruction.

- Returns:

- an 8 bit value result of the write (EECON1 bit 3). 0 indicates the write operation completed. 1 indicates the write prematurely terminated. See the Microchip PIC16F87/88 Data Sheet for details.

button_down()

- A macro that returns TRUE when the button has been pressed.

wait_for_button_release()

- A macro that blocks until the button has been released.

#int_timer1 void tick_up (void)

- The tick timer that uses timer 1. By default in the GNAT environment, timer 1 interrupts every 64k instructions cycle (15 times per second for 4 MHz).
- Its only function is to increment the global variable “ticks”.

get_ticks (void)

- A macro that returns the current value of the tick timer discussed above. Applications should use this function rather than directly accessing “ticks”, as manually manipulating this variable can interfere with the proper execution of receiving IR communications.
- Returns:
 - The unsigned 32 bit value of the tick timer.

4.3.3 GnatCommAPI.h

Includes the routines, variables, and timing parameters used for sending and receiving data via the IR emitters and receivers.

- #define IR_OUT_ON_ALL 0x0F - set transmitter to this value to use all emitters simultaneously
- #define SEND_ON_B0, SEND_ON_B1, SEND_ON_B2, SEND_ON_B3 - use these to send on individual emitters by setting transmitter to this value. You may also bitwise OR these together to send on arbitrary emitters.

- enum `RECV_STATES` - defines the states of the receive routine state machine.
- signed int `recv_error` - used to indicate whether a valid packet has been received (0 = valid data, 2 = invalid data, -1 = no data)
- counters and parameters used for receipt of data (`getting_ir`, `recv_state`, `bit_number`, `receiver`, `byte_number`, etc.)
- `#define` statements - define the correspondence between each receiver and a pin in PortB

Routines to send a packet of data include:

```
send_capsule()
wait_for_channel()
```

The data packet to be sent contains an application payload buffer of 20 bytes. The application payload structure should fit in this buffer. For example:

Byte	Description
1	Data byte 1
2	Data byte 2
	...
	...
	...
20	Data byte 20

Routines to receive a packet of data include:

```
get_capsule()
get_recv_status()
get_recv_queue_status()
```

The Port B interrupt handler is responsible for receiving incoming IR. It determines if this IR represents a valid data packet. If so, it populates the API defined global variable `recv_pkt_buf` (which is defined in `GnatCommAPI.h`) with the data portion of the packet (also called the data capsule) and sets the global receive state variable, `recv_error` to `APPLICATION_MESSAGE`.

Again, if the application is to be tested with the GNAT simulator do not access these variables directly. `get_recv_status()` will return the value of `recv_error`. `get_capsule()` will retrieve the data capsule directly from `recv_pkt_buf` if the receive queue is not being used, and will retrieve the next capsule in the queue if the receive queue is activated.

Since packet reception is performed via interrupt handlers, the application only needs to check the value returned by `get_recv_status()`. If it returns `NO_MESSAGE` then no data has been received. If it returns `APPLICATION_MESSAGE` then data has been successfully received and is retrieved

via `get_capsule()`. This maximum size of a data capsule is 20 bytes. The application is responsible for processing this data before the next packet arrives or the queue fills.

#int_rb

- Interrupt handler for IR Received
- Called upon any change in PortB pins 4 – 7
- **NOTE: data will be stored in `recv_pkt_buf.data`, a 20 byte payload buffer area.**
- The global receive status variable, `recv_error`, is also set:
 - `APPLICATION_MESSAGE` = valid data received
 - `NO_MESSAGE` = no data yet received

The receive queue is configured by this section of `GnatBaseAPIConfig.h`:

```

/*-----
   Controls for IR communication received packet buffering.
   -----*/
#define USE_IR_RECEIVE_BUFFERING

#if defined USE_IR_COMMUNICATION && defined USE_IR_RECEIVE_BUFFERING

#define IR_RECEIVE_BUFFER_SIZE 2

// NOTE: one of the two defines below must be uncommented
// #define ON_QUEUE_OVERFLOW_OVERWRITE_NEWEST_CAPSULE
#define ON_QUEUE_OVERFLOW_OVERWRITE_OLDEST_CAPSULE

#endif

```

Uncomment the define for `USE_IR_RECEIVE_BUFFERING` to activate the receive queue. `IR_RECEIVE_BUFFER_SIZE` defines how large the queue is. The other 2 defines indicate how the IR receive interrupt handler should deal with a full receive queue.

The receiving helper routines:

```

receiving_data()
recv_safe_delay()
addl_action_on_ir_receive()

```

The full function prototypes and descriptions follow.

`void send_capsule (uint8 *data, uint8 transmitter, uint16 power, uint8 num_bytes)`

- Send a packet of data via the emitters
- Inputs:
 - uint8 *data: pointer to the start location of the data to transmit. The maximum payload size for this release of the API is 20 bytes.
 - uint8 transmitter: defines the transmitters to use. Bits 0 – 3 correspond to IROUT1 – IROUT4
 - uint16 power: the transmission power level
 - uint8 num_bytes: the number of bytes to transmit

uint8 wait_for_channel (uint16 wait_cycles)

- This function returns TRUE if there have been no incoming IR communications for wait_cycle instruction cycles. Returns FALSE if any data arrives in this time interval.
- Inputs:
 - uint16 wait_cycles: the number of instruction cycles to wait.
- Returns:
 - TRUE if there have been no incoming IR communications for wait_cycle instruction cycles. FALSE if any data arrives in this time interval.

recv_status_t get_recv_status (void)

- Used to determine if any valid capsule has arrived.
- Returns: an enumerated type recv_status_t. The values this type may take on in this release of the API are:

```
NO_MESSAGE = -1
APPLICATION_MESSAGE = 0
```

If APPLICATION_MESSAGE is returned then call get_capsule() to retrieve the data. If the receive queue is being used then the next capsule in the queue is returned. If not, then the data currently in recv_pkt_buf is returned.

void get_capsule (uint8 *payload_buf, uint8 length)

- If the receive queue is being used get_capsule() places the next capsule in the receive queue into the user defined buffer pointed to by payload_buf. If the receive queue is not being used the capsule is read directly from recv_pkt_buf.
- Inputs:

- uint8 *payload_buf: a pointer to the application defined receive buffer. The uint8 pointer datatype is used to make reception on a byte by byte basis possible. The application need not make a cast for the application receive buffer.
- uint8 length: the size in bytes of the application defined receive buffer.

uint8 get_rcv_queue_status(void)

- Returns: the number of capsules currently in the receive queue.
- Use the defined constant

```
#define EMPTY_QUEUE 0
```

to keep your application code readable. If the queue is not empty call get_capsule() to process the next available capsule.

- Example:

```
if (get_rcv_queue_status() != EMPTY_QUEUE)
    get_capsule(&app_rcv_pkt_buf, sizeof(app_rcv_pkt_buf));
```

void addl_action_on_ir_receive(void)

- If the application requires additional actions to be taken by the port B handler upon receipt of a good data packet, the application may define this function to perform that action.

This function will be called as the final function call in the port B interrupt handler.

For this function to be called uncomment

```
//#define USE_ADDL_ACTION_ON_IR_RECEIVE
```

in GnatBaseAPIConfig.h

uint8 receiving_data (void)

- Used to determine if any receiver is currently active due to receiving an IR communication.
- Returns:

- 0 if no receiver is active. Non-zero otherwise.

void rcv_safe_delay (uint16 time_counter)

- This is an IR communication-aware delay function. If a communication packet is received while in this function it will immediately return.
- Inputs:
 - uint16 time_counter: 16 bit counter to decrement.

4.3.4 GnatLib.h and GnatLib.c

The Gnat library contains procedures for a wide range of functionality. These include mostly LED display functions, but also included are functions for logging to the data EEPROM and checking the battery voltage level.

```
get_battery_dacvalue()
display_led()
flash_red_quick()
flash_green_quick()
display_data_byte()
display_count()
log_eeprom()
write_eeprom_buffered()
ext_eeprom_write_data()
ext_eeprom_read_data()
read_thermal_sensor()
```

uint16 get_battery_dacvalue (void)

- Returns a value that can be converted into a voltage level for the battery; more specifically, the DAC value that ends up generating a signal of about equal strength as the battery. This can be converted into a voltage by the formula:

$$(((\text{returned_value} * -1.32/4096) + 2.2) * 2.4)$$

- Returns:
 - The 16 bit DAC value.

void display_led (uint8 led, uint16 brightness, uint16 time)

- Turns on an LED for a specified period of time at a specified brightness
- Inputs:
 - uint8 led: selected LED
 - uint16 brightness: power level - (**NOTE: the lower the value, the higher the power level**)
 - uint16 time: display time (in ms)

void flash_red_quick (void)

- Blinks the red LED 5 times quickly.

void flash_green_quick (void)

- Blinks the green LED 5 times quickly.

void display_data_byte (uint8 data_byte)

- For each of the 8 bits of data_byte starting at the least significant bit, turns on the green LED for a bit value of 1 and the red LED for a bit value of 0. Each bit activates the LED for 1 second.
- Inputs:
 - uint8 data_byte: the 8 bits of data to be displayed.

void display_count (uint8 count, uint8 time)

- Displays a count value, such as hop count, as a series of green LED flashes in a constant amount of time. The duration of each flash is time/count. We have used this a way to flash ranges of hops counts in constant time. This is an IR communication-aware function.
- Inputs:
 - uint8 count: the data value to be displayed as a series of green LED flashes.
 - uint8 time: the time constant over which count is flashed.

Note that the EEPROM logging function that follows may be configured to perform buffered writes to data EEPROM. Buffering writes requires a 16 (default value) byte area in RAM to be reserved. This is done by uncommenting the line:

```
//#define USE_BUFFERED_DATA_EEPROM_WRITE
```

in the GnatBaseAPIConfig.h file. The default size can be changed on the line below:

```
//#define BUFFERED_EEPROM_ARRAY_SIZE 16
```

If buffering is not used then log_eeprom() writes directly to the data EEPROM. Logging begins at location 0x10.

When the buffer fills, log_eeprom() will automatically flush previously saved data to the EEPROM. write_eeprom.buffered() should only be called if an immediate flush is needed. Care should be taken when using buffered logging because if the PIC crashes unflushed data will be lost.

void log_eeprom (uint32 value, uint8 num_bytes)

- Logs an 8, 16 or 32 bit value into the next buffer location if buffering is enabled, or directly to the data EEPROM if buffering is not enabled. When the buffer fills, log_eeprom() will first call write_eeprom.buffered() to flush the buffer before saving the new value to the buffer.

- Inputs:
 - uint32 value: the data value to be logged. It may actually be an 8, 16, or 32 bit value. The compiler takes care of the cast.
 - uint8 num_bytes: the number of bytes of the value. Should be either 1, 2 or 4. Use the sizeof() function for safety.

void write_eeprom_buffered (void)

- Flushes the 16 byte buffer to the EEPROM location starting at location 0x10.

uint8 ext_eeprom_write_data(uint16 address, uint8 data)

- Writes the 8 bit data value to the indicated address in the i^2c data EEPROM.
- Inputs:
 - uint16 address: the address to be written to.
 - uint8 data: the value to be written.

uint8 ext_eeprom_read_data(uint16 address, uint8 *data_byte)

- Reads the 8 bit data value at the indicated address in the i^2c data EEPROM and copies it into the storage location pointed to by data_byte.
- Inputs:
 - uint16 address: the address to be read from.
 - uint8 *data_byte: a pointer where the value at address is to be stored.
- Returns:
 - TRUE if read was successful, FALSE otherwise.

uint8 read_thermal_sensor(uint8 *data_byte)

- Reads the 8 bit thermal sensor value from the i^2c thermal sensor, and copies it into the storage location pointed to by data_byte. This reading is in degrees Celsius.
- Inputs:
 - uint8 *data_byte: a pointer where the thermal sensor value is to be stored.
- Returns:
 - TRUE if read was successful, FALSE otherwise.

4.4 Sample Applications

These sample applications demonstrate how to use the API.

4.4.1 sample.c

This sample program shows the general outline of a PIC program and also shows how the GNAT API header files should be included.

The program first initializes the PIC microprocessor to run at 4 MHz. Next, it blinks the green LED quickly 3 times to show that initialization has been successful. Lastly, it enters an endless loop that blinks the red LED. The source code for sample.c is in Appendix C.

4.4.2 SendReceive_v2.c

This application demonstrates the basic capabilities of the GNAT to display LED's, and to transmit and receive data.

- When the GNATs are powered up they flash the green LED 3 times.
Each time you push the button generation is incremented and a data packet is transmitted.
- When a data packet is received, the green LED displays the generation.

4.4.3 PathPlan_v2.c

This application is used to demonstrate GNAT GOAL location propagation. It shows how a message propagates through the network, and how each NODE keeps track of its distance (hop_count) from the GOAL.

- You must assign a unique Gnat ID at position 0x0F in onchip EEPROM. The Gnats keep a list of neighbors, and must identify packets by the sender's ID.
When the GNATs are powered up they flash the green LED 3 times to show they have passed the initialization phase.
- There are 3 states a GNAT may be in. They all start out in the "node phase" where they are awaiting a hop_count signal that they may or may not display.
- A GNAT enters the second state when its button is pressed. This makes this GNAT a GOAL node. It sends hop_count information to surrounding GNATs, but with the qualification that they do NOT display.
- A NODE that receives a network message will increase the hop_count, re-transmit the message, and display the hop_count using the green LED.
- Another button press on the goal GNAT will cause the node GNATs to display their hop_counts.

- Further button presses on the goal GNAT will cause it to change from sending display to no display messages to node GNATs.
- Node GNATs will also send “I am here” messages to their neighbors. GNATs keep a neighbor list, and the “I am here” messages reset the timeout for removing that particular neighbor from the list.

4.5 Programming Tips for the PIC Microprocessor

During our development of the Gnat API and applications we have found several issues that concern the PIC itself and the CCS C compiler.

4.5.1 Compiler Case Sensitivity

By default, the CCS C compiler is not case sensitive. To turn on case sensitivity use the “#case” compiler directive. This directive is included in the GNAT API in the GnatBaseAPI.h file.

4.5.2 Program Counter, Function Call Stack, and Lack of Function Pointers

In general purpose processors it is either possible to read and write to the program counter (PC) directly or to manipulate it indirectly by changing the return address in the function call stack and then “return”ing. The function call stack is usually implemented in RAM.

Neither the program counter (PC) nor the function call stack are directly addressable/accessible on the PIC. The function call stack is implemented in hardware as a bank of registers and are manipulated only by the CALL and RETURN family of PIC instructions. It *is* possible to write to the PC indirectly if you know the exact address ahead of time, but the PIC provides no functionality to read the PC.

The PC is a 13 bit register. The least significant 8 bits are reflected in the PCL special function register. This is a fully functional read/write register for the PC. There is also a write only buffer for the upper 5 bits of the PC called PCLATH. To jump to a specific address takes 2 instructions: 1) preload PCLATH, 2) have PCL be the destination register for an operation. The write to PCL triggers an update of the PC from both PCL and PCLATH. Again, you must know the exact address of the instruction ahead of time.

But, PCLATH is write only. It is not a place to read the upper 5 bits of the PC. So, if the upper 5 bits could read the PC could be set to arbitrary locations in program EEPROM.

This also answers the question (for those who were still wondering) why there aren’t function pointers: the data bus is 8 bits wide and the instruction bus is 14 bits wide.

See the Microchip PIC16F87/88 Data Sheet under sections “2.0 Memory Organization” and “2.3 PCL and PCLATH” for more details.

4.5.3 CCS C function bugs

We discovered bugs in these 2 functions provided in the CCS C library:

- `erase_program_eeprom()`
- `write_program_eeprom()`

The problem is actually due to a bug in a previous version of the Microchip PIC16F87/88 Data Sheet on p. 31. The bug in the data sheet has since been corrected.

But, this bug was reflected in the CCS C implementation of the `erase_program_eeprom()` function. We have to date not seen a bug fix from CCS.

We have provided replacement functions: `gnat_erase_program_eeprom()` and `gnat_write_program_eeprom()`. These have been fully tested and are documented in the previous section.

4.5.4 Unused Functions

The CCS C compiler, in an effort to save on program memory space (and due to there not being a CCS linker compatible with the Microchip assembler object code format), compiles all source code lines, but also performs reachability tests to find functions that are not called. These functions not included in the executable hex image. This is true not only for user defined functions, but also for functions in standard libraries such as `stdlib.h` and `string.h`.

The `project_name.tre` file records the function call tree. The `project_name.LST` file contains the compiled source code as Microchip assembly language.

4.5.5 Interrupt Servicing and saving Global Interrupt state

The PIC `INTCON` register contains the interrupt enable and flag bits. Bit 7 is the global interrupt enable bit. See the Microchip PIC16F87/88 Data Sheet for a full listing.

When multiple interrupts are enabled and multiple interrupts occur, the first interrupt that fired is not always the first one to be serviced. The PIC interrupt dispatcher has a definite order in which it polls the interrupt flags. The Global Interrupt Enable (GIE) bit is cleared upon entry into the interrupt dispatcher to ensure that interrupt handlers cannot be interrupted (but this does not prevent interrupt flags being set). This can be overridden by using the “fast” label on the interrupt handler routine. The CCS C compiler will clear the interrupt flag upon leaving the handler routine. If multiple interrupts have fired (or if an interrupt fired while servicing another) the next interrupt in the polling sequence will be serviced.

For performing critical section (uninterruptible) code segments, merely using the CCS C functions for disabling interrupts and re-enabling interrupts is not sufficient. We provide 2 functions to assist in this: `get_global_interrupt()` and

`restore_global_interrupt()` to ensure that the global interrupt state is accurately restored. See the full description of these functions in the section above.

Here is a code snippet demonstrating the use of `get_global_interrupt()` and `restore_global_interrupt()`:

```
{
    short save_int;
    save_int = get_global_interrupt();
    disable_interrupts(GLOBAL);

    // your critical section code here

    restore_global_interrupt(save_int);
}
```

5 GNAT Development Environment

5.1 Software and Hardware Required

- Software:
 - MPLAB Integrated Development Environment (IDE). Download the latest version from the Microchip web site: <http://www.microchip.com>. The manual for MPLAB IDE can be found at: <http://ww1.microchip.com/downloads/en/DeviceDoc/51025e.pdf>
 - CCS C Compiler for Microchip PICmicro MCUs. Acquire and install from CCS, Inc.: <http://www.ccsinfo.com/picc.shtml>. The compiler manual can be downloaded from: <http://www.ccsinfo.com/ccsmanual.zip>.
 - Provided by GT:
 - * GNAT API
 - * Sample Applications
- Hardware:
 - Microchip PICSTART Plus Development Programmer
 - PC with a RS-232 port running Windows XP
 - Power supply for programming adapter: 5V / .1A
 - Provided by GT:
 - * Programming adapter for GNATs
 - * GNATs and batteries

5.2 Setting Up the Development Environment

1. Purchase/acquire/download the hardware and/or software listed above.
2. Install the MPELAB IDE on the PC
3. Install the CCS C Compiler on the PC
4. Connect the Microchip PICSTART Plus Development Programmer
 - (a) Use the serial cable provided to connect the programmer to the PC's RS-232 port
 - (b) Connect the A/C adapter to an outlet and to the programmer
5. Start the MPLAB IDE
6. Configure the IDE for the Programmer
 - (a) Choose Programmer/Select Programmer/PICSTART Plus
 - (b) Choose Programmer/Settings
 - (c) Click on the Communications Tab
 - (d) Select the comm. Port corresponding to the RS-232 port where you connected the programmer
 - (e) Click OK
 - (f) Select Programmer/Enable Programmer - the IDE should display the progress at the bottom of the screen
7. Install the gnat programming adapter to the PICSTART Plus programmer. Be careful to align the pins correctly with the socket. Leave enough room between the adapter and the base of the programmer so that a gnat can be connected to the programming socket.
8. Connect the Power leads from the adapter to a power supply: 5V, .1Amp

5.3 Create a new Project

Here we describe how to get started with a new project using the API provided as well as a simple application that will send and receive data between two gnats.

1. Create a directory for the new project (referred to as Project1)
2. Copy the following files into the new directory Project1:
 - (a) GnatBaseAPIConfig.h
 - (b) GnatBaseAPI.h
 - (c) GnatCommAPI.h
 - (d) GnatLib.h
 - (e) sample.c
3. Start the MPLAB IDE
4. Create a new project
 - (a) Choose Project/Project Wizard
 - (b) Click Next
 - (c) Select a device: choose PIC16F87 from the drop-down box
 - (d) Click Next
 - (e) Select a language toolsuite
 - i. Active Toolsuite: select CCS C Compiler for PIC12/14/16/18 from the drop-down box
 - ii. Toolsuite Contents: CCS C Compiler (ccsc.exe) should be highlighted
 - iii. Location:
C:\Program
files\Picc\CCSC.exe,
or use the Browse to locate the file CCSC.exe
 - iv. Click Next
 - (f) Name your project
 - i. Project name: Project1
 - ii. Project Directory: use Browse to locate the Project1 directory created above and Click Select
 - iii. Click Next
 - (g) Add any existing files to your project
 - i. Select sample.c
 - ii. Click Add
 - iii. Select the checkbox next to the file in the right-hand window
 - iv. Click Next
 - (h) Click Finish

5.4 Compiling and Downloading a Project

1. If the project is not open, then open the project
 - (a) Select Project/Open
 - (b) Navigate to the Project1 directory
 - (c) Select Project1.mcp
 - (d) Click Open
2. Compile the code: select Project/Compile. The output window will display the status to the compile
3. Connect a gnat to the programming adapter
 - (a) Make sure the gnat is not powered you can slide a piece of paper between the battery and the connector to avoid having to remove and re-insert the battery
 - (b) Turn on the power supply to the programmer
 - (c) Make sure the programmer is powered
 - (d) Place the programming/power switch on the adapter to program
 - (e) Insert a gnat into the programming socket
4. Program the gnat
 - (a) Select Programmer/Select Programmer/PICSTART Plus
 - (b) Select Programmer/Enable Programmer
 - (c) Select Programmer/Program
 - (d) You will see the progress of the programmer at the bottom of the screen.
5. Remove the gnat from the programmer when done
6. Power the gnat to operate.

A Signals Organized by Function

A.1 Gnat hardware version 2

Power Controls

Signal	Port	Pin and Description	Processor Pin
\overline{TCNTRL}	A1	Power to D/A Converter and Power Regulator	24
\overline{RCNTRL}	A6	Power to IR Receivers	20

D/A Converter Signals

Signal	Port	Pin and Description	Processor Pin
\overline{DACS}	A7	D/A Sync	21
DACLK	A4	D/A Clock Input	28
DADIN	A0	D/A Data In	23

Infrared Emitters

Signal	Port and Pin	Description	Processor Pin
IROUT1	B1	IR Emitter 1	8
IROUT2	B2	IR Emitter 2	9
IROUT3	B3	IR Emitter 3	10
IROUT4	B0	IR Emitter 4	7

Infrared receivers

Signal	Port and Pin	Description	Processor Pin
$\overline{IRIN1}$	B5	IR Receiver 1	13
$\overline{IRIN2}$	B4	IR Receiver 2	12
$\overline{IRIN3}$	B7	IR Receiver 3	16
$\overline{IRIN4}$	B6	IR Receiver 4	15

LED Displays

Signal	Port and Pin	Description	Processor Pin
LOUT1	A2	LED Out 1 - Red	26
LOUT2	A3	LED Out 2 - Green	27

Input Button

Signal	Port and Pin	Description	Processor Pin
\overline{BUTT}	A5	Button Input	1

A.2 Gnat hardware version 1

Power Controls

Signal	Port	Pin and Description	Processor Pin
\overline{TCNTRL}	B0	Power to D/A Converter and Power Regulator	7
\overline{RCNTRL}	A7	Power to IR Receivers	21

D/A Converter Signals

Signal	Port	Pin and Description	Processor Pin
\overline{DACS}	A6	D/A Sync	20
DACLK	A4	D/A Clock Input	28
DADIN	A0	D/A Data In	23

Infrared Emitters

Signal	Port and Pin	Description	Processor Pin
IROUT1	B1	IR Emitter 1	8
IROUT2	B2	IR Emitter 2	9
IROUT3	B3	IR Emitter 3	10
IROUT4	A1	IR Emitter 4	24

Infrared receivers

Signal	Port and Pin	Description	Processor Pin
$\overline{IRIN1}$	B6	IR Receiver 1	15
$\overline{IRIN2}$	B7	IR Receiver 2	16
$\overline{IRIN3}$	B5	IR Receiver 3	13
$\overline{IRIN4}$	B4	IR Receiver 4	12

LED Displays

Signal	Port and Pin	Description	Processor Pin
LOUT1	A2	LED Out 1 - Red	26
LOUT2	A3	LED Out 2 - Green	27

Input Button

Signal	Port and Pin	Description	Processor Pin
\overline{BUTT}	A5	Button Input	1

B Built-in commands

These built-in commands are provided by the ‘C’ compiler to facilitate access to the PIC 16F87’s ports, special registers, timers and counters, interrupts.

#int_timerX - function to handle an interrupt on timerx overflow

#int_rb - function to handle an interrupt generated by a change in PortB pins 4 - 7

bit_clear (var, bit) - clears the specified bit in the given variable

bit_set(var, bit) - sets the specified bit in the given variable

bit_test (var, bit) - tests the specified bit in the given variable, returns 0 or 1

delay_ms (time) - performs a delay of the specified length in milliseconds

delay_us (time) - performs a delay of the specified length in microseconds

disable_interrupts (level) - disable interrupts at the specified level

enable_interrupts (level) - enables interrupts at the specified level

get_timerX() - returns the value of a real-time clock/counter

input (pin) - returns the state of the indicated pin.

input_x() - returns an entire byte from a port

output_x (value) - output an entire byte to a port

output_high (pin) - sets a given pin to the high state

output_low (pin) - sets a given pin to the low state

rand() - returns a pseudo-random integer

set_timerX(value) - sets the count value of a real-time clock

set_tris_x (value) - allow the i/o port direction registers to be set

setup_timer_0 (mode) - sets up the timer mode

srand(n) - seeds the pseudo-random number generator

C sample.c

```
/*
 *
 * Gnat sample program
 *
 * This program expects as input byte code that has been compiled
 *
 * Revision date:
 *   Feb 7, 2005
 *
 * Revision History:
 *
 * Contributors:
 *   Victor Bigio
 *   Eric Dodson
 *   Arya Irani
 *   Keith O'Hara
 *
 */

#include "GnatBaseAPI.h"    // Always include this file.

#ifdef USE_IR_COMMUNICATION // Change this in GnatBaseAPIConfig.h
#include "GnatCommAPI.h"
#endif

#include "GnatLib.h"       // Gnat library

/*-----
   Program Constants
   -----*/

/*-----
   Function prototypes
   -----*/

/*-----
   Global variable definitions
   -----*/

/*-----
```

```

    main() -
    -----*/
void main() {

    uint8 i;

    // Initialize the GNAT hardware
    init();

    // Blink green LED 3 times after init
    for (i = 0; i < 3; i++) {
        display_led (GREEN_LED, GREEN_LED_BR, 100);
        delay_ms(100);
    }

    // main loop
    while (1) {
        // Your code here

        // For example, to blink the red LED forever...
        turn_on_led(RED_LED, RED_LED_BR);
        delay_ms(500);
        turn_off_led(RED_LED);
        delay_ms(500);
    }
}

```

D GnatBaseAPIConfig.h

```
/*
 *
 * GnatBaseAPIConfig.h
 * The configuration file for the Base API for the
 * GNAT hardware platform
 *
 * Version: 2.1
 *
 * Revision date:
 * April 7, 2005
 *
 * Revision History:
 * April 7, 2005 - ERD - Added new Gnat hardware code
 *
 * Contributors:
 * Victor Bigio
 * Eric Dodson
 * Arya Irani
 * Keith O'Hara
 *
 */

#ifndef __GNATBASEAPICONFIG_H_
#define __GNATBASEAPICONFIG_H_

/*-----
 Define GNAT_V1 for compilation for the original Gnat
 (without the thermal sensor and 32K offchip flash)
-----*/
// #define GNAT_V1
#ifndef GNAT_V1
#define GNAT_V2
#endif

/*-----
 Don't let "int", "short", etc types be defined.
 Use with simulator only.
-----*/
// #define STRICT_TYPES

/*-----
```

```

    USE_CPU_SPEED is used to statically set the CPU speed.
    -----*/
#define USE_CPU_SPEED SPEED_31_25_KHZ
#define USE_CPU_SPEED SPEED_125_KHZ
#define USE_CPU_SPEED SPEED_250_KHZ
#define USE_CPU_SPEED SPEED_500_KHZ
#define USE_CPU_SPEED SPEED_1_MHZ
#define USE_CPU_SPEED SPEED_2_MHZ
#define USE_CPU_SPEED SPEED_4_MHZ
#define USE_CPU_SPEED SPEED_8_MHZ

/*-----
    Optional modules
    -----*/
#define USE_INTERPRETER

#ifdef USE_INTERPRETER
#define SIXTEENBIT_INTERPRETER
#ifndef SIXTEENBIT_INTERPRETER
#define EIGHTBIT_INTERPRETER
#endif
#endif

/*-----
    These define the default functionality of init()
    -----*/
#define SET_PORT_DIRECTION_IN_INIT
#define TURN_ON_RECEIVERS_IN_INIT
#define TURN_ON_DAC_IN_INIT
#define SETUP_TIMER_0_IN_INIT
#define SETUP_TIMER_1_IN_INIT
#define SETUP_TIMER_2_IN_INIT
// if an interrupt handler is defined you must enable interrupts
#define ENABLE_INTERRUPTS_IN_INIT

/*-----
    Override API provided interrupt handlers
    -----*/
#define USE_DEFAULT_TIMER0_INTERRUPT_HANDLER
#define USE_DEFAULT_TIMER1_INTERRUPT_HANDLER
#define USE_DEFAULT_TIMER2_INTERRUPT_HANDLER
#define USE_DEFAULT_PORTB_INTERRUPT_HANDLER

```

```

/*-----
   Determine the type of communication.
   -----*/
#define USE_IR_COMMUNICATION
//#define USE_BLOCKING_RECV <--- not yet implemented

/*-----
   Uncomment for buffered data eeprom writes
   NOTE: this will consume of RAM!
   The default size is 16 bytes; change this value below.
   To stop processing on a full buffer; uncomment appropriate
   line below.
   -----*/
//#define USE_BUFFERED_DATA_EEPROM_WRITE
//#define BUFFERED_EEPROM_ARRAY_SIZE 16
//#define STOP_PROCESSING_ON_FULL_BUFFER

/*-----
   Size of the data buffer in the communication capsule
   -----*/
#define CAPSULE_DATA_BUFFER_SIZE 16

/*-----
   Controls for IR communication received packet buffering.
   -----*/
//#define USE_IR_RECEIVE_BUFFERING

#if defined USE_IR_COMMUNICATION && defined USE_IR_RECEIVE_BUFFERING

#define IR_RECEIVE_BUFFER_SIZE 2

// NOTE: one of the two defines below must be uncommented
//#define ON_QUEUE_OVERFLOW_OVERWRITE_NEWEST_CAPSULE
#define ON_QUEUE_OVERFLOW_OVERWRITE_OLDEST_CAPSULE

#endif

/*-----
   If the application needs to have the comm API perform
   additional actions from within the IR receive interrupt
   handler, the application must do so from within and

```

```

define this function.

void addl_action_on_ir_receive(void);

-----*/
//#define USE_ADDL_ACTION_ON_IR_RECEIVE

/*-----
DISABLE_INTERRUPTS_IN_DISPLAY_LED
DISABLE_INTERRUPTS_IN_SET_OUTPUT_VOLTAGE
-----*/
//#define DISABLE_INTERRUPTS_IN_DISPLAY_LED
//#define DISABLE_INTERRUPTS_IN_SET_OUTPUT_VOLTAGE

/*-----
Uncomment if comparator will be used
-----*/
//#define USE_COMPARATOR

/*-----
The parity bit calculation is a small piece of code that can
be included or excluded as needed.
If both types are needed in an application then, choose
NO_PARITY_BIT and have your send routines calculate them.
NOT YET IMPLEMENTED.
-----*/
//#define NO_PARITY_BIT
//#define USE_ODD_PARITY
//#define USE_EVEN_PARITY

/*-----
The same argument holds for the checksum.
NOT YET IMPLEMENTED.
-----*/
//#define CALC_CHECKSUM

/*-----
Optional modules
NOT YET IMPLEMENTED.
-----*/
//#define USE_BOOTLOADER

```

```
//#define USE_SCHEDULER
//#define SIMULATOR

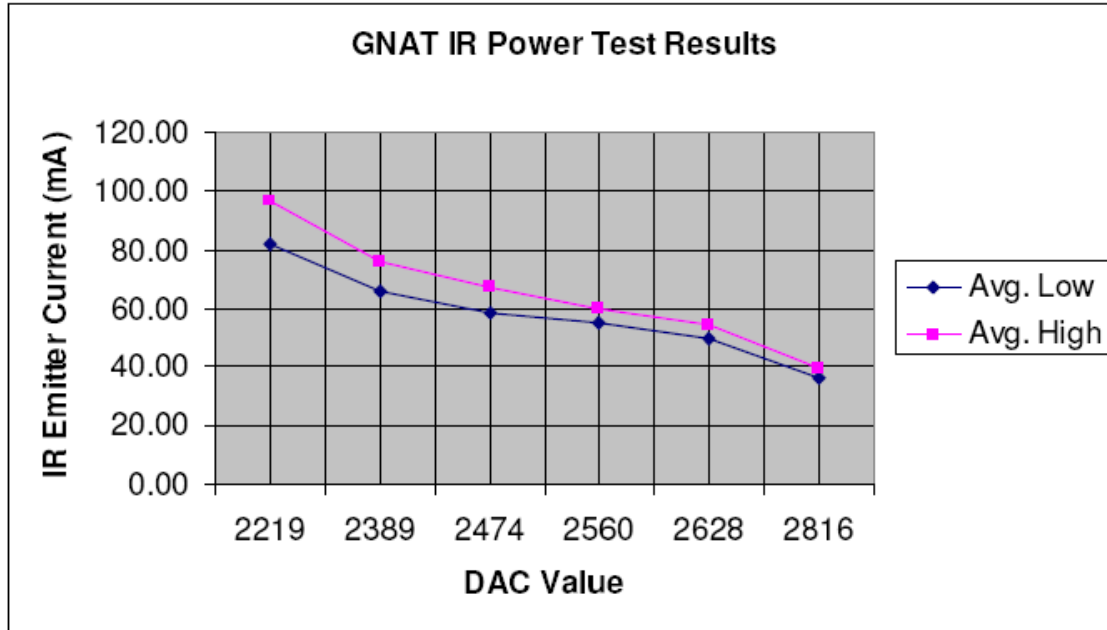
#endif
```

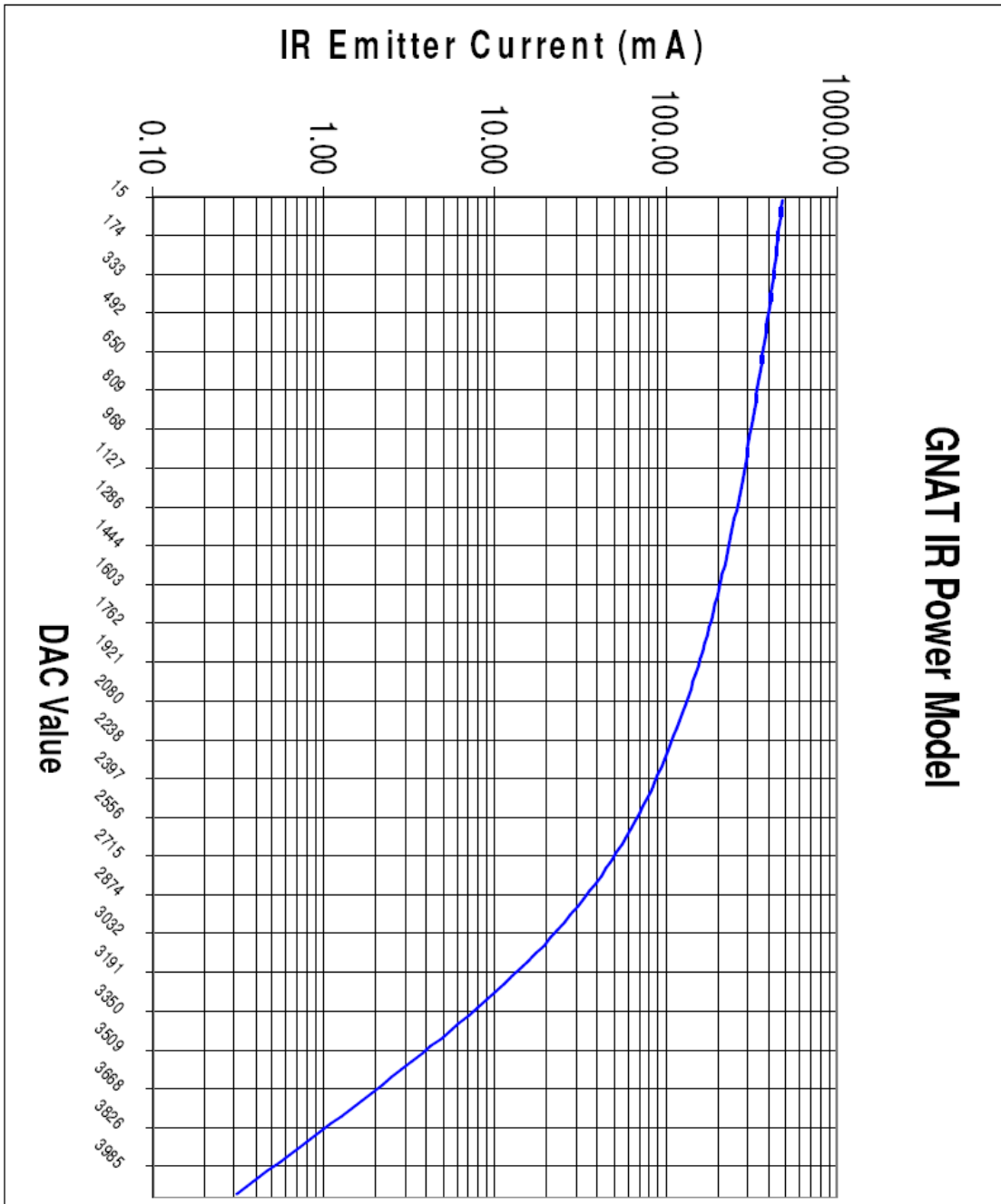
E Data EEPROM Memory Map

The data EEPROM in this version of GNAT is 256 bytes. While most of this is available for application data logging there are some areas which are reserved: namely the first 16 bytes.

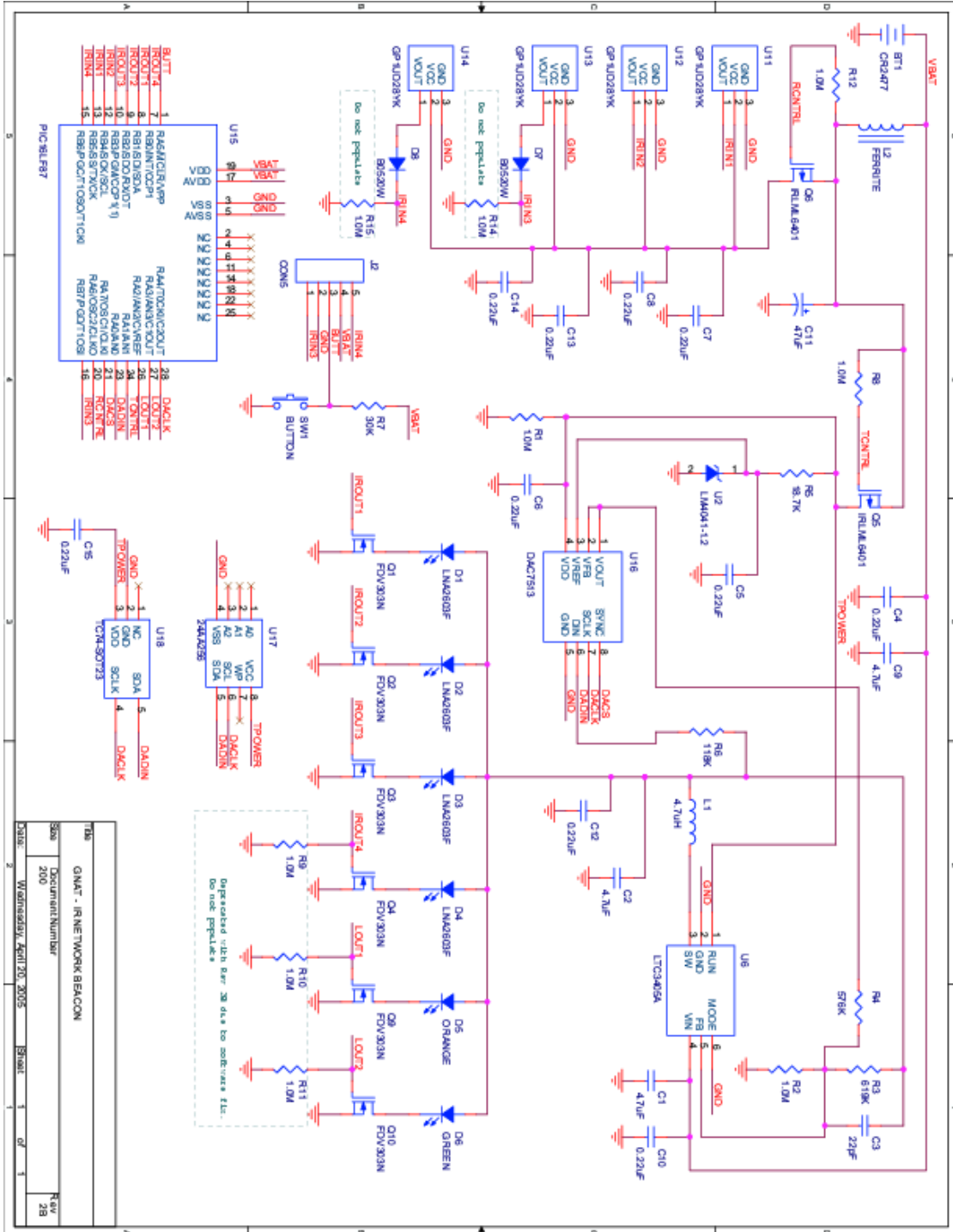
```
0x00 - 0x0E Reserved for Bootloader
      0x0F Gnat ID
0x10 - 0xFF Application Logging area
```

F IR Power Test and Power Model

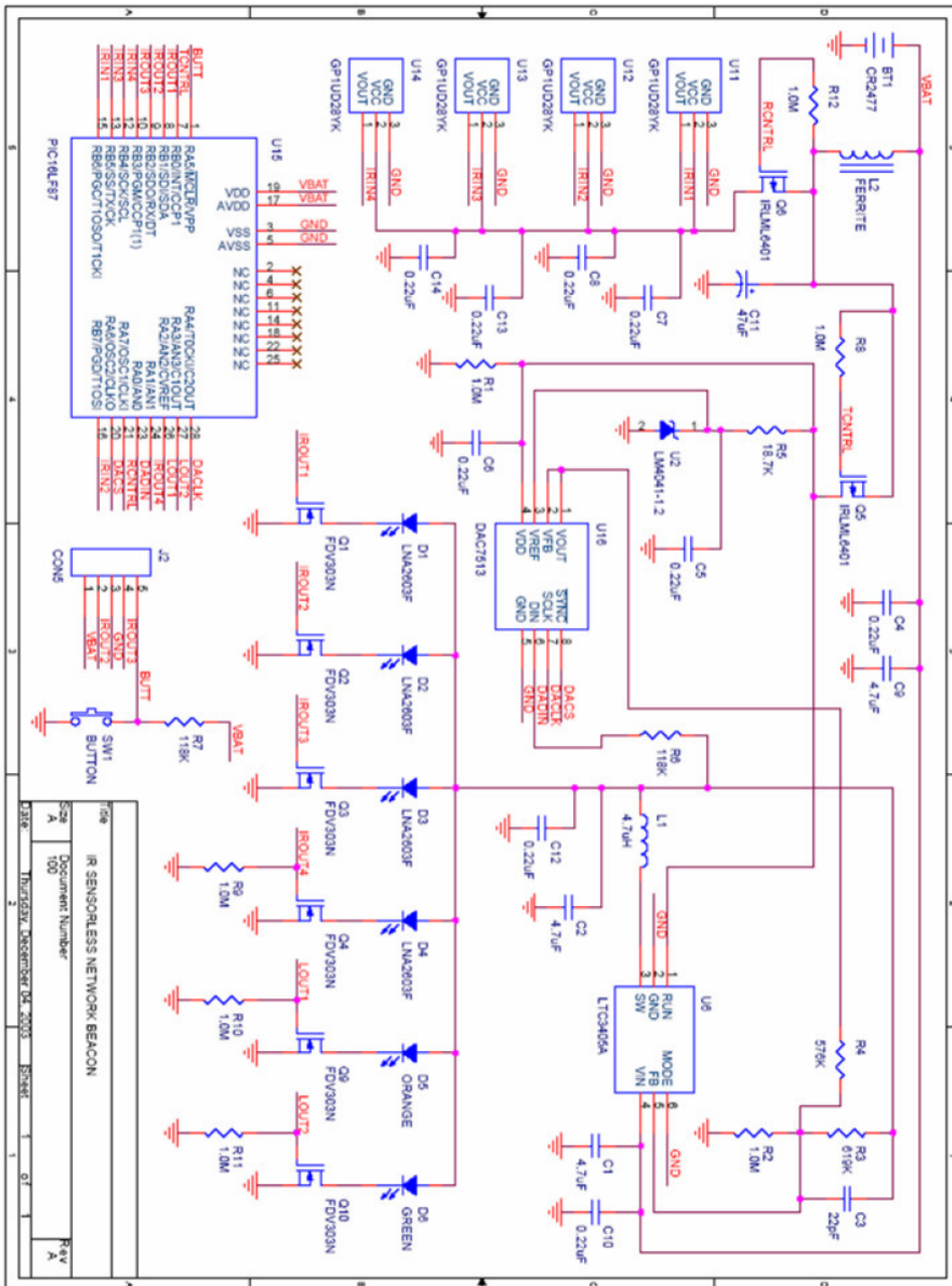




G GNAT Hardware Version 2 Circuit Diagram



H GNAT Hardware Version 1 Circuit Diagram



I List of Parts

Qty	Part Number	Description
1	PIC16LF87-I/ML	Microchip PIC processor 2.0V
7	ECJ-0EB0J224K	CAP .22UF 6.3V CERAMIC X5R (Decoupling)
1	BLM15AG102SN1D	FERRITE CHIP 1000 OHM 200MA 1 ohm DCR
4	LNA2603F	Panasonic IR Emitter Side View 160 degree
1	LTST-C190KGKT	LED GREEN 574 nm InGaAlP High Efficiency
1	GM1JJ35200AE	LED ORANGE 627nm InGaAlP High Efficiency
1	EVQ-P2402M	Momentary push button
1	ERJ-2RKF1183X	RES 118K OHM 1/16W 1%
1		RES 39.2K OHM 1/16W 1%
7	ERJ-2RKF1004X	RES 1.00M OHM 1/16W 1%
4	GP1UD28YK	Sharp 40kHz IR receiver Side View 2.7V
1	ECE-V0GA470SR	CAP 47UF 4V VS ELECT SMD
1	CR2477	Lithium Coin Cell 1000mAh 24mm diameter
1	BH1000-G	CR2477 Coin Cell Holder
1	LTC3405AES6	Switching Regulator 300mA 1.5Mhz 96%eff 2.5V
1	ELJ-EA4R7MF	INDUCTOR 4.7UH 0.18ohm DCR 240mA
3	ECJ-1VB0J475M	CAP CERAMIC 4.7UF 6.3V X5R
1	ERJ-2RKF6193X	RES 619K OHM 1/16W 1%
1	C0402C220J5GACTU	CAP CERAMIC 22PF 50V NP0 (Compensation)
1	DAC7513N/250	D/A Converter R-R 12bit 2.7V
1	ERJ-2RKF5763X	RES 576K OHM 1/16W 1%
1	LM4041CIM3-1.2	IC precision 0.5% micropower voltage ref 1.2V
1	ERJ-2RKF1872X	RES 18.7K OHM 1/16W 1%
6	FDV303N	MOSFET N-CH LOGIC low gate drive 0.6ohm
2	IRLML6401	HEX/MOS P-CH -12V -4.3A low gate drive 0.05ohm
1	24AA256	MicroChip 256K <i>i²c</i> CMOS Serial EEPROM *
1	TC74-SOT23	MicroChip <i>i²c</i> Tiny Serial Digital Thermal Sensor *

* Available in GNAT Hardware Version 2 only.