

“*EXPLORING* THE CHAMELEON AVR 8-BIT”

USER MANUAL AND PROGRAMMING GUIDE

Andre' LaMothe

CHAMELEON™ AVR 8-Bit User Manual v1.0

“Exploring the CHAMELEON AVR 8-Bit – A Guide to Programming the CHAMELEON AVR 8-Bit System”

Copyright © 2009 Nurve Networks LLC

Author

Andre’ LaMothe

Editor/Technical Reviewer

The “Collective”

Printing

0001

ISBN

Pending

All rights reserved. No part of this user manual shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the user of the information contained herein. Although every precaution has been taken in the preparation of this user manual, the publisher and authors assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

Trademarks

All terms mentioned in this user manual that are known to be trademarks or service marks have been appropriately capitalized. Nurve Networks LLC cannot attest to the accuracy of this information. Use of a term in this user manual should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this user manual as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “*as is*” basis. The authors and the publisher shall have neither liability nor any responsibility to any person or entity with respect to any loss or damages arising from the information contained in this user manual.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

eBook License

This electronic user manual may be printed for personal use and (1) copy may be made for archival purposes, but may not be distributed by any means whatsoever, sold, resold, in any form, in whole, or in parts. Additionally, the contents of the DVD this electronic user manual came on relating to the design, development, imagery, or any and all related subject matter pertaining to the CHAMELEON™ systems are copyrighted as well and may not be distributed in any way whatsoever in whole or in part. Individual programs are copyrighted by their respective owners and may require separate licensing.

Licensing, Terms & Conditions

NURVE NETWORKS LLC, . END-USER LICENSE AGREEMENT FOR CHAMELEON AVR HARDWARE, SOFTWARE, EBOOKS, AND USER MANUALS

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS BEFORE USING THIS PRODUCT. IT CONTAINS SOFTWARE, THE USE OF WHICH IS LICENSED BY NURVE NETWORKS LLC, INC., TO ITS CUSTOMERS FOR THEIR USE ONLY AS SET FORTH BELOW. IF YOU DO NOT AGREE TO THE TERMS AND CONDITIONS OF THIS AGREEMENT, DO NOT USE THE SOFTWARE OR HARDWARE. USING ANY PART OF THE SOFTWARE OR HARDWARE INDICATES THAT YOU ACCEPT THESE TERMS.

GRANT OF LICENSE: NURVE NETWORKS LLC (the "Licensor") grants to you this personal, limited, non-exclusive, non-transferable, non-assignable license solely to use in a single copy of the Licensed Works on a single computer for use by a single concurrent user only, and solely provided that you adhere to all of the terms and conditions of this Agreement. The foregoing is an express limited use license and not an assignment, sale, or other transfer of the Licensed Works or any Intellectual Property Rights of Licensor.

ASSENT: By opening the files and or packaging containing this software and or hardware, you agree that this Agreement is a legally binding and valid contract, agree to abide by the intellectual property laws and all of the terms and conditions of this Agreement, and further agree to take all necessary steps to ensure that the terms and conditions of this Agreement are not violated by any person or entity under your control or in your service.

OWNERSHIP OF SOFTWARE AND HARDWARE: The Licensor and/or its affiliates or subsidiaries own certain rights that may exist from time to time in this or any other jurisdiction, whether foreign or domestic, under patent law, copyright law, publicity rights law, moral rights law, trade secret law, trademark law, unfair competition law or other similar protections, regardless of whether or not such rights or protections are registered or perfected (the "Intellectual Property Rights"), in the computer software and hardware, together with any related documentation (including design, systems and user) and other materials for use in connection with such computer software and hardware in this package (collectively, the "Licensed Works"). ALL INTELLECTUAL PROPERTY RIGHTS IN AND TO THE LICENSED WORKS ARE AND SHALL REMAIN IN LICENSOR.

RESTRICTIONS:

- (a) You are expressly prohibited from copying, modifying, merging, selling, leasing, redistributing, assigning, or transferring in any matter, Licensed Works or any portion thereof.
- (b) You may make a single copy of software materials within the package or otherwise related to Licensed Works only as required for backup purposes.
- (c) You are also expressly prohibited from reverse engineering, decompiling, translating, disassembling, deciphering, decrypting, or otherwise attempting to discover the source code of the Licensed Works as the Licensed Works contain proprietary material of Licensor. You may not otherwise modify, alter, adapt, port, or merge the Licensed Works.
- (d) You may not remove, alter, deface, overprint or otherwise obscure Licensor patent, trademark, service mark or copyright notices.
- (e) You agree that the Licensed Works will not be shipped, transferred or exported into any other country, or used in any manner prohibited by any government agency or any export laws, restrictions or regulations.
- (f) You may not publish or distribute in any form of electronic or printed communication the materials within or otherwise related to Licensed Works, including but not limited to the object code, documentation, help files, examples, and benchmarks.

TERM: This Agreement is effective until terminated. You may terminate this Agreement at any time by uninstalling the Licensed Works and destroying all copies of the Licensed Works both HARDWARE and SOFTWARE. Upon any termination, you agree to uninstall the Licensed Works and return or destroy all copies of the Licensed Works, any accompanying documentation, and all other associated materials.

WARRANTIES AND DISCLAIMER: EXCEPT AS EXPRESSLY PROVIDED OTHERWISE IN A WRITTEN AGREEMENT BETWEEN LICENSOR AND YOU, THE LICENSED WORKS ARE NOW PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, OR THE WARRANTY OF NON-INFRINGEMENT. WITHOUT LIMITING THE FOREGOING, LICENSOR MAKES NO WARRANTY THAT (i) THE LICENSED WORKS WILL MEET YOUR REQUIREMENTS, (ii) THE USE OF THE LICENSED WORKS WILL BE UNINTERRUPTED, TIMELY, SECURE, OR ERROR-FREE, (iii) THE RESULTS THAT MAY BE OBTAINED FROM THE USE OF THE LICENSED WORKS WILL BE ACCURATE OR RELIABLE, (iv) THE QUALITY OF THE LICENSED WORKS WILL MEET YOUR EXPECTATIONS, (v) ANY ERRORS IN THE LICENSED WORKS WILL BE CORRECTED, AND/OR (vi) YOU MAY USE, PRACTICE, EXECUTE, OR ACCESS THE LICENSED WORKS WITHOUT VIOLATING THE INTELLECTUAL PROPERTY RIGHTS OF OTHERS. SOME STATES OR JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES OR LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY MAY LAST, SO THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU. IF CALIFORNIA LAW IS NOT HELD TO APPLY TO THIS AGREEMENT FOR ANY REASON, THEN IN JURISDICTIONS WHERE WARRANTIES, GUARANTEES, REPRESENTATIONS, AND/OR CONDITIONS OF ANY TYPE MAY NOT BE DISCLAIMED, ANY SUCH WARRANTY, GUARANTEE, REPRESENTATION AND/OR WARRANTY IS: (1) HEREBY LIMITED TO THE PERIOD OF EITHER (A) Five (5) DAYS FROM THE DATE OF OPENING THE PACKAGE CONTAINING THE LICENSED WORKS OR (B) THE SHORTEST PERIOD ALLOWED BY LAW IN THE APPLICABLE JURISDICTION IF A FIVE (5) DAY LIMITATION WOULD BE UNENFORCEABLE; AND (2) LICENSOR'S SOLE LIABILITY FOR ANY BREACH OF ANY SUCH WARRANTY, GUARANTEE, REPRESENTATION, AND/OR CONDITION SHALL BE TO PROVIDE YOU WITH A NEW COPY OF THE LICENSED WORKS. IN NO EVENT SHALL LICENSOR OR ITS SUPPLIERS BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR ANY DAMAGES WHATSOEVER, INCLUDING, WITHOUT LIMITATION, THOSE RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER OR NOT LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, AND ON ANY THEORY OF LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE OF THE LICENSED WORKS. SOME JURISDICTIONS PROHIBIT THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, SO THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING ANY FAILURE OF ESSENTIAL PURPOSE OF ANY LIMITED REMEDY.

SEVERABILITY: In the event any provision of this License Agreement is found to be invalid, illegal or unenforceable, the validity, legality and enforceability of any of the remaining provisions shall not in any way be affected or impaired and a valid, legal and enforceable provision of similar intent and economic impact shall be substituted therefore.

ENTIRE AGREEMENT: This License Agreement sets forth the entire understanding and agreement between you and NURVE NETWORKS LLC, supersedes all prior agreements, whether written or oral, with respect to the Software, and may be amended only in a writing signed by both parties.

NURVE NETWORKS LLC
12724 Rush Creek Lane
Austin, TX 78732
support@nurve.net
www.xgamestation.com

Version & Support/Web Site

This document is valid with the following hardware, software and firmware versions:

- CHAMELEON AVR 8-Bit Board Revision A. or greater.
- Atmel AVR Studio 4.14 or greater.
- Arduino Toolchain 0017 or greater for Windows/Linux/Mac OS X.
- Propeller IDE 1.26 or greater.

The information herein will usually apply to newer versions but may not apply to older versions. Please contact Nurve Networks LLC for any questions you may have.

Visit **www.xgamestation.com** & **www.chameleon-dev.com** for downloads, support and access to the Chameleon user community and more!

For technical support, sales, general questions, share feedback, please contact Nurve Networks LLC at:

support@nurve.net / nurve_help@yahoo.com

"Exploring the CHAMELEON AVR 8-Bit"

User Manual and Programming Guide

LICENSING, TERMS & CONDITIONS	3
VERSION & SUPPORT/WEB SITE.....	4
"EXPLORING THE CHAMELEON AVR 8-BIT" <i>USER MANUAL AND PROGRAMMING GUIDE</i>	5
0.0 INTRODUCTION AND ORIGINS.....	11
1.0 ARCHITECTURAL OVERVIEW	13
1.1 Package Contents	15
1.2 Chameleon AVR "Quick Start" Demo.....	16
First Things First.....	16
Playing Crate-It!.....	17
1.3 The Atmel Mega AVR328P Chip.....	19
1.3 The Parallax Propeller Chip	24
1.31 Propeller Core (COG) Video Hardware	26
1.4. System Startup and Reset Details	26
PART I - HARDWARE DESIGN PRIMER	26
2.0 5.0V & 3.3V POWER SUPPLIES	29
3.0 RESET CIRCUIT	29
4.0 ATMEL 6-PIN ISP PROGRAMMING PORT	30
4.1 AVR ISP Programming Port	30
5.0 SERIAL USB UART PROGRAMMING PORT	31
6.0 USB SERIAL UART	33
7.0 ATMEL AVR 328P SUBSYSTEM	34
8.0 PARALLAX PROPELLER SUBSYSTEM	35
8.1 The Propeller Local 8-Bit I/O Port	35
9.0 THE SPI BUS AND COMMUNICATIONS SYSTEM	36
10.0 VGA GRAPHICS HARDWARE	36

10.1 Origins of the VGA.....	37
10.2 VGA Hardware Interface.....	38
10.3 VGA Signal Primer	41
10.3.1 VGA Horizontal Timing.....	42
10.3.2 VGA Vertical Timing.....	42
10.3.3 Generating the Active VGA Video	43
11.0 NTSC/PAL COMPOSITE VIDEO HARDWARE	44
11.1 Video Hardware Interface.....	44
11.2 Introduction to NTSC Video	45
11.2.1 Interlaced versus Progressive Scans.....	45
11.2.2 Video Formats and Interfaces	47
11.2.3. Composite Color Video Blanking Sync Interface	47
11.2.4 Color Encoding.....	49
11.2.5 Putting it All Together.....	49
11.2.6 Generating B/W Video Data.....	52
11.2.7 Generating Color Video Data	52
11.2.8 NTSC Signal References	53
12.0 KEYBOARD & MOUSE HARDWARE.....	53
12.1 Keyboard Operation.....	54
12.1.1 Communication Protocol from Keyboard to Host.....	55
12.1.2 Keyboard Read Algorithm.....	56
12.1.3 Keyboard Write Algorithm.....	56
12.1.4 Keyboard Commands.....	57
12.2 Communication Protocol from Mouse to Host	58
12.2.1 Basic Mouse Operation	59
12.2.2 Mouse Data Packets	59
12.2.3 Modes of Operation	60
12.2.4 Sending Mouse Commands.....	61
12.2.5 Mouse Initialization	63
12.2.6 Reading Mouse Movement.....	63
13.0 THE I/O HEADERS.....	63
14.0 AUDIO HARDWARE	65
14.1 A Little Background on Low Pass Filters (EE stuff)	66
14.1.1 Pulse Code Modulation (PCM).....	67
14.1.2 Frequency Modulation (FM)	68
14.1.3 Pulse Width Modulation (PWM).....	68
15.0 INSTALLING THE TOOL CHAINS: AVRSTUDIO, ARDUINO, AND PROPELLER IDE	75
15.1 ATMEL'S AVR STUDIO TOOLCHAIN OVERVIEW.....	76
15.1.1 Installing AVR Studio 4.xx (Optional)	77
15.1.2 Installing the AVR ISP MKII Hardware (Optional)	81
15.1.3 Installing WinAVR™	82

15.1.4 Building a Project and Testing the Tool Chain.....	86
15.1.4.1 Setting up the Project Options	89
Map File Example (.map)	91
List File Example (.lss).....	91
Hex File Example (.hex).....	92
15.1.4.2 Adding Files to the Project	94
15.1.5 Setting up the AVR ISP MKII Hardware.....	98
15.1.7 Final Words on AVR Studio Tool Chain Installation	105
15.2 ARDUINO TOOLCHAIN SETUP	105
15.2.1 Installing the Arduino Toolchain in Windows	106
15.2.2 Copying the Files to Your Hard Drive	107
15.2.3 Preparation to Launch the Arduino Tool for the First Time.....	110
15.2.3.1 Installing a Serial Terminal Program.....	112
15.2.3.1 Running the Arduino Tool.....	114
15.2.3.2 Loading the Hello World Sketch.....	118
15.2.3.3 A Couple Notes About the Arduino Version of Hello World	121
15.3 INSTALLING THE PARALLAX PROPELLER IDE	121
15.3.1 Launching the Propeller Tool.....	124
16.0 CHAMELEON INTER-PROCESSOR ARCHITECTURE OVERVIEW.....	128
16.1 Master Control Program (MCP).....	129
16.2 Selecting the Drivers for the Virtual Peripherals.....	131
16.2.1 Complete Data Flow from User to Driver	132
16.3 Remote Procedure Call Primer (Theory).....	133
16.3.1 ASCII or Binary Encoded RPCs	134
16.3.2 Compressing RPC for More Bandwidth.....	135
16.3.3 Our Simplified RPC Strategy	135
16.4 Virtual Peripheral Driver Overview	136
16.4.1 Normalization of Drivers for Common RPC Calls in Future.....	137
17.0 CHAMELEON AVR API OVERVIEW	137
17.1 System Library Module.....	142
15.1.1 Header File Contents Overview	142
17.1.2 API Listing Reference.....	143
17.1.3 API Functional Declarations.....	144
18.0 UART AND RS-232 LIBRARY MODULE PRIMER	145
18.1 The Architecture of the UART API Library and Support Functionality.....	146
18.2 Header File Contents Overview	147
18.3 API Listing Reference	148
18.4 API Functional Declarations	148

19.0 SPI AND I²C LIBRARY MODULE PRIMER	152
19.1 SPI Bus Basics	153
19.1.1 Basic SPI Communications Steps	155
19.2 I ² C Bus Basics.....	155
19.2.1 Understanding I ² C Bus States.....	156
19.3 Header File Contents Overview	158
19.4 API Listing Reference	160
19.5 API Functional Declarations	161
20.0 NTSC LIBRARY MODULE PRIMER	167
20.1 Sending Messages to the Propeller Directly	168
20.2 Header File Contents Overview	169
20.3 API Listing Reference	169
20.4 API Functional Declarations	169
21.0 VGA LIBRARY MODULE PRIMER	171
21.1 Header File Contents Overview	172
21.2 API Listing Reference	172
21.3 API Functional Declarations	173
22.0 GFX LIBRARY MODULE PRIMER	175
22.1 GFX Driver Architectural Overview	175
22.1.1 GFX Driver Register Interface.....	180
22.2 Header File Contents Overview	180
22.3 API Listing Reference	184
22.4 API Functional Declarations	186
23.0 SOUND LIBRARY MODULE PRIMER.....	194
23.1 Header File Contents Overview	195
23.2 API Listing Reference	195
23.3 API Functional Declarations	195
24.0 KEYBOARD LIBRARY MODULE PRIMER	197
24.1 Header File Contents Overview	197
24.2 API Listing Reference	198

24.3 API Functional Declarations	199
25.0 MOUSE LIBRARY MODULE PRIMER.....	200
25.1 A Brief Mouse Primer	201
25.2 Header File Contents Overview	201
25.3 API Listing Reference	201
25.3 API Functional Declarations	202
26.0 PROPELLER LOCAL I/O PORT MODULE PRIMER.....	203
26.1 Header File Contents Overview	203
26.2 API Listing Reference	203
26.3 API Functional Declarations	204
27.0 CHAMELEON HANDS-ON PROGRAMMING TUTORIALS AND DEMOS	205
27.1 Setup to Compile the Demos and Tutorials.....	205
27.1.1 Differences Between the AVRStudio and Arduino Demos and General Porting Strategies.....	205
Include Files.....	205
Renaming the main() Function and setup().....	206
Serial I/O Libraries	208
27.1.2 Setup for AVRStudio Version of Demos	209
27.1.3 Setup for Arduino Version of Demos.....	209
Copying the Sketches from DVD	210
27.1.3 Setting the Chameleon Hardware Up	210
28.1 Graphics Demos.....	211
28.1.1 NTSC Printing Demo.....	211
28.1.2 NTSC Glowing Top/Bottom Overscan Demo	213
28.1.3 NTSC Smooth Scrolling Tilemap Demo.....	215
28.1.4 VGA Printing Demo	217
28.1.5 Dual NTSC / VGA Printing Demo	219
28.1.6 VGA Star Field Demo.....	220
29.1 Sound Demos	222
29.1.1 Sound Demo	223
30.1 Input Device Demos.....	225
30.1.1 Keyboard Demo.....	225
30.1.2 Mouse "ASCII Paint" Demo.....	228
31.1 Serial, FLASH, and Port I/O Device Demos	230
31.1.1 Propeller Local Port LED Blinker Demo	231
31.1.2 Serial RS-232 Communications Demo.....	233
31.1.3 FLASH Memory Demo (with XModem Protocol Bonus Example)	238
X-Modem Protocol	239
Sending X-Modem Files from the PC	243
32.1 Native Mode / Bootloader mode	248
33.1 Developing Your Own Propeller Drivers.....	249
33.1.1 Adding SPIN Driver Support for the Status LED	250
31.1.2 Adding AVR Support at the Client/Master Side	252

34.1 Advanced Concepts and Ideas.....	254
35.1 Demo Coder Applications, Games, and Languages	255
35.1.1 Chameleon BASIC by David Betz.....	255
35.1.2 Crate-It! by JT Cook	255
Epilog - From Intel 4004 to the Multiprocessing/Multicore Chameleon.....	256
APPENDIX A: SCHEMATICS	257
APPENDIX B: ATMEL AVR 328P PINOUT	259
APPENDIX C – BOARD LAYOUT AND I/O HEADERS	260
APPENDIX D - USING THE AVR IN "STAND-ALONE" MODE.	261
APPENDIX E - USING THE PROPELLER IN "STAND-ALONE" MODE.	261
APPENDIX F – PORTING HYDRA AND PARALLAX DEVELOPMENT BOARD APPLICATIONS TO THE CHAMELEON.....	262
APPENDIX G - RUNNING ON THE MAC AND LINUX.	262
APPENDIX H – OVERCLOCKING THE AVR AND PROPELLER.....	263
H.1 Overclocking the Propeller Microcontroller.....	263
H.2 Overclocking the AVR328P Microcontroller.....	263
APPENDIX I: ASCII / BINARY / HEX / OCTAL UNIVERSAL LOOKUP TABLES	264
APPENDIX J: ANSI TERMINAL CODES.....	266
Examples	267

0.0 Introduction and Origins

Welcome to the **Hardware/Programming** Manual for the **Chameleon AVR 8-Bit Development Kit**. This is a light-hearted, non-censored, no-holds-barred manual. If you find technical errors or have comments, simply write them down in a file and please send them to us at support@nurve.net or post on our forums and we will continue to merge them into this working document.

The document contains two main sections –

- **The Hardware Manual** - This is a fast and furious circuit description of the **Chameleon AVR 8-Bit Board's** dual processor implementation around the **Atmel AVR 328P** and **Parallax Propeller** processors. The Atmel 328P chip is 8-bit, single cycle instruction (in most cases) supporting 16-bit math and multiplication. It has 32K of FLASH and 2K of SRAM. We are running it at 16 MHz on the Chameleon, but its capable of running 20 MHz nominally and beyond when over clocked. The Parallax Propeller chip is used for media processing and runs at 80Mhz per core with 8 cores. Each instruction takes 4 clocks thus the Propeller runs at 20 MIPS per core nominally. It has 32K of SRAM and 32K ROM.
- **The Programming Manual** – This is the nitty gritty of the manual and has examples of programming graphics, sound, keyboard, mice, I/O, etc., and explains how the processors work together along with their various tool chains.

NOTE

The Chameleon PIC 16-Bit is very similar to the Chameleon AVR 8-Bit. In fact, the processor was removed from my originally design and replaced by the PIC 16 and the I/O interfaced re-connected, so if you learn one system you learn them both more or less.

The Chameleon's were designed to be application boards to solve real world problems that you might have day to day in your designs, engineering classes, or work. The Chameleons philosophy is to be very powerful, but very simple. I have always been very interested in graphics and media processing, however, its is very hard to get a simple embedded system or development board to output NTSC, PAL, VGA, etc. Thus, I set out to design something that wasn't a development or educational kit like our other products, but was something you just "use" to solve problems that is also good at **displaying** information and **interfacing** to user input devices. Nonetheless, I wanted the Chameleons to be able to do graphics, produce sound, interface to keyboards and mice and be as small as a credit card. There are a number of approaches to this design, but with multiprocessing and multicore so popular, I thought it would be interesting to use **both** concepts in a product. The Chameleon is the culmination of these ideas. To create a product that with very little coding you could output NTSC/PAL graphics, VGA graphics, read keyboards, and mice as well as communicate with serial ports and perform digital and analog I/O.

The selection of the processors was difficult as it always is. We have to balance price, performance, user base, flexibility and a number of other factors. Additionally, since the Chameleon is a dual processor design, I had to think of a clean way to interface the processors such as shared memory, SPI, I²C, etc. Taking all those factors into consideration, I decided to use the **Atmel AVR AT328P** processor for the Master and the **Parallax Propeller** chip for the Slave. The PIC version uses a PIC24 16-bit processor, but the idea is the same.

Thus, the idea is that the AVR (or PIC) with its huge fan/user base will be the ring leader, you code in C/ASM on the AVR processor and then over a SPI interface you send commands to the Propeller chip (which is running a SPI driver) and then issues the commands from the AVR to various processors (more on this in the Architecture Overview). This is the perfect fusion and balance of the two processors. From the AVR's perspective the Propeller chip is simply a media processor, the programmer need not know a single thing about the Propeller if he doesn't want to. On the other hand, this setup is incredibly flexible. The Propeller can run any set of drivers on it that we wish and thru an agreed on communications protocol the AVR can control these drivers remotely and get work done.

Finally, when I started designing the AVR version of the Chameleon a couple years ago, I kept hearing about this product called the Arduino, finally I went to the site and checked it out, but was surprised that there was no "Arduino" per se. The Arduino isn't a product so much as it's an ideal or methodology of software and tools. There is nothing but a single AVR ATmega 168/328 on each Arduino board there is no extra hardware to do anything. So I looked deeper and found out the Arduino guys came to the same conclusion I did about embedded system programming – its too hard!

The tool chains area nightmare, installation has too many steps, and its just too complicated for newbie's that just want to play with something. So what they did was not concentrate on hardware, but concentrate on **software**. They took the AVR tool GNU GCC tool chain and covered it up with a Java application. Therefore, from your perspective as a programmer you don't need to know anything about AVRStudio, GNU GCC, WinAVR, etc. All you need to know is how to type in the

Arduino text editor and press a couple buttons. Now, programming AVR's (and PICs) is a challenge as well. You normally need an ISP (in circuit programmer) and to use another tool. Thus, you first write your code with AVRStudio, compile it, then you have a binary that you download with yet another software tool and a physical ISP programmer.

This again is a nightmare and a mess for newbies, so the Arduino guys also integrated this into the tool and supplied a "bootloader" on the AVR chips their Arduino's ship with, therefore all you need is a serial connection to the AVR chip on the Arduino hardware and the Arduino tool can initiate communications and download to the FLASH memory.

Therefore, with a single AVR chip programmed with the Arduino bootloader and the Arduino software tool, you can write code and immediately download to the chip. Moreover, they ported the tool to Mac and Linux, so the experience from Windows, to Mac OS X to Linux is more or less the same (once you have the tool installed). This is good since AVRStudio does NOT even work on Mac or Linux!!!

Anyway, I knew I had to try and be compatible with the Arduino tool since it would be very cool if people could use Arduino software and tools on the Chameleons. Of course, I was using the new Atmel 328P in my designs and Arduino was still using the Atmel 168, but I hoped they would upgrade in time and they did! So now the Chameleons will run the Arduino tool chain and you can use Chameleons as supped up Arduinos'. Of course, our boards are physically different and our headers are slightly different, but more or less with a little work any program designed for the Arduino can be ported to the Chameleon in a matter of minutes. Then you get the power of the Propeller media processor to display text, graphics, read keyboards, mice, and make sounds, etc.

As an example, imagine you have an Arduino servo controller program and it uses one of the PWM ports on the AVR 328P. Since we are using the same chip, the Chameleon has the same PWM ports and we export the pins out to the digital and analog I/O headers just like the Arduino. But, now imagine that instead of just sending a signal on the digital ports to move the servo right or left, you can use a PS/2 keyboard? Or mouse? And now imagine that you can display the position and angle on a VGA monitor and even draw the servo in color graphics on the screen!

So the Chameleon is like a super charged Arduino with support for graphics, audio, keyboard, mice and much more! And the cool thing is that the code you have to write to support the added features and capabilities of the Chameleon are usually just a few lines that call API functions to communicate with the Parallax Propeller media processor.

NOTE

The Chameleon PIC 16-Bit is very similar to the Chameleon AVR 8-Bit, but porting the Arduino software to the Chameleon version has not been done yet. However, all the pieces are there. There is a GNU GCC compiler for the PIC24, and other than that its just a matter of porting the Arduino libraries and runtime software template. The Java application is chip agnostic and is nothing more than a Java app. So hopefully, someone ports the Arduino software to run on the Chameleon PIC as well, so all us PIC users can experience the ease of use and multiplatform support as well.

IMPORTANT!

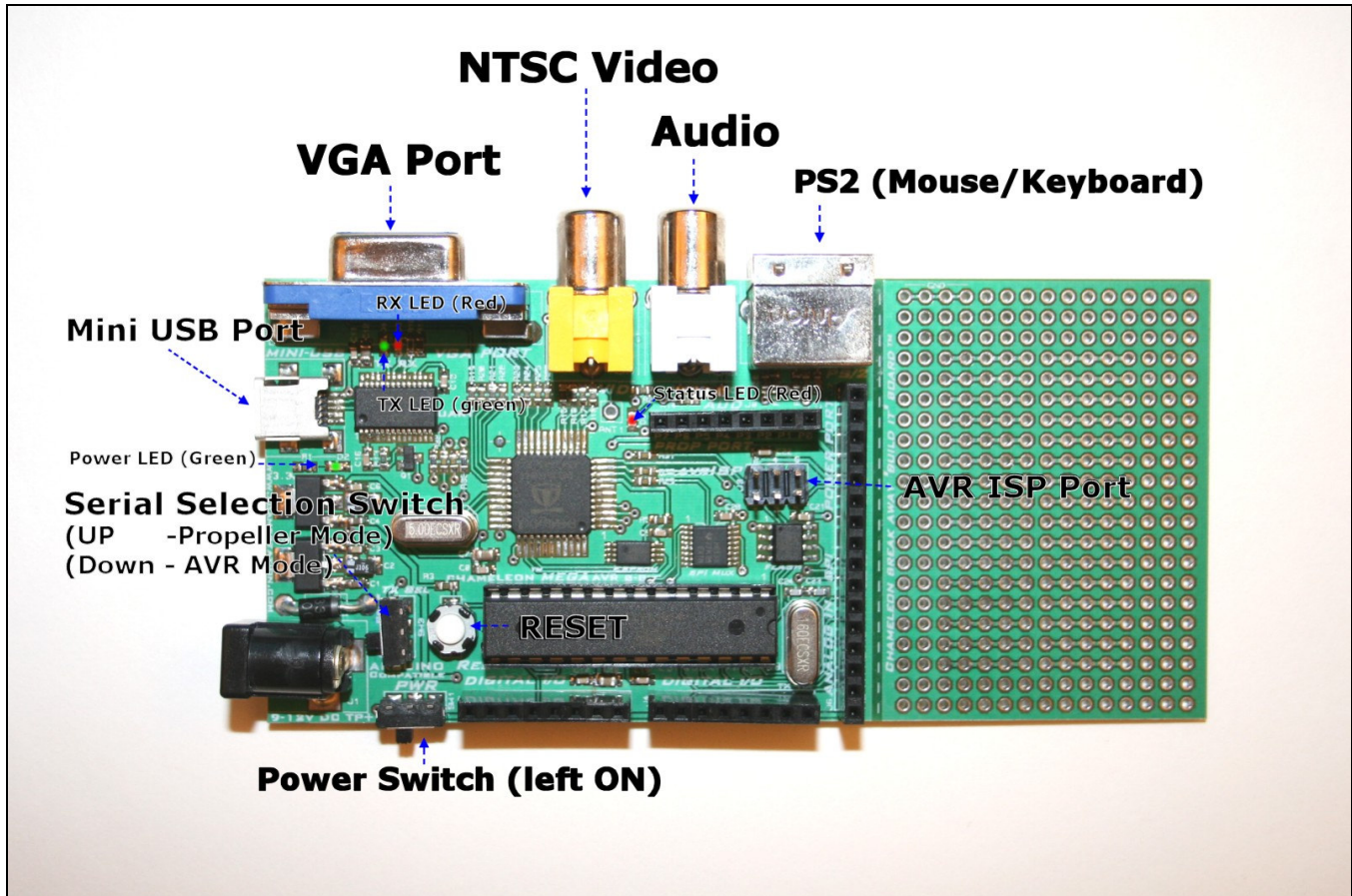
Last, but not least, peruse the entire manual **before** doing anything. There are a few items that are embedded in the middle or end that will help you understand things, so best to read the whole thing first **then** go ahead and start playing with the hardware and programming.

So without further ado, let's begin!

1.0 Architectural Overview

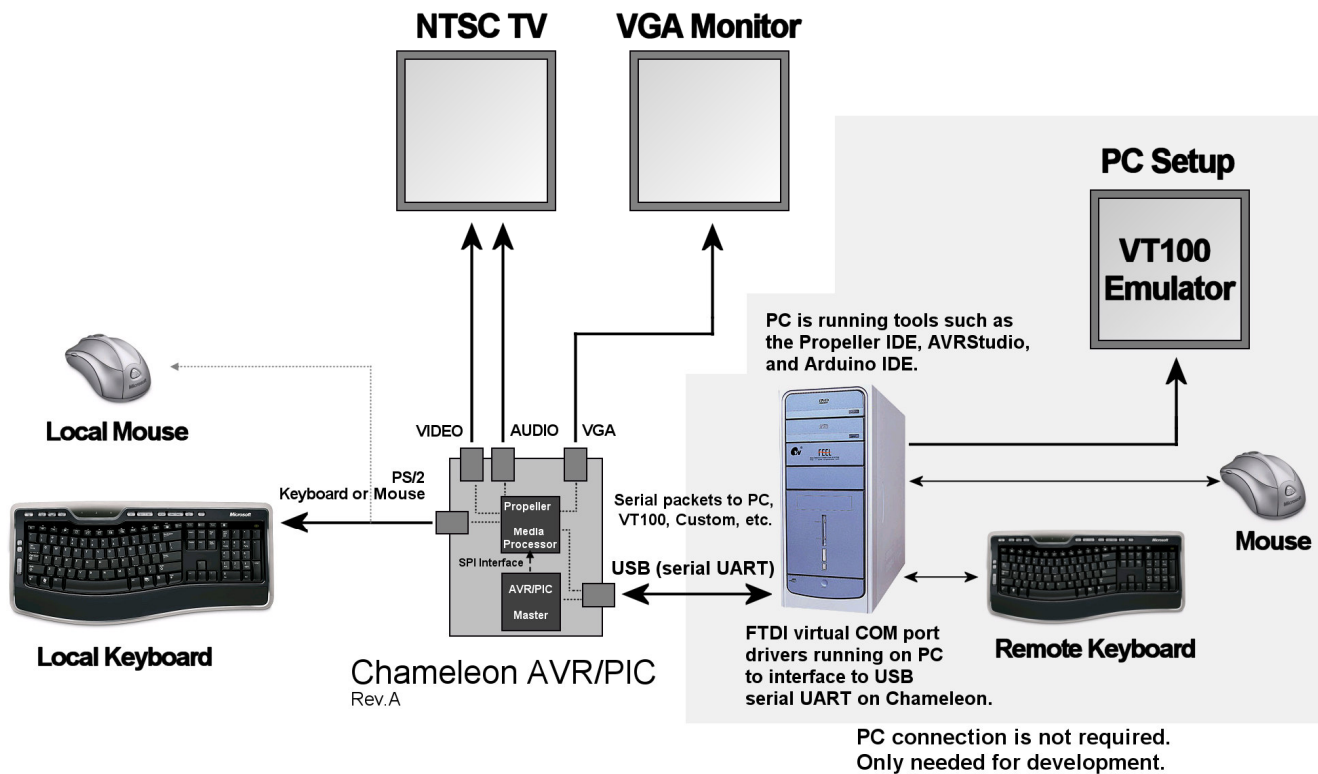
Part I – Hardware Manual

Figure 1.0 – The Chameleon AVR 8-Bit.



The **Chameleon AVR 8-Bit** or simply the “**Chameleon AVR**” is developed around the **Atmel 8-Bit Mega AVR 328P28-Pin DIP QFP**. Figure 1.1 shows an image of the Chameleon AVR 8-bit annotated. The Chameleon AVR has the following hardware features:

- 28-Pin DIP Package version of the AVR 328P, runs at 16 MHz with a maximum speed of 20 MHz suggested by Atmel. However, it can easily be over clocked to 25-30 MHz.
- Parallax Propeller multicore processor, 32K RAM, 32K ROM, QFP 44-Pin package running at 80 MHz per core with 64K Byte serial EEPROM for program storage (2x required memory).
- RCA Video and Audio Out Ports.
- HD15 Standard VGA Out Port.
- Single PS/2 Keyboard (and mouse) Port.
- Single 9V DC power in with regulated output of 5.0V @ 500mA and 3.3V @ 500 mA on board to support external peripherals and components (the AVR 328P is 5V, Propeller is 3.3V).
- Removable XTALs to support faster speeds and experimenting with various reference clocks.
- Expansion headers that exposes I/O, power, clocking ,etc. for both the AVR and Propeller.
- USB UART built in with Mini-B connector.
- 6-PIN ISP (**In System Programming**) interface compatible with Atmel **AVRISP MKII** as well as other 3rd party programmers.
- 1 MByte of SPI FLASH storage chip that can be used as local storage or a full FAT file system.
- “Break Away” experimenter prototyping board that can be snapped off for prototyping and adding extra hardware to the Chameleon.

Figure 1.1 – Chameleon system level diagram.

Referring to Figure 1.1, this is a system level diagram of the relationship of the Chameleon to all the system components. First, everything to the right is optional and only needed for programming the Chameleon. Once the Chameleon is programmed it's a stand alone application that you can put anywhere you like. With that in mind, let's take a quick look at how things work. To begin with, there are two processors on the Chameleon AVR, the AVR ATmega 328P and the Parallax Propeller chip. The AVR chip is used as the **"master"** processor. You write your application on there in C/C++, ASM, or with the Arduino tool and then using a very simply API you send **"messages"** to the Propeller chip. The messages direct the Propeller chip to perform tasks. These tasks can ultimately be anything, but for now, we have set the Propeller up so it can generate NTSC and VGA graphics, audio, and read keyboards and mice. So the Propeller does all the work for you, the AVR does very little, thus freeing the AVR to take the role as master controller.

The messages sent to the Propeller are transferred over a high speed **SPI** (serial peripheral interface) link. The AVR has SPI hardware built in, so sending bytes to the Propeller is as simple as setting up a few registers and writing some bytes. However, the Propeller has no SPI hardware, so we had to write drivers that emulate the SPI protocol with software. This means there are limits to the speed you can send SPI traffic as well as the software SPI drivers are not very robust, they are just **"starter"** drivers for you, I suggest you improve them.

As an example of how the system works, let's say you have a AVR C/C++ program you developed with AVRStudio or maybe the Arduino tool. It has a A/D convertor and measures temperature. The temperature is then sent out to a crude LCD screen and looks ugly. Also, there are some controls for the program that you must use some external push button switches to set, but it would be nice if you had a keyboard or mouse for user input. This is no problem for the chameleon!

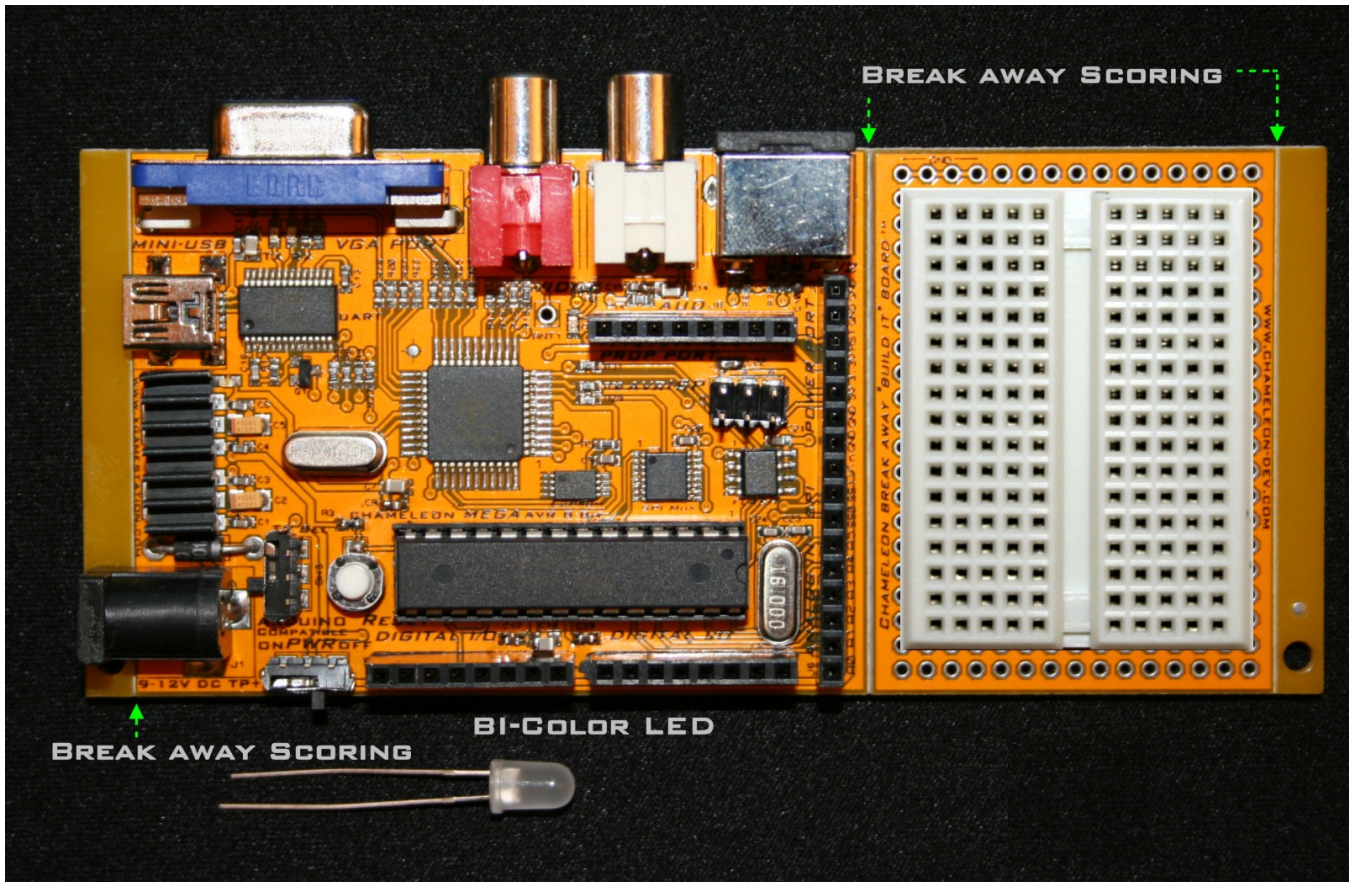
You would take you original program, compile it for the Chameleon, then add a few lines of code from our NTSC or VGA API that command the Propeller to draw text on the screen. Then you can print out your temperature information nicely on a little NTSC/VGA monitor. Moreover, you can write some simple GUI controls, so with the mouse or keyboard you could make command selections on the NTSC/VGA screen! Very slick! As another example, you can use the Chameleon's built in USB serial port to communicate to the PC. So say you want to control some motors, or do some A/D, but you want the PC to take that information and plug it into some larger application. Normally, to get something interfaced with the PC is nightmare in the era or post legacy devices where only USB connectors exist and there are no parallel or DB9 serial ports. But, with the Chameleon its easy, you just write your driver code on the Chameleon, use the serial UART library to communicate to the PC and send the information. Moreover, you can display status and what's going on thru one of the local video ports of the Chameleon, so even if the PC is remote the local Chameleon can display what's going on and what its being told to do by the PC.

The possibilities are limitless with the Chameleon. We wanted to take the work out of generating video, audio, reading keyboards and mice out of the picture for you, but at the same time insulate you from what's going on under the hood – unless you really want to know.

Next, let's inventory your Chameleon AVR 8-Bit Kit.

1.1 Package Contents

Figure 1.2 – Chameleon AVR 8-Bit kit contents (DVD not shown).



You should have the following items in your kit, referring to Figure 1.2.

- Chameleon AVR 8-Bit System Board.
- DVD ROM with all the software, demos, IDE, tools, and this document.
- White solderless experimenter board to be affixed to “break away” experimenter prototyping area on Chameleon if desired.
- (1) Bi-color Red/Green LED.

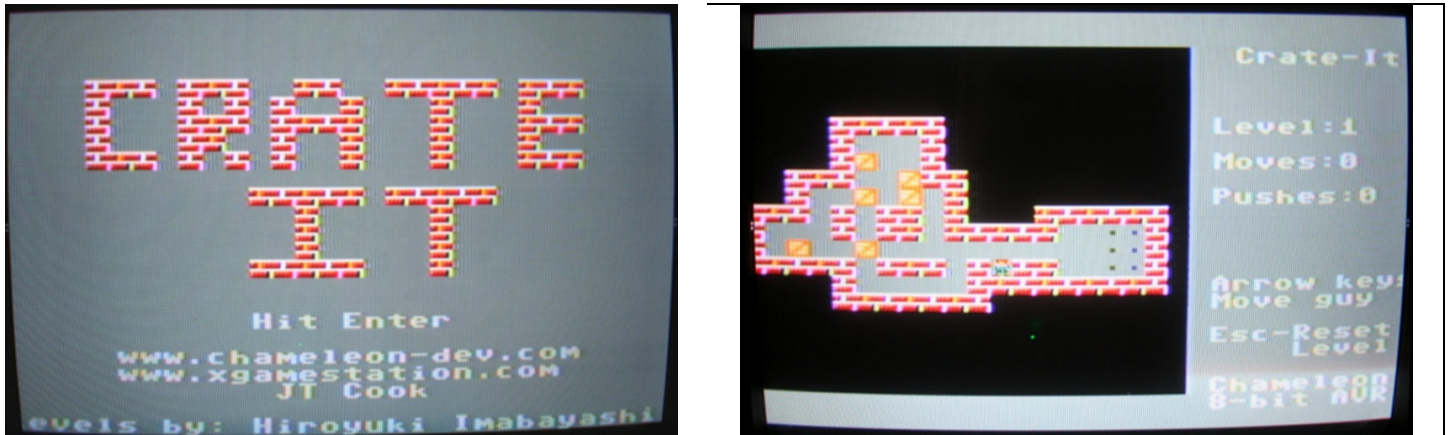
Not Included

- 9V power supply with tip +, ring -, 2.1mm barrel.
- Mini B USB cable to go from PC to Chameleon AVR.
- AVR MK II ISP Programmer.

The Chameleon will run off the USB cable, so you do not require a wall adapter for development. Also, the Chameleon AVR is pre-loaded with the Arduino bootloader, therefore, if you use the Chameleon in “Arduino mode” only you will **not** need an **Atmel AVR ISP MK II** programmer, but we recommend one so that you can use AVR Studio to write more advanced C/C++ and ASM programs and to have more control over the platform. Additionally, if you inadvertently damage the FLASH'ed bootloader you will have to reload it. Thus, you need an AVR ISP MK II or similar programming hardware tool in the event you need to re-FLASH the bootloader.

1.2 Chameleon AVR "Quick Start" Demo

Figure 1.3 – The Console demo running – Crate-it!



The Chameleon AVR 8-bit is pre-loaded with a demo programmed into the AVR's FLASH memory, a screen shot is shown in Figure 1.3 that illustrates images of both the NTSC and VGA monitors during the demo. We will use this to test your system out. The following are a series of steps to try the demo out and make sure your hardware is working.

IMPORTANT!

We are going to assume you are running Windows XP or similar. Linux and Mac systems are similar for this simple demo experiment. However, Linux and Mac detect and install the FTDI USB drivers differently which we will cover in their respective setups in the Appendices.

First Things First...

Before we plug the Chameleon in and test it, a couple things to pay attention to. First, the Chameleon can be powered by **either** a 9V DC adapter or draw power from the USB port connection. If you have a 9V DC adapter then you can use it for power. However, to "talk" to the Chameleon you are going to need a mini-B USB cable no matter what. Therefore, when we get to the power setup in **Step 3**, you can either plug in both the 9V adapter and the USB cable or just the USB cable.

If you plug in the USB port cable then the PC will detect a new USB device and if you have Windows SP 2/3 then it should install the FTDI drivers itself. If it doesn't, it's alright we will perform driver installation later during the software installation phase. Point is, the moment you plug the Chameleon into the Windows PC, plug and play is going to detect the new USB device of the serial UART built into the Chameleon, so don't panic!

Step 1: Place your Chameleon AVR on a flat surface, no carpet! Static electricity!

Step 2: Make sure the power switch at the front is in the **OFF** position, this is to the **RIGHT**.

Step 3: Plug your 9V DC wall adapter in and plug the 2.1mm connector into the bottom right power port on the Chameleon AVR. If you do not have a power adapter then you can power the Chameleon AVR with the USB port. Plug a USB mini-B cable from your PC to your Chameleon AVR. We suggest using a **powered** USB hub since the Chameleon will draw a lot of power.

Step 4: Insert the A/V cable into the **ORANGE** (video) and **WHITE** (audio) port of the Chameleon AVR located top-right of the board and then insert them into your NTSC/Multi-System TV's A/V port.

Step 5: Plug a PS/2 compatible keyboard into the PS/2 port on the Chameleon (top right).

Step 6: Turn the power **ON** by sliding the ON/OFF switch to the **LEFT** then hit the **Reset** button as well (next to the AVR chip).

Step 7: The demo will start immediately, it's a little block pushing game called "Crate-it!". Use the keyboard arrow keys to move your character and push blocks around and move them to their resting positions (usually on the opposite side of the screen).

You should see something like that shown in Figure 1.3. The actual program that is loaded into the AVR is located on your DVD here:

DVD-ROM:\ CHAM_AVR \ SOURCE \ CRATE_IT_V010.C

The Arduino version is located in the \Sketches directory on the DVD here:

DVD-ROM : \ CHAM_AVR \ TOOLS \ ARDUINO \ SKETCHES \ CHAM_AVR_CRATE_IT

Both versions are nearly identical, the Arduino version has simply been converted into an Arduino Sketch.

Of course, the demo needs many other driver and system files to link with, but we will get to this latter when we discuss the installation of AVR Studio/WinAVR/Arduino and the tool chain in general for C/C++ and ASM programming.

The demo took about a week to develop and was written by one of our Demo coder's **JT Cook** to see what he could do with the Chameleon in a week and rely 100% on our drivers. The results are pretty amazing, and the cool thing is the game literally was ported to the PIC version in a matter of minutes. So as an extra bonus by leveraging the Propeller to do all media processing, the AVR/PIC processor running the applications (or games) is usually in pure C/C++ and since the interface APIs to the Propeller are the same you get the exact same experience when you port an applications from the Chameleon AVR to PIC and vice versa. Of course, the PIC version is faster and has more memory – but, use AVR users like it that way – a challenge!

Hit the **Reset** button over and over and the demo will reset and reload immediately, If the system ever locks up (rare, and always due to bad code), then simply hit **Reset** a few times or cycle the power.

Playing Crate-It!

Crate-it! is your standard block pushing game where you want to get the objects (blocks) from one side of the screen to resting positions on the other side. The problem is that if a block hits up against an immovable object, they can get stuck and you are out of luck. These types of games are very easy to develop, but hard to play. You have to really think about what order you move blocks and you have to be weary about getting stuck.

This version requires an NTSC monitor, and the local PS/2 keyboard plugged in. Also, make sure to have the audio port connected to your TV, there is sound.

To play, simply move the character with the arrow keys and push the blocks around. The first level has the blocks on the left side and their resting positions on the right side, so you have to "push" the moveable blocks from the left side to the right side without getting stuck, or putting yourself in a corner. The first level has 6 blocks that must be moved.

This concludes the **Quick Start** demo.

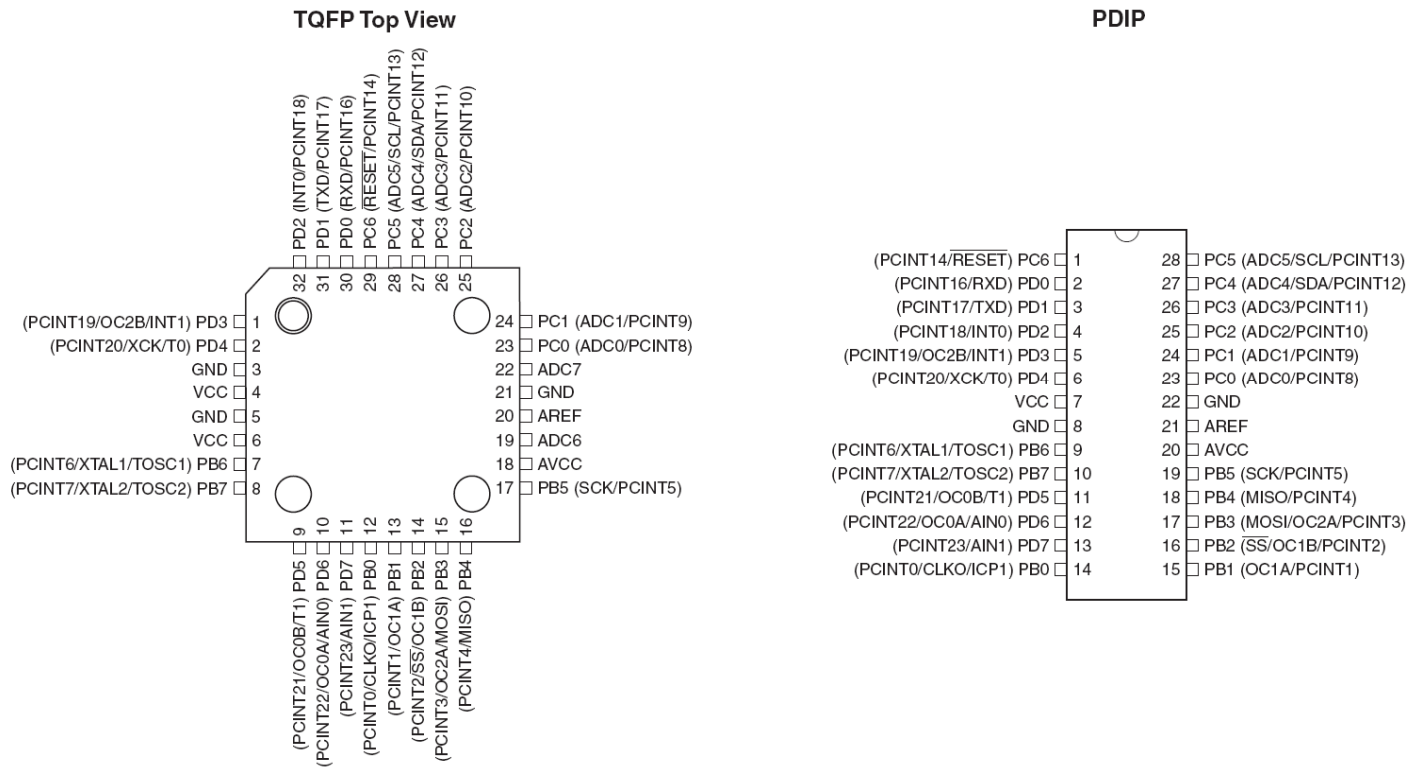
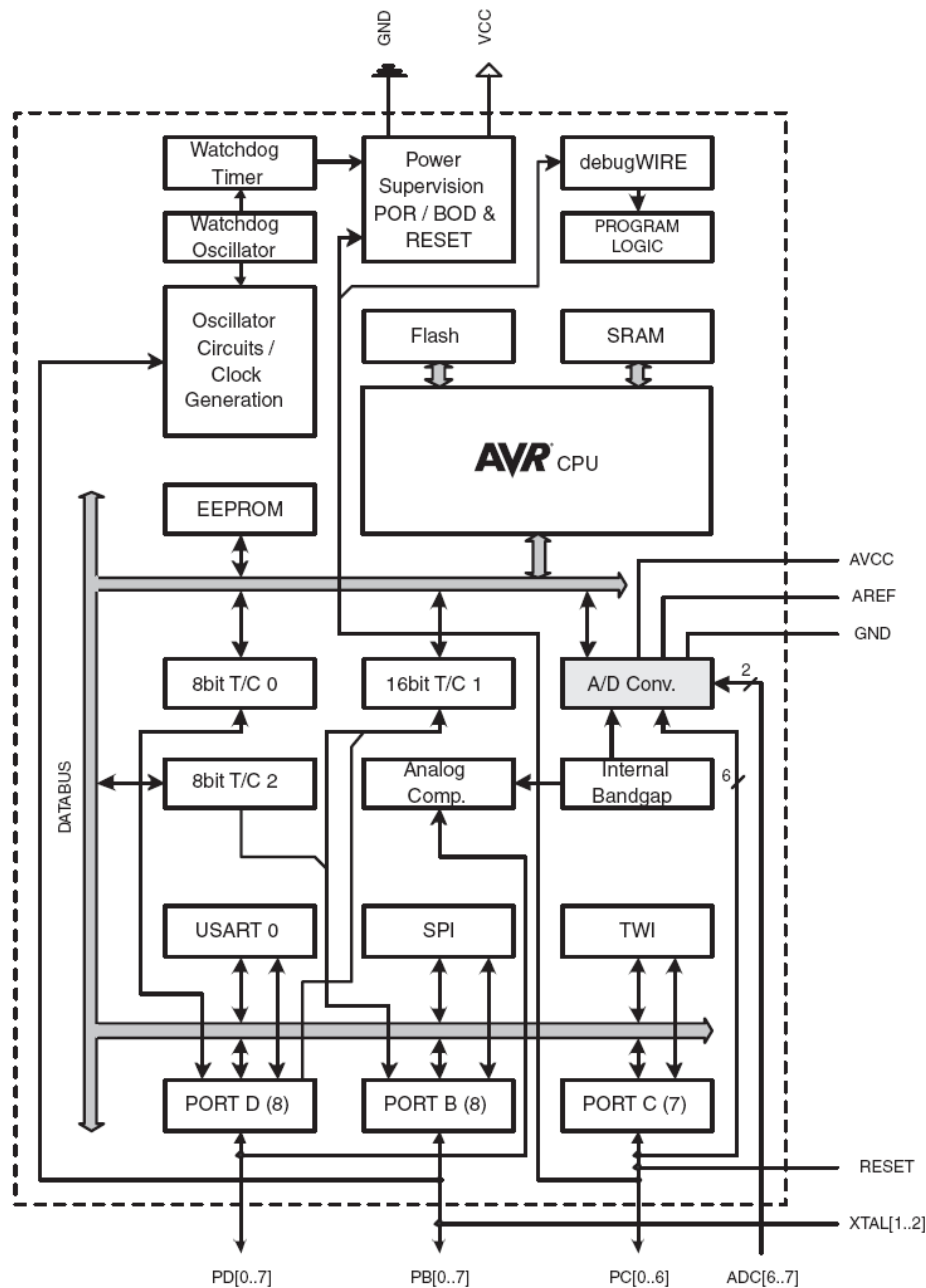
Figure 1.4 – The Atmel Mega AVR 328P packing for 28-Pin TQFP and PDIP packages.

Figure 1.5 – The Atmel Mega AVR 328P architecture block diagram.

1.3 The Atmel Mega AVR328P Chip

The Atmel AVR 328P (P stands for Pico Power) comes in a number of packages including 28-pin **DIP** (dual inline package). We are using the Mega AVR 328P DIP package in the Chameleon. Figure 1.4 shows the packaging and pin out of the chip. The AVR 328P was designed as a general purpose, high performance microcontroller with an 8-bit data path, single clock execution on many instructions, as well as support for 16-bit operations and built in multiply instructions. The chip has a large FLASH memory of **32K** Bytes, but a smaller 2K Byte SRAM which makes a lot of RAM intensive applications challenging; however, the Chameleon is designed for control applications and 2K of RAM should be more than enough for anything you need. Moreover, the 328P has 2x the amount of FLASH as the 168 which successfully was used in thousands if not tens of thousands of Arduinos and everyone did just fine. For complete details on the processor, please review the datasheet located on the DVD-ROM here:

DVD-ROM: \ cham_avr \ docs \ datasheets \ **ATMega48_88_168_328_doc8161.pdf**

Table 1.2 below shows the various differences between the 48, 88, 168, and 328 variants for reference. Additionally, there is a useful EEPROM memory as well that can be used as a little disk drive or memory storage for whatever purposes. Of

course, the AVR 328P allows re-writing to the FLASH as well, so unused portions of FLASH can be used for storage; however, it's not ideal to constantly re-write FLASH since there is a limit to the number of times it can be re-written; 100,000 give or take. Additionally, you can use the 1MByte SPI FLASH on the Chameleon as well for storage.

NOTE

FLASH memories typically have a maximum number of times they can be written; something in the range of 10,000 to 100,000. This doesn't mean that at 10,001 or 100,001 the memory won't work, it just means the erase cycles and write cycles may take longer to get the memory to clear or write. And this then degrades further as the write/erase cycles persist. Thus, if you were to code all day and re-write your FLASH 100x times a day, then at 100,000 re-write cycles, you would have 3-4 years before you ever saw any problems. On the other hand, if you write code to use the FLASH as a solid state disk and constantly re-write the memory 10,000x a run, you can see how quickly you might degrade the memory. Thus, use the EEPROM for memory you need to update and still be non-volatile and save the life of the FLASH.

Table 1.2 - Differences between ATmega 48P, 88P, 168PP, and 328P.

Device	Flash	EEPROM	RAM
ATmega 48P	4 K Byte	256 Bytes	512 Bytes
ATmega 88P	8 K Byte	512 Bytes	1 K Byte
ATmega168P	16 K Byte	512 Bytes	1 K Byte
ATmega328P	32 K Byte	1 K Byte	2 K Byte

Note: The "P" suffix simply means "Pico Power" and has nothing to do with the chip operation or functionality. The Pico power version is identical to the non-pico power for our purposes and I will use them interchangeably.

Figure 1.5 shows the AVR 328P architecture in block diagram form and Table 1.3 lists the pins and their function for the AVR 328P.

NOTE

Since the AVR 328P has so many internal peripherals and only a finite number of pins, many functions are multiplexed on the I/O pins such as SPI, I2C, UARTs, A/D, D/A, etc. Thus, when you enable one of the peripherals they will typically override the I/O functionality and take on the special functions requested. However, when you don't enable any peripherals each I/O pin is a simple I/O pin as listed in Table 1.3.

Table 1.3 – The AVR 328P general pin descriptions.

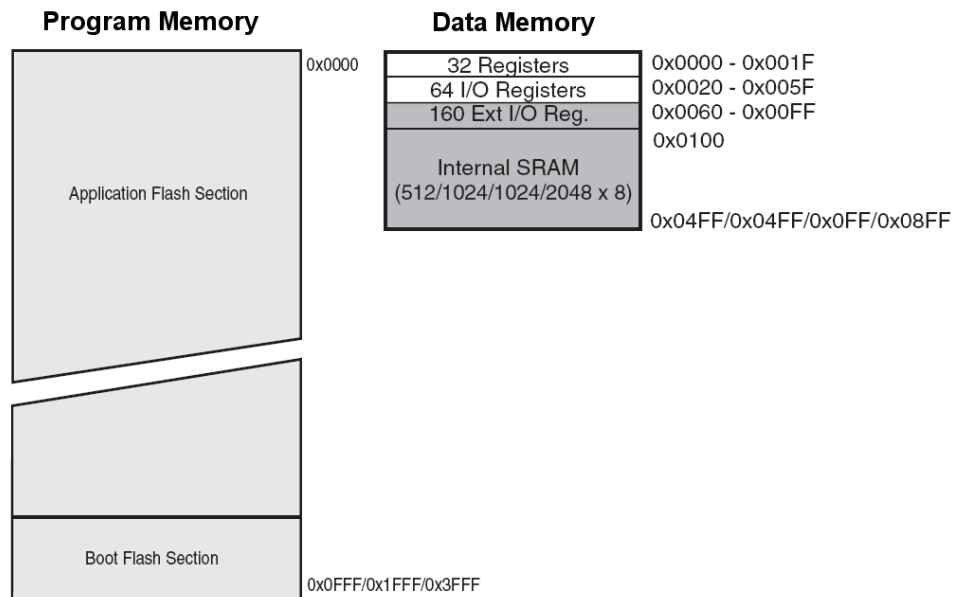
Pin Group	Description
Port B (PB7:PB0)	Port B is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port B output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port B pins that are externally pulled low will source current if the pull-up resistors are activated. The Port B pins are tri-stated when a reset condition becomes active, even if the clock is not running. Port B also contains the SPI interface signals as well as the clocking input pins.
Port C (PC6:PC0)	Port C serves as analog inputs to the Analog-to-digital Converter. Port C is an 7-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port C output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port C pins that are externally pulled low will source current if the pull-up resistors are activated. The Port C pins are tri-stated when a reset condition becomes active, even if the clock is not running. Note: even though Port C only has 7-pins bonded out on the chip, the port register is still 8-bits wide, but the 8 th bit is simply ignored.
Port D (PD7:PD0)	Port D is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port D output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port D pins that are externally pulled low will source current if the pull-up resistors are activated. The Port D pins are tri-stated when a reset condition becomes active, even if the clock is not running. Port D also contains the UART signals.

/RESET – Reset input. A low level on this pin for longer than the minimum pulse length will generate a reset, even if the clock is not running.
XTAL1 - Input to the inverting Oscillator amplifier and input to the internal clock operating circuit.
XTAL2 – Output from the inverting Oscillator amplifier.
VCC – Main power, 5V.
GND – System ground.
AVCC - AVCC is the supply voltage pin for the Analog-to-digital Converter. It should be externally connected to VCC, even if the ADC is not used. If the ADC is used, it should be connected to VCC through a low-pass filter.
AREF - This is the analog reference pin for the Analog-to-digital Converter.

The AVR 328P is a **8-bit RISC-like** architecture chip with instructions being either **16 or 32-bits** in size (mostly 16-bit). The memory model is a **"Hardware Architecture"** meaning that the data and memory are located in separate memories that are not addressed as a contiguous space, but rather as separate memories with different instructions to read/write to them. This allows faster execution since the same buses aren't used to access data and program space. Therefore, you will typically access SRAM as a continuous 2K of memory and program/FLASH memory is in a completely different address space as is EEPROM memory. Thus, there are 3 different memories that the AVR 328P supports. Additionally, the AVR 328P maps registers as well as all its I/O ports in the SRAM memory space for ease of access. Figure 1.6 show these memories.

TIP	Harvard as opposed to Von Neumann architecture are the two primary computer memory organizations used in modern processors. Harvard was created at Harvard University, thus the moniker, and likewise Von Neumann was designed by mathematician John Von Neumann. Von Neumann differs from Harvard in that Von Neumann uses a single memory for both data and program storage.
------------	--

Figure 1.6 – FLASH and SRAM memory layouts.



Referring to Figure 1.6, **Program Memory** is composed of both a **"Boot FLASH Section"** and the **"Application Section"**. The boot section holds boot ROM code and can be different sizes or disabled. The application section then holds the actual run-time code for the application. The Data Memory is stored in SRAM of course and is **2048** bytes in length. However, due to register space allocation there is a shift in the addresses; the first 256 byte addresses are used to access registers and system I/O then from address **[\$0100 - \$08FF]** is the 2K block of free memory. Not shown in the

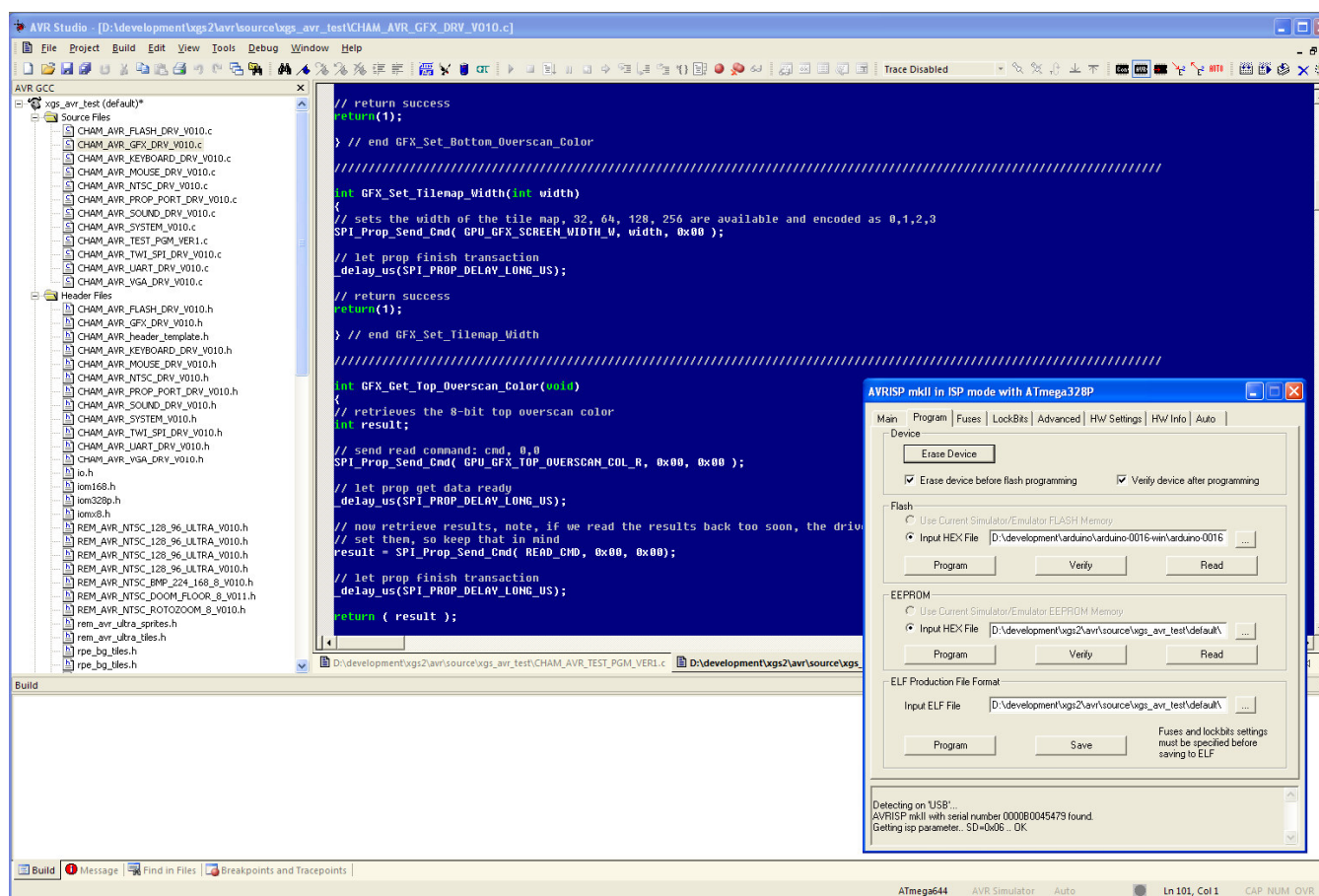
memory map is the “**stack**” which will need to go somewhere when you run your code, the C/C++ compiler sets this up for you. When programming in pure ASM, you would have to set the stack pointer appropriately

The AVR C/C++ compiler is based on the **GNU GCC** tool chain and thus is not the best optimizing compiler on the planet, but isn't bad. Hence, our approach will be to use C primarily for our coding and any time critical performance code you will want to write in assembly language if you need to. But, we want to rely on C as much as possible for ease of use. ASM should be used for drivers when necessary via APIs that can be called from C. Additionally, the Arduino tool chain uses C/C++ as well. Even though the Arduino folks call the language “Processing”, its just good old C/C++!!!

NOTE

The tool of choice for native AVR development is of course **AVR Studio**. This tool was developed by Atmel and supports source level debugging, various programmer and ICE debuggers etc. However, the C/C++ compiler is a plug in based on GNU GCC called “**WinAVR**”. We will discuss the installation of the tool chain shortly, but keep in mind the separation. Additionally, we will be using straight “C” for coding. C++ is supported, but due to its overhead and lack of 100% support for embedded applications, we will avoid it. C is much more compatible with all embedded systems and C++ is just asking for trouble especially with the compiler.

Figure 1.7 – Atmel AVR Studio 4 in action.



AVR Studio is shown in Figure 1.7, it's a standard Windows application that is used to develop applications for the entire line of AVR microcontrollers. It's very similar to Visual C++, but not as polished, you will find little quirks with file management focus issues, etc. The IDE allows viewing of the various files and statistics as builds are performed. The IDE supports the built in Atmel assembler directly or GNU GCC. When in Atmel ASM mode then all your programs must be ASM files using Atmel ASM directives and syntax, they are assembled and linked and a final binary image is generated ready for download to the target (Chameleon in this case). This is **not** how we will use the tool in most cases. Typically, we will use it in GCC mode, so we can code in both C and ASM. However, when you use the GCC mode of operation you are using the entire GCC tool chain including GCC (the C compiler), GAS (GNU assembler), make, librarian and so forth. Luckily, all of this is handled for you automatically as are the creation of Make scripts and so forth.

But, when you use GCC and the GNU assembler GAS, you must use all GNU syntax which is fine for C, but a bit tedious for ASM. The Atmel assembler is a little cleaner in my opinion, but GAS is fine as well. In any event, when you create a new project for the Chameleon you will select GCC and then you can write .C or .S (asm) files and then will be compiled, assembled and linked together. More of the mechanics of this later, but that's the idea.

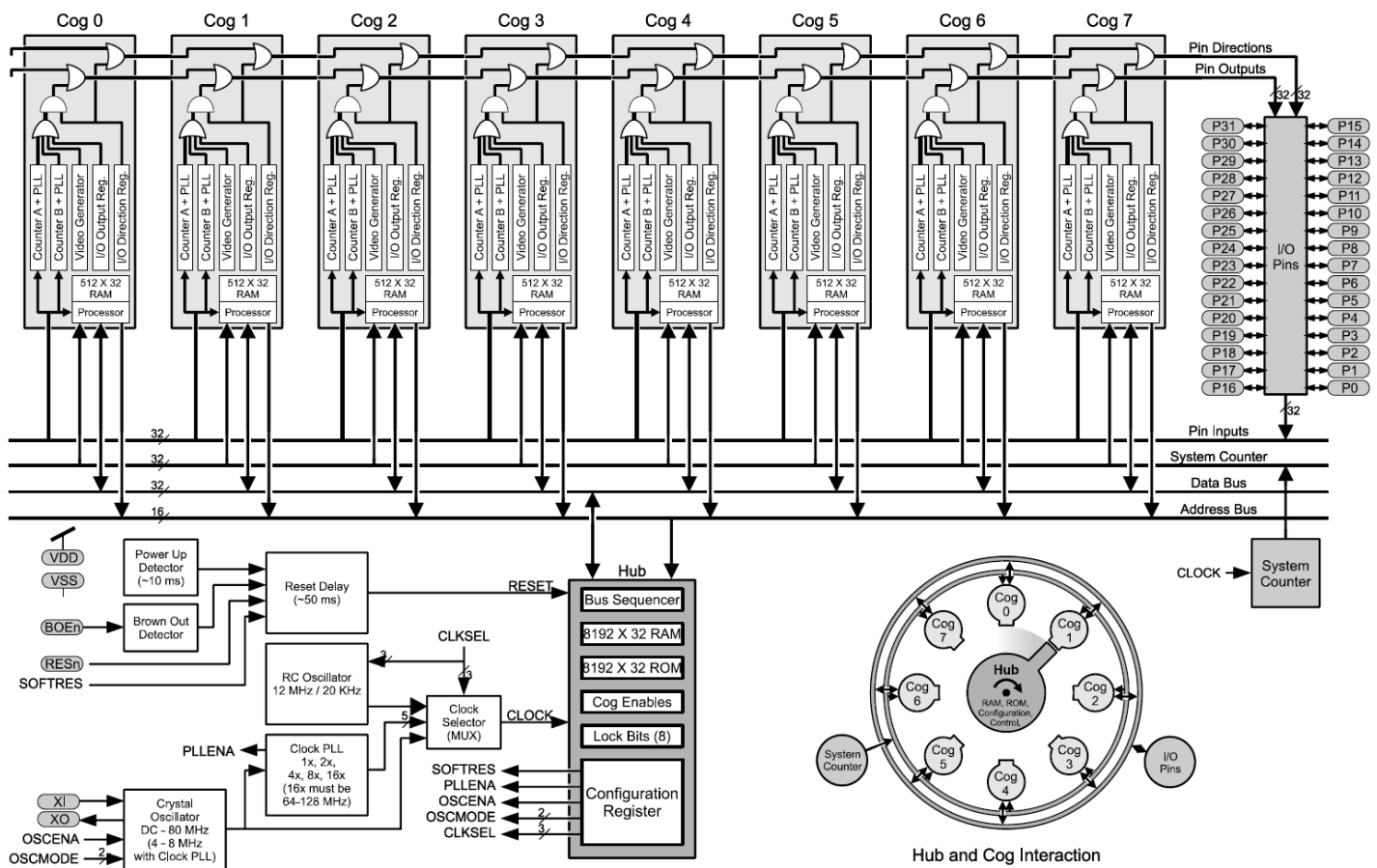
Additionally, you can use inline assembly with GCC, but the syntax is terrible as shown in the code snippet below which simply swaps a pair of integers:

```
asm volatile("mov __tmp_reg__, %A0" "\n\t"
"mov %A0, %D0" "\n\t"
"mov %D0, __tmp_reg_" "\n\t"
"mov __tmp_reg__, %B0" "\n\t"
"mov %B0, %C0" "\n\t"
"mov %C0, __tmp_reg_" "\n\t"
: "=r" (value)
: "0" (value)
);
```

As you can see there is a lot of pontification with syntax. If you are a 80xxx coder and have used the VC++ inline assembler or Borland for that matter you should be appalled at the above syntax.

Hence, I personally avoid the inline assembler, but if you want to write some inline ASM you can. However, I recommend against this since it makes it hard to port. Better, to simply add an external .S ASM file and then put your ASM functions in there, and call them from C. This way, you use straight C, and straight ASM, and both are easily portable to other compilers and assemblers by other vendors. Anyone that has used the GNU GCC inline assemblers knows they would rather poke dull forks in their eyes!

Figure 1.8 – The Parallax Propeller Chip block diagram.



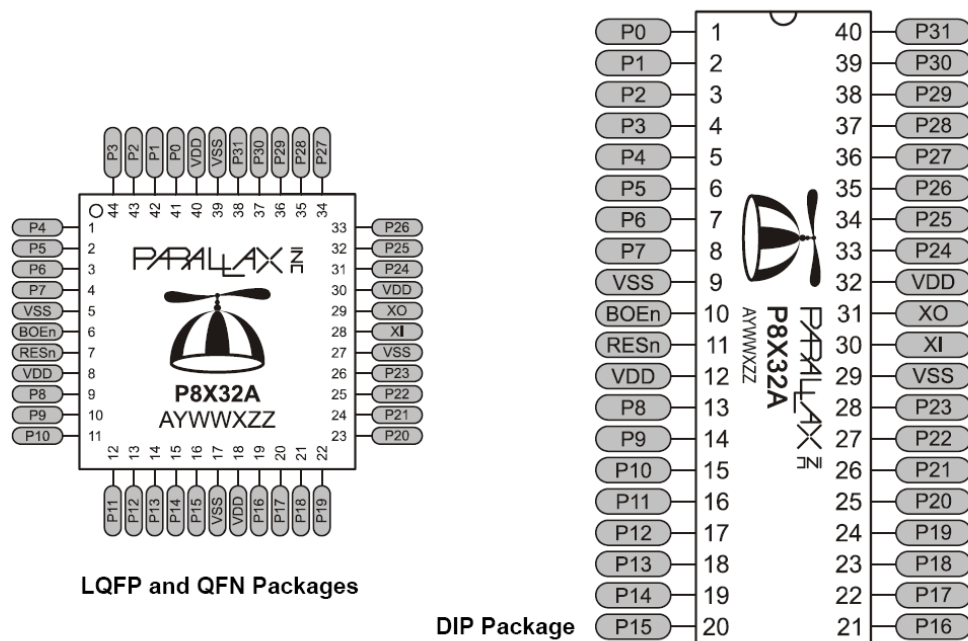
1.3 The Parallax Propeller Chip

The Parallax Propeller chip is a low cost multicore processor with a very simple design and programming model. Referring to Figure 1.8 take a look at the block diagram of the chip. The Propeller chip consists of (8) symmetrical processing units (cores) that are identical. Each core has a 32-bit CPU, 512 32-bit words of memory, a pair of counters and a crude video streaming unit. The cores run at 80MHz nominal and take 4 clocks to execute an instruction. Therefore, each core is capable of running at 20 MIPS. Additionally, each core has access to the 32 I/O pins of the chip as well as a shared 32K Byte RAM and 32K Byte ROM. Access to the RAM is arbitrated by a "hub" that gives each core access to the RAM in lock step, thus its impossible for one core to access RAM while another core is, thus, no potential for corruption of memory. For complete details on the processor, please review the datasheet located on the DVD-ROM here:

DVD-ROM: \ cham_avr \ docs \ datasheets \ PropellerDatasheet-v1.2.pdf

Referring to the pinout of the Propeller as shown in Figure 1.9, the philosophy of the Propeller is to keep things really simple; therefore, there are no peripherals (other than the video hardware) on the chip. The idea is that if you want a SPI port, UART, D/A, A/D, etc. you will write one with pure software and run it on one of the cores as a "virtual peripheral". Therefore, chip space and silicon aren't wasted on peripherals you will never use. Rather, the Propeller is 100% programmable and you simply load peripherals as software.

Figure 1.9 – The Propeller pinouts.



As you can see, the Propeller chips have nothing more than power, reset, clock, and 32 I/O pins. There are no dedicated peripherals whatsoever (actually, that's not completely true – each core has video hardware).

Table 1.4 – Propeller pin and signal descriptions.

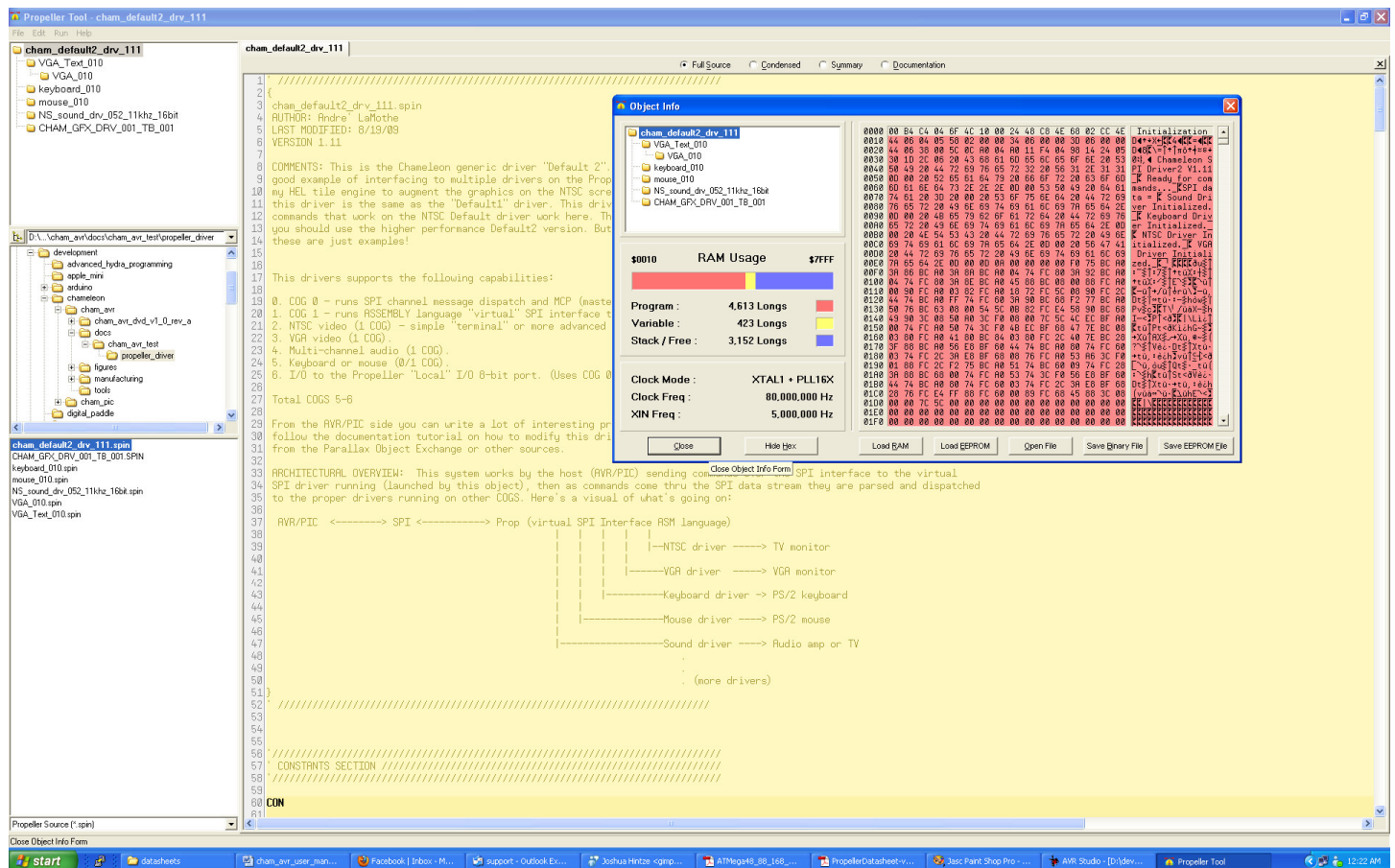
Pin Name	Direction	Description
P0 – P31	I/O	General purpose I/O Port A. Can source/sink 40 mA each at 3.3 VDC. CMOS level logic with threshold of $\approx \frac{1}{2}$ VDD or 1.6 VDC @ 3.3 VDC. The pins shown below have a special purpose upon power-up/reset but are general purpose I/O afterwards. P28 - I2C SCL connection to optional, external EEPROM. P29 - I2C SDA connection to optional, external EEPROM. P30 - Serial Tx to host. P31 - Serial Rx from host.
VDD	---	3.3 volt power (2.7 – 3.6 VDC)
VSS	---	Ground

BOEn	I	Brown Out Enable (active low). Must be connected to either VDD or VSS. If low, RESn becomes a weak output (delivering VDD through 5 kΩ) for monitoring purposes but can still be driven low to cause reset. If high, RESn is CMOS input with Schmitt Trigger.
RESn	I/O	Reset (active low). When low, resets the Propeller chip: all cogs disabled and I/O pins floating. Propeller restarts 50 ms after RESn transitions from low to high.
XI	I	Crystal Input. Can be connected to output of crystal/oscillator pack (with XO left disconnected), or to one leg of crystal (with XO connected to other leg of crystal or resonator) depending on CLK Register settings. No external resistors or capacitors are required.
XO	O	Crystal Output. Provides feedback for an external crystal, or may be left disconnected depending on CLK Register settings. No external resistors or capacitors are required.

The Parallax Propeller is really useful for doing many things at once without trying to use interrupts, multitasking, and round robin programming techniques to perform multiple tasks. You simply write your code and run it on a core. Then if that code crashes or is broken it does not affect the other cores at all! This is exactly what the Chameleon needs – a way to run multiple processes that each perform some kind of media task and then these processes wait for the master (the AVR) to send commands. Each core does not care what the neighboring cores are doing, only what it is doing.

Luckily, if this all sounds like this is way too complicated, you don't have to worry about it! You do not have to program the Propeller chip at all if you do not want to. The AVR communicates to it via a simple API, and from the AVR (your) perspective, the Propeller is just a media slave that generates NTSC/PAL/VGA graphics, plays sounds, reads mice and keyboards and more. However, to really explore the power of the Chameleon you will definitely want to play with the Propeller chip and modify its kernel driver(s) and so forth. This is actually quite easy to do with a single tool that interfaces to the Chameleon AVR over the USB port. Its called the Propeller IDE, and Figure 1.10 below shows a screen shot of it.

Figure 1.10 – The Propeller IDE in action.



The Propeller IDE allows you to program the Propeller chip in its native BASIC like language called "Spin" as well as assembly language. The editor has lots of features like syntax highlighting and code marking to help you keep things straight. Spin uses tabs for block nesting, so a little tricky if you're a BASIC or C/C++ programmer and not used to spaces/tabs indicating nesting. In any event, if you want to program the Propeller you will use Spin/ASM primarily. However, there are 3rd party C compilers, a FORTH is available, and other languages if you search the Parallax site.

1.31 Propeller Core (COG) Video Hardware

Lastly, let's talk about the video hardware for a moment. Probably the coolest thing about the Propeller chip is that each core has a little state machine that can stream video data out. These little devices are called Video Streaming Units or VSUs and every core has one. Thus, you can in theory drive 8 different video signals at the same time. The VSUs are not graphics engines, GPUs, even 2D stuff. They are nothing more than serializers that send bytes out to port. However, they send bytes out at a specific rate and the analog signals they can generate can be timed to synthesize NTSC or PAL signals including blanking, sync, chroma, and color burst signals. In essence, with each core you can use software to set up the VSU units, then the unit will send out data representing one portion or "chunk" of the video signal, then you feed it more. Therefore, you have to know how to generate NTSC, PAL, or VGA yourself, but the VSU helps and offloads a lot of the timing and critical coding you would need to do usually.

Alas, once again, you don't have to worry about this. Myself, and numerous other programmers have written countless video drivers for the Propeller chip. Everything from bitmap graphics, text modes, to sprite engines. So if you want graphics, you just find a driver, add it to your program and then make calls to it. More on this later when we learn how to modify the Chameleon drivers at the end of the manual.

1.4. System Startup and Reset Details

The Chameleon AVR has no operating system or external peripherals that need initialization, thus the AVR 328P more or less is the entire "system". When the AVR powers up out of reset, it will either load the boot loader (if there is one), or begin execution from the primary application memory of the FLASH. The initial configuration of the chip is defined by the "fuse" bit settings which are controlled by the programming of the chip. In our case, the fuse settings are mostly default, with the Arduino boot loader and set to clock from an external high speed xtal. We will discuss exact details when we get to the software and IDE configuration. Once the AVR 328P boots and starts executing code, whatever is in the application program area is executed, simple as that.

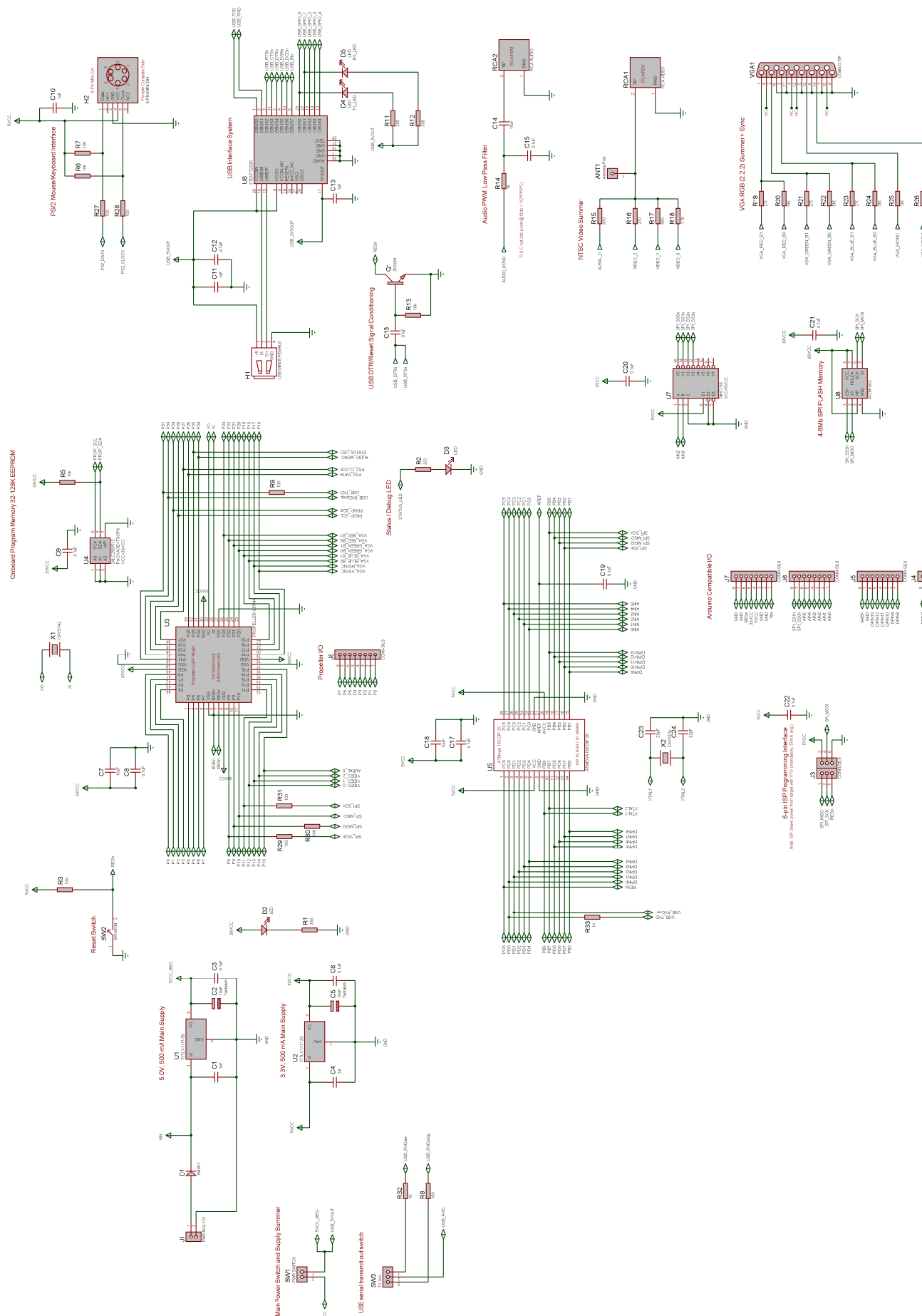
While this is all happening the Propeller chip boots as well and will start up doing whatever its programmed to do. Normally, this will be to load the default SPI driver system and "wait" for commands from the AVR. However, there is no reason the Propeller couldn't be programmed with a game, or other application and ignore the commands on the agreed to SPI interface pins. Therefore, the Chameleon can be used as a stand alone AVR or Propeller controller if you wish. However, the point is to use them together than leverage their strong points; the AVR's simple programming model, C/C++ support, huge library of software and customer base along with the multicore support of the Propeller and its ability to run multiple tasks at once as well as its special ability to generate video

With that in mind, now let's discuss every one of the hardware modules in the design.

Part I - Hardware Design Primer

In this section, we are going to cover every one of the Chameleon AVR's hardware sub-sections, so you can get an idea of what's going on and how the systems and signals are routed. Taken as a whole the Chameleon is rather complex, but as you will see, each individual sub-system is rather straightforward. Let's begin with a birds eye view of the entire system schematic on the next page in landscape mode (Figure 2.1a). Please take a good look at it, and see if you can locate the AVR, Propeller, FLASH memory, EEPROM, and power supplies to help with our discussions. Then right under it in Figure 2.1(b) is a PCB view so you can see where everything is as well as a clear view of the I/O headers in black and white with high contrast. The board is so small its, a bit hard to see things.

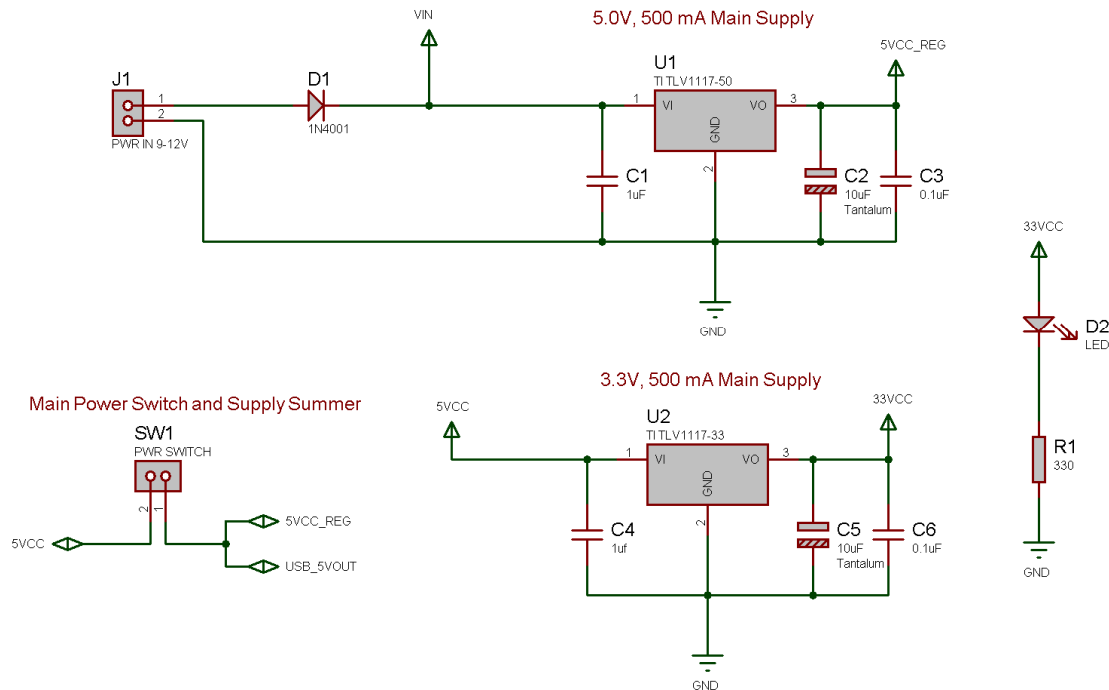
Figure 2.1(a) – The complete Chameleon AVR 8-bit schematic.





2.0 5.0V & 3.3V Power Supplies

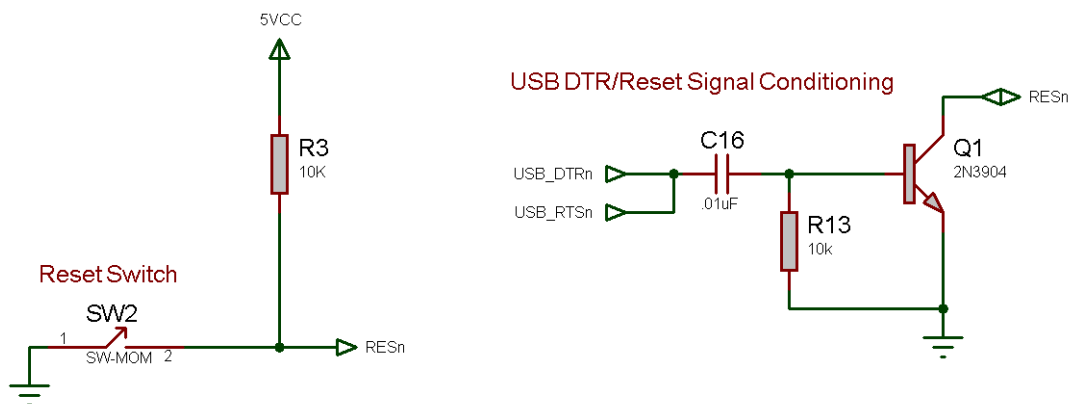
Figure 2.1 – The Chameleon AVR power supply design.



The Chameleon AVR system has a dual 3.3V / 5.0V power supply system that is fed by a single 9V DC unregulated input from a wall adapter or feeds from the USB port header. The supplies design is shown in Figure 3.1 for reference. The supplies are straight forward, the only interesting thing is the coupling of the USB power and the regulated power at node 1 of the power switch **SW1**. This allows the system to use power from either the regulated supply or the USB port. The Chameleon AVR is a mixed voltage system to support 5V AVR and 3.3V Propeller chip. Both supplies are independent and can each supply up to 500 mA for all your needs. Additionally, the power supplies are exported to the expansion header **J7** for external interfacing.

3.0 Reset Circuit

Figure 3.1 – The Chameleon AVR reset circuit.



The Chameleon AVR can be reset externally via a number of sources including the reset switch **SW2**, the ISP programming port as well as pressing as thru the **DTR** or **RTS** lines of the USB UART. The reset circuitry on the AVR

328P and Propeller don't require much conditioning, thus the manual reset circuit is nothing more than a switch and a pull up resistor as shown in Figure 3.1. Typically, you will reset the system yourself via the reset button (next to the AVR), but the Propeller tool as well as the Arduino tool both use the serial port lines **DTR** to control the reset of the respective chips during programming.

4.0 Atmel 6-Pin ISP Programming Port

The Chameleon AVR has two different programming ports:

- 6-pin ISP (In System Programming) port used to program the AVR chip directly
- USB UART used to program the AVR (with bootloader) as well as Propeller chip.

For more technical information on these ports and protocols please refer to these documents on the DVD:

DVD-ROM:\CHAM_AVR\DOCS\ISP_JTAG\ATAVRISPUserGuide.pdf

DVD-ROM:\CHAM_AVR\DOCS\ISP_JTAG\AVR_JTAG_ICE_USER_MANUAL_DOC2475.pdf

Let's discuss both programming ports and their relevance beginning with AVR's built in ISP port.

4.1 AVR ISP Programming Port

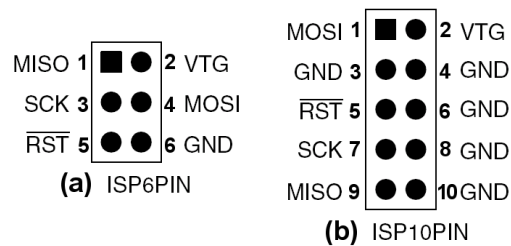
The ISP port is for programming the AVR chip only. Atmel and other 3rd party manufacturers sell programmers that are compatible with AVR Studio. The AVR product called the "**AVR ISP MKII**" is typically used from the Atmel line of products as shown in Figure 4.1.

Figure 4.1 – The Atmel AVR ISP MKII programmer.



The interface is typically 6-pin, but some hardware uses a 10-pin interface as well. The ISP interface was designed to be controlled by either a simple RS-232 or parallel interface with a modicum of hardware. The Chameleon AVR, uses the 6-pin interface variant as shown in Figure 4.2(a) (located right below the Propeller I/O port header).

Figure 4.2 – Diagram of the AVR ISP interfaces.



The ISP programming port is directly connected to the AVR chip's SPI (Serial Peripheral Interface) interface and consists of the following signals as shown in Table 4.1.

Table 4.1 – AVR ISP signals.

Signal	6-pin	10-pin	I/O	Description
MISO	1	9	Input	Data from target AVR to AVR ISP.
MOSI	4	1	Output	Data from AVR ISP to target.
SCK	3	7	Output	Serial clock from AVR ISP.
RESET	5	5	Output	Reset from AVR ISP.
VTG	2	2	POWER	Power from target to power AVR ISP (< 50mA).
GND	6	3,4,6,8,10	POWER	System ground.

The ISP protocol supports little more than reading and writing AVR (FLASH and EEPROM), resetting it, manipulating the "fuse" register settings and so forth. Debugging is **NOT** supported. However, you can build a AVR ISP programmer fairly easy and lots of schematics and designs are on internet that need only a RS-232 or parallel port. However, the Atmel product is USB based for speed.

TIP

The AVR ISP uses the primary SPI interface on the AVR, therefore, if you have anything connected to the SPI pins on the interface headers you might need to remove it during ISP programming, so you don't have conflicts.

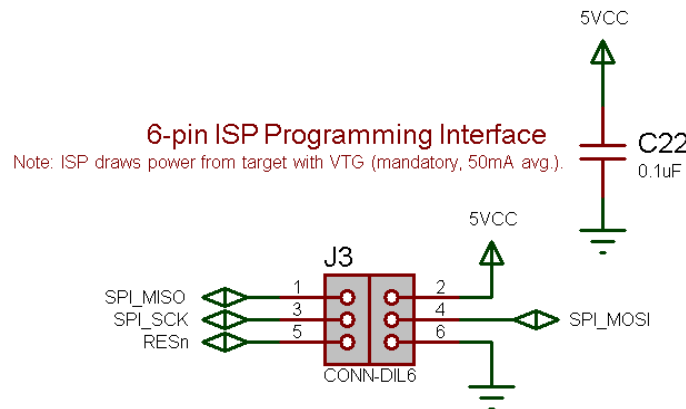
Figure 4.3 – The Chameleon ISP programming interface schematics.

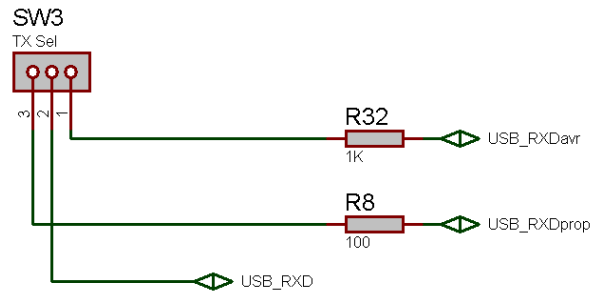
Figure 4.3 shows the actual circuitry for the ISP interface. As you can see its fairly straightforward. Nothing more than the port header routed to the proper pins and a bypass cap on the power signals. Also, the ISP programmer pulls its power from the Chameleon which is on the order of **50mA**.

5.0 Serial USB UART Programming Port

The Propeller chip as well as the AVR can also be programmed via the serial USB UART. The Propeller uses serial to communicate as default; however, the AVR uses serial only when there is a "bootloader" programmed into the AVR that "listens" to the serial port and there is a protocol in place as well as tool on the other end that follow the protocol agreement to re-program the FLASH. Therefore, the native method to re-program the Propeller is via the serial connection, but the AVR is typically re-programmed via the USP port, thus the serial re-programming is a trick/hack that is simply supported by a bootloader that supports it. With that in mind, the next section cover the USB UART design, but before we take a look at that there is one sub-section of the design we need to look at and that's the serial port "routing" switch shown in Figure 5.1.

Figure 5.1 – Serial port routing/selector switch.

USB serial transmit out switch



The USB UART has a standard TX/RX serial line pair. But, there are two targets we need these to go to; the Propeller chip and the AVR, thus, we just can't connect them in parallel since if both the Propeller and AVR try to transmit at the same time then there will obviously be a problem. Moreover, the impedance reflected by tying the TX/RX to both devices is not desired. Thus, a mechanical switch SW3 is used to switch the RX lines which at very least are needed to allow both devices to share the serial port. Therefore, when you want to talk to the Propeller chip via the USB serial port you must place the switch into the Propeller mode (**UP**), and similarly when you want to talk to the AVR you must put the switch into the AVR mode (**DOWN**).

TIP

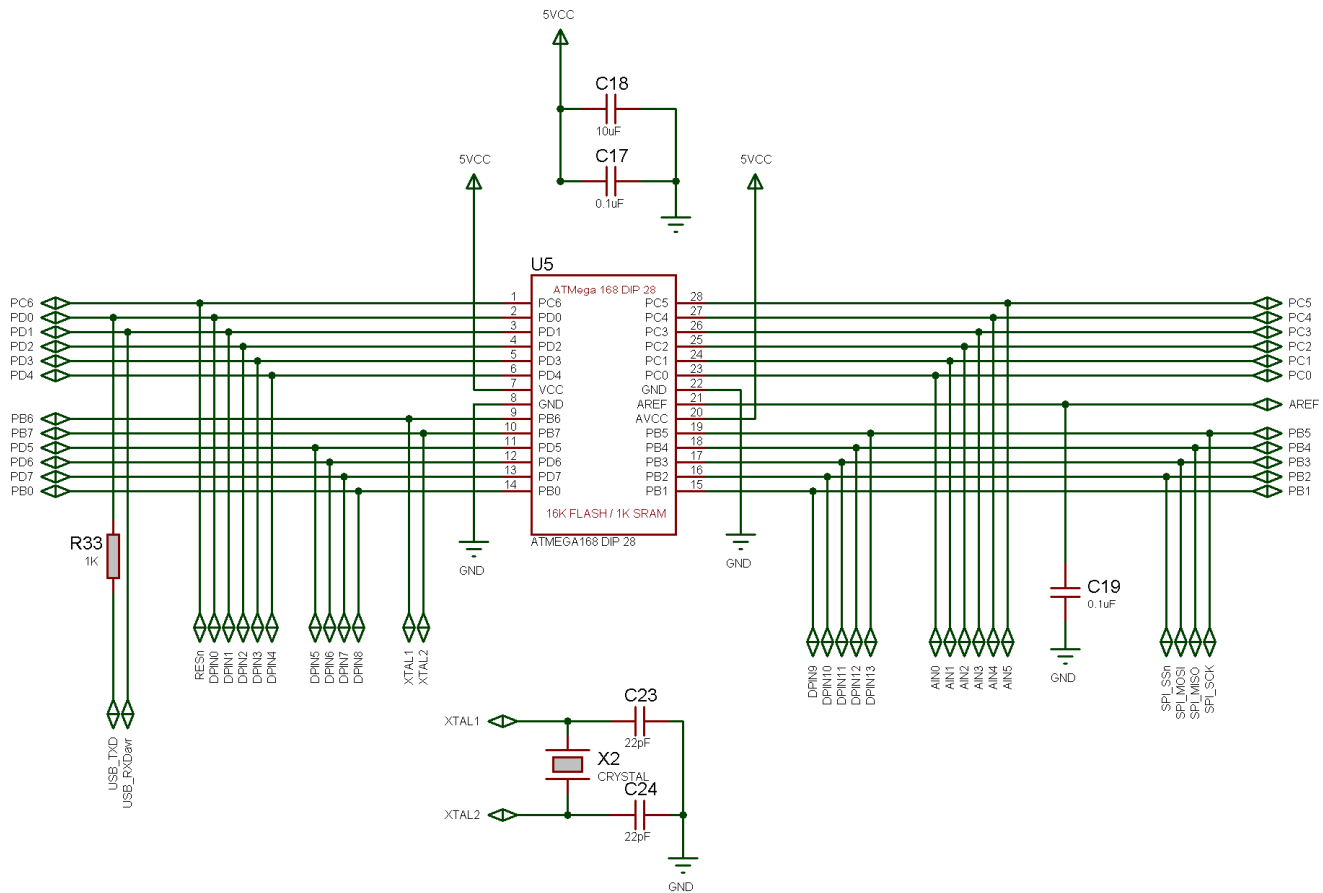
Typically, you will leave the switch in down (AVR mode) most of the time unless you are constantly re-programming the driver on the Propeller chip.

DVD-ROM: \ CHAM_AVR \ DOCS \ DATASHEETS \ DS_FT232R.pdf

33

7.0 Atmel AVR 328P Subsystem

Figure 7.1 – The AVR 328P Subsystem (not including SPI mux).



Referring to Figure 7.1, you can see the schematic refers to the ATMEGA 168 in the notation, this is because the 168 and 328 are 100% compatible, but the 328P has 2X the FLASH and RAM. Thus, when we started the design only the 168 was available. Moving on, the chip is very simple to interface, just power and an external XTAL (which is removable) at 16 MHz. Take note of the SPI interface lines on the right, these are very important since they are used to communicate with the Propeller chip as well as the on board 1MB FLASH.

8.0 Parallax Propeller Subsystem

Figure 8.1 – The Propeller Subsystem.

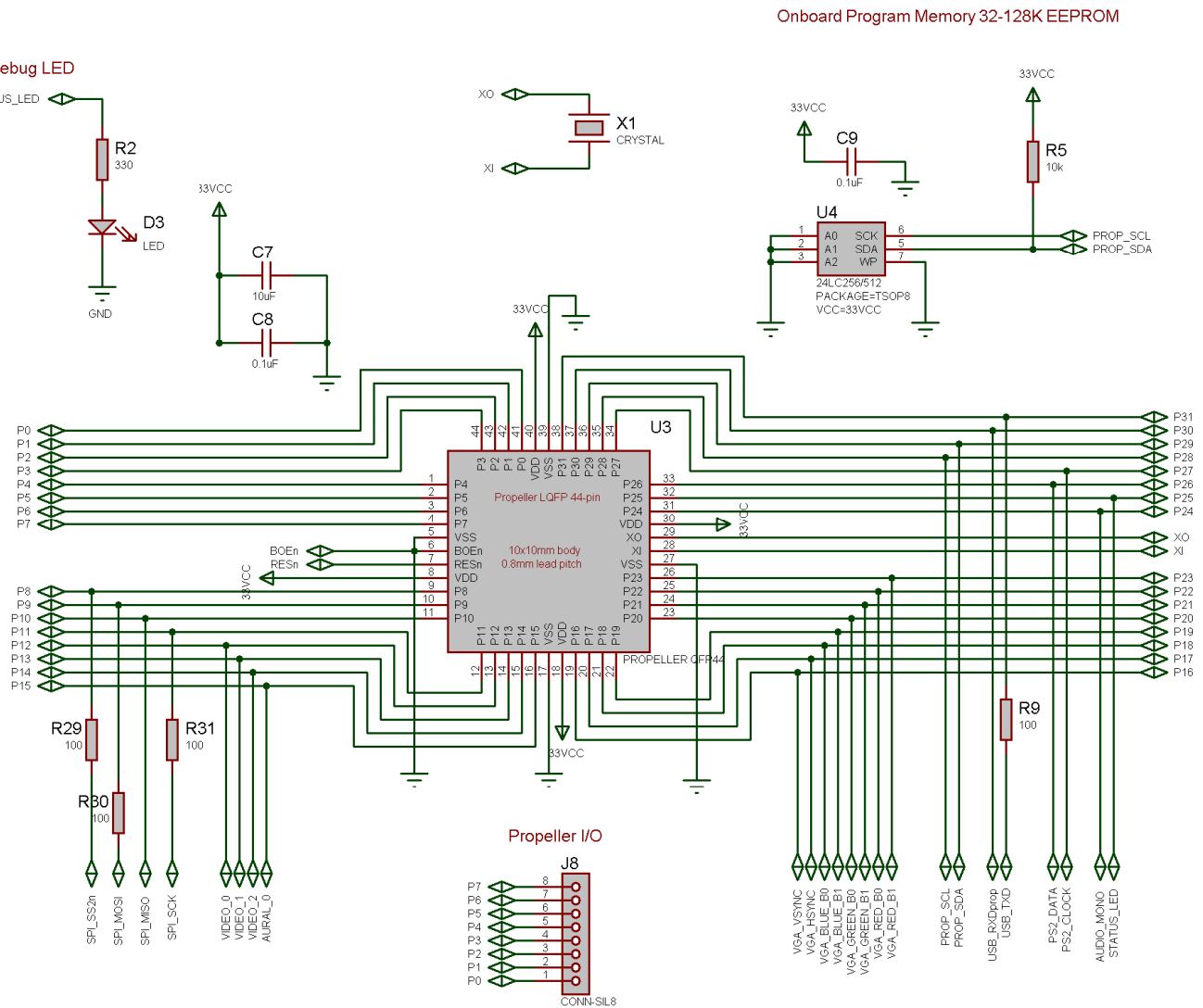


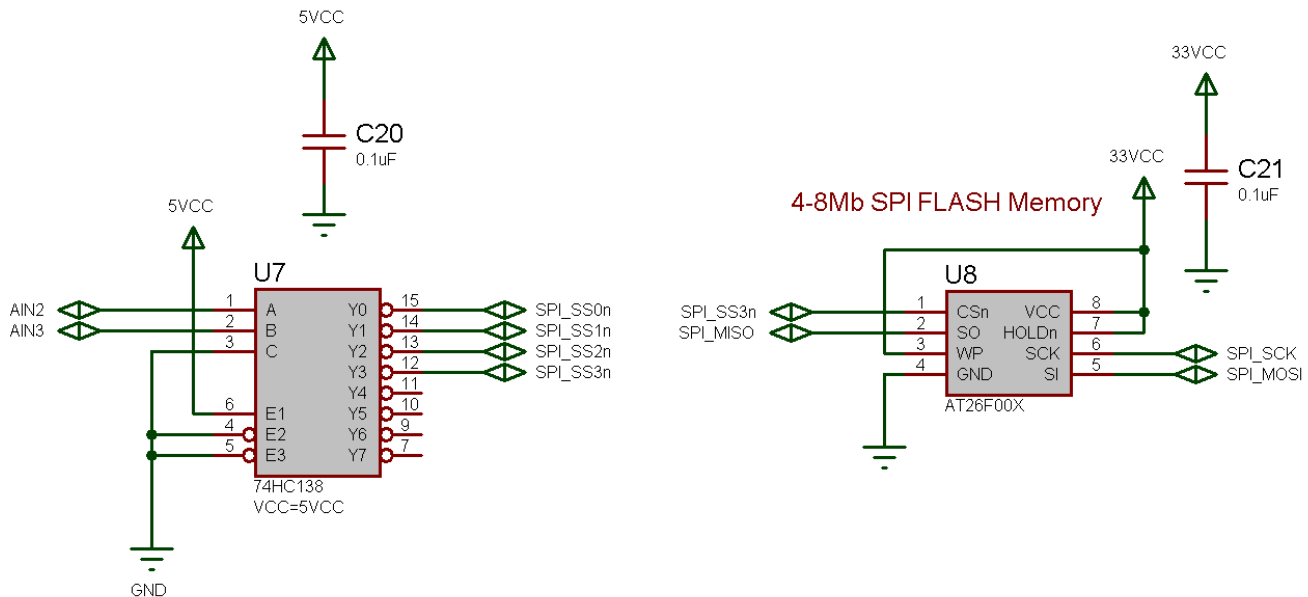
Figure 8.1 depicts the entire Propeller chip subsystem. It consists of the Propeller chip itself (QFP package), the 64K I²C serial EEPROM for program storage, a single LED, and the Propeller Port I/O header at **J8**. Additionally, the Propeller is clocked by a 5 MHz nominal XTAL at **X1**. The only thing interesting about the design is the free Propeller Local Port at **J8**, it connects to Propeller I/O pins P0...P7. Thus, you can connect other devices to this header, or potentially drive another NTSC/PAL/VGA video device or you can just use it at 8 more digital I/O ports. Lastly, the Propeller chip only needs a 32K EEPROM to load its runtime image into its local SRAM, however, we are using a 64K EEPROM which not only allows 2 images, but potentially a single image with additional program assets like graphics, sound, or code.

8.1 The Propeller Local 8-Bit I/O Port

The Propeller Local Port is an 8-bit port consisting of 8 free digital I/O pins from P0...P7 on the Propeller chip. You can do anything you want with the port. You can even power devices from the port by using some of the pins as power and virtual ground. The Propeller can source up to **40mA** per pin, so if you tie 2 lines together and set them HIGH, you can source up to **80mA**, more than enough for most applications.

9.0 The SPI Bus and Communications System

Figure 9.1 – The SPI multiplexer design.



The Chameleon uses SPI (serial peripheral interface) communications as the electrical communications interface to the Propeller chip as well as the on board 1MB FLASH memory. The idea was we wanted to interface the AVR with both the Propeller and the FLASH. The Propeller has no communications peripherals, so we could use anything we wanted; 2 lines, 4 lines, etc. and any kind of protocol we wished. However, since the FLASH memory was SPI, and there are 1000's of SPI devices out there, we decided to create a virtual SPI interface/driver for the Propeller and let it act as a SPI slave device. With this design decision then we can communication with the Propeller, FLASH and any other external SPI devices all with the AVR's built in hardware (very fast) and we have a nice clean communications bus for everything. The only challenge was supporting enough SPI devices, so the design uses a couple AVR I/O pins as chip selects for a multiplexer. Referring to Figure 2.10, the idea is you place a 2-bit code on a pair of I/O pins (**AIN2**, **AIN3**) which are fed into a 74138 decoder, this gives us 4 active low chip select signals (**SPI_SS0n**...**SPI_SS3n**) which we can export out to 4 SPI devices on the bus!

Referring to the schematic **SPI_SS3n** chip selects the FLASH. And if you look back a page or two at the Propeller schematic on the left hand side locate the SPI signals, you will see that the Propeller uses **SPI_SS2n** as its chip select respectively. The remaining chip selects (**SPI_SS0n**, **SPI_SS1n**) are exported out to the interface header J6 on pins (7,8) respectively. So, you can hook up 2 more SPI devices easily with the **MISO**, **MOSI**, **SCLK** lines and then connect on of the aforementioned SPI selects to enable it. All the other SPI devices will be disabled and thus no bus contingency issues.

10.0 VGA Graphics Hardware

In this chapter, we are going to take a look at the VGA hardware on the Chameleon AVR. The VGA hardware is very simple consisting of little more than (8) I/O lines, and (3) D/A converters based on resistors. The Propeller chip does all the work generating the VGA signal, but its nice to understand what's going on, so we are going to briefly take a look at the VGA specification, so you have a conversational understanding of it if you haven't programmed direct VGA before and why VGA is both good and bad from a video generation point of view. So, here's what's in store:

- Origins of VGA.
- Chameleon AVR VGA hardware design.
- VGA signal primer.

10.1 Origins of the VGA

Before we start into the design, let's get some history and terminology correct. First, the last time I saw a real VGA monitor was about 20 years ago, the new 21st century monitors that you are used to are super-sets of VGA, they are typically multisync, variable refresh, and can go up to 2400x2400 or more. The "VGA" specification is more of a base point than anything anyone actually uses anymore. IBM released the VGA or "**Video Graphics Array**" technology in 1987 roughly. It was touted to change the world (definitely an improvement from CGA and EGA), but it was doomed from the start with not enough resolution, shortly after, higher resolution monitors were available and the VGA specification was enhanced with the **Super VGA** standard, but this was only a momentary hold on its execution, what was needed was a more progressive standard that could change at any time, and thus the VESA "**Video Electronics Standard Association**" graphics cards and drivers were created and the rest is history. However, the original specs for the VGA are shown below.

- 256 KB Video RAM.
- 16-color and 256-color modes.
- 262,144-value color palette (six bits each for red, green, and blue).
- Selectable 25 MHz or 28 MHz master clock.
- Maximum of 720 horizontal pixels.
- Maximum of 480 lines.
- Refresh rates up to 70 Hz.
- Planar mode: up to 16 colors (4 bit planes).
- Packed-pixel mode: 256 colors, 1-byte per pixel (Mode 13h).
- Hardware smooth scrolling support.
- Some "raster operations" (Raster Ops) support to perform bitmap composition algorithms.
- Barrel shifter in hardware.
- Split screen support.
- Soft fonts.
- Supports both bitmapped and alphanumeric text modes.

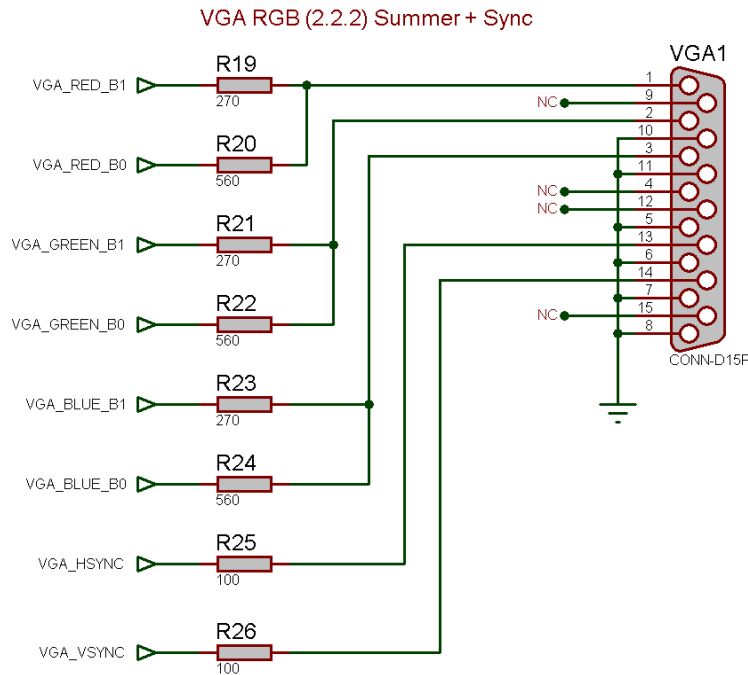
Standard graphics modes supported by VGA are:

- 640×480 in 16 colors.
- 640×350 in 16 colors.
- 320×200 in 16 colors.
- 320×200 in 256 colors (Mode 13h, "**chunky graphics mode**").

I spent many years writing graphics algorithms and drivers for VGA cards to push them to the outer limit; now, my cell phone has better graphics! However, the VGA spec is great for baseline computer displays and any standard modern computer monitor will display old school 640x480 VGA modes and refresh rates. However, the Propeller chip can generate VGA modes at much higher resolutions and there are a number of multicore hi-resolution drivers that you can employ that go all the way up to **1200x1024** or greater last time I checked on the **Parallax Object Exchange**. You can use any of these drivers with the Chameleon! You just need to modify the Propeller driver module, include the new VGA drivers, make connections to it via commands, and you are off and running.

10.2 VGA Hardware Interface

Figure 10.1 – The Chameleon AVR VGA interface and support hardware.

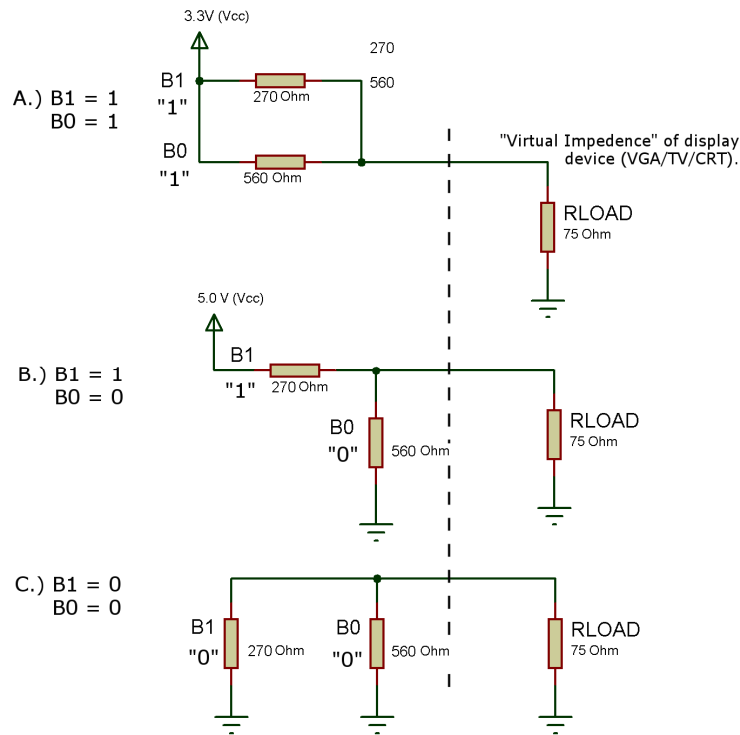


The VGA hardware for the Chameleon is very minimal since the Propeller chip is doing all the signal generation with direct control of the VGA signal via its built in VSU (video streaming unit). Of course, someone still has to write a driver for the VGA to work on the Propeller, but that's been done 100x. Referring to Figure 10.1, you see that there are 8 signals connected to the VGA port, Table 10.1 shows the relationship between I/O port bits and signals.

There are **2-bits** for each channel; **Red**, **Green**, and **Blue** as well as a single bit for **HSYNC** and **VSYNC** which are TTL level. These outputs are connected directly to the HD15 connector via a series of 2-bit D/A converters that sum each 2-bit channel pair (R1, R0), (G1, G0), and (B1, B0) and directly interface the sync bits. The VGA spec dictates that for each channel **0%** intensity is **0.0V** and **100%** intensity is **1.0V** (some references use 0.7V). Also, each channel input to the VGA monitor itself per input has a nominal impedance of **75 Ohms** (similar to the NTSC input impedance).

Table 10.1 – The Video Hardware.

Prop Port Bit	Chameleon Signal	Description
P23	VGA_RED_B1	Bit 0 of RED channel.
P22	VGA_RED_B0	Bit 1 of RED channel.
P21	VGA_GREEN_B1	Bit 0 of GREEN channel.
P20	VGA_GREEN_B0	Bit 1 of GREEN channel.
P19	VGA_BLUE_B1	Bit 0 of BLUE channel.
P18	VGA_BLUE_B0	Bit 1 of BLUE channel.
P17	VGA_HSYNC	VGA HSYNC TTL level.
P16	VGA_VSYNC	VGA VSYNC TTL level.

Figure 10.2 – Electrical model of VGA inputs and 2-bit D/A channels.

Therefore, from the D/A's perspective with both bits on, the circuit looks like that shown in Figure 10.2 (for any particular channel; R, G, or B). Noting, that the Propeller is a 3.3V device with both channel bits HIGH, that means that we have a 270 Ohm in parallel with a 560 Ohm and this combination in series with the intrinsic impedance of 75 Ohms of the VGA, thus, we can do a simple computation to compute the voltage at the 75 Ohm VGA input realizing we have a voltage divider configuration:

$$\text{VGA_IN_RED} = 3.3\text{V} * 75 \text{ Ohm} / ((270 \text{ Ohm} || 560 \text{ Ohm}) + 75 \text{ Ohm}) = 0.95\text{V}$$

Which is what we desire, performing similar math for all 4 combinations of inputs we get the results shown in Table 10.2 for codes.

Table 10.2 – VGA input voltages for all 2-bit inputs.

B1	B0	Code	VGA Voltage	Description
0	0	0	0.0V	Black
0	1	1	0.30V	Dark Gray
1	0	2	0.64V	Gray
1	1	3	0.95V	White

Considering there are 4 shades of each channel; R, G, and B that means that there are a total of $4*4*4 = 64$ colors available in VGA mode (without any tricks). Not bad! For reference, the VGA connector has the following pin out shown in Table 10.3. The VGA port on the Chameleon AVR is a female **HD15** (High Density) which is standard on the back of video card hardware. It will connect to any VGA monitor, but be careful when making connections, it's a special "low profile" connector and not that sturdy, so hold it when you plug the VGA cable into it.

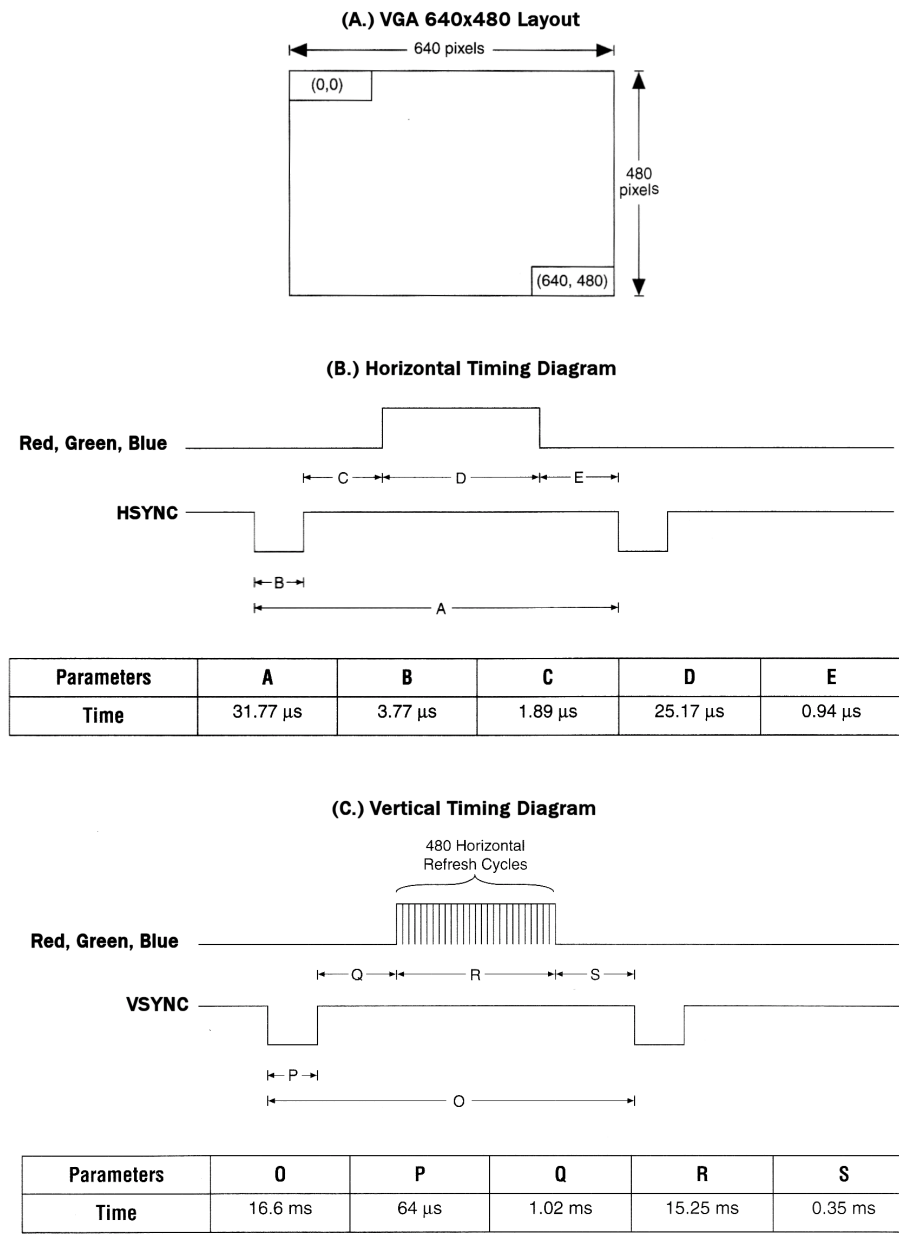
Table 10.3 - VGA female header pin out (HD15)

Pin	Function	Color Code (most manufactures)
1	RED Video	BLACK
2	GREEN Video	BLACK/WHITE
3	BLUE Video	BROWN
4	RESERVED	BROWN/WHITE
5	-GROUND	RED+SHIELD
6	-RED Ground	ORANGE
7	-GREEN Ground	YELLOW
8	-BLUE Ground	MINT GREEN
9	NC	NC
10	-GROUND	GREEN
11	ID0 (Ground)	BLUE
12	ID1 (NC)	PURPLE
13	VSYNC	GREY
14	HSYNC	WHITE
15	NC	SHRIMP

10.3 VGA Signal Primer

The VGA standard is much easier to understand than the NTSC standard is, in fact, using a visual explanation (Figure 10.3) is usually all people need to see, but still there are some gotcha's so we are going to discuss them as well in the brief primer. To start, the actual important signals you need to generate for VGA are (referring to Table 10.3 are) Red, Green, and Blue video as well as Hsync and Vsync. The R,G,B signals are analog voltages representing the instantaneous intensity of any particular channel and range from 0-1.0V, the Hsync and Vsync are simply TTL signals, usually active LOW (but these can be inverted on most monitors) where a logic LOW is sync, and a logic HIGH is no sync. Now, let's look more closely at the signal itself.

Figure 10.3 – The VGA timing specifications.



To begin with the VGA standard as implemented is 640x480 pixels across as shown in Figure 2.13(a). The image is rendered left to right, top to bottom, similarly to the NTSC/PAL signals, and thus a similar syncing scheme is used that is composed of both a horizontal sync pulse each line and a vertical sync pulse each frame. However, there are no color burst signals, serrations, pre-equalizations pulses, etc. the interface is nearly digital as noted.

The dot clock used in a basic VGA generation system is 25.175 MHz, all timing can be derived from this base frequency. Typically, designers like to run the system dot clock at this frequency and compute all events as a number of clocks, for example, getting ahead of myself, take a look at Figure 10.3(b), the Hsync pulse which is labeled "B" in the table (and a

negative polarity), its **3.77 μ s**, therefore at a dot clock of 25.175 MHz, or inverting this to get the period time we get **39.72194638 ns**. This is how long a pixel takes, anyway, dividing this into our Hsync time we get:

Number of dot clocks for Hsync pulse = 3.77 μ s / 39.72194638 ns = 94.90 clocks.

Call it 95 dot clocks, thus you can simply use a counter and count 95 clocks, drive Hsync LOW and that it. The entire VGA signal can be generated like this. Of course, the tough part is when you get to video, here you only have roughly 39 nanoseconds to do whatever you are going to do, this amounts to more or less doing nothing, but accessing a video buffer as fast as you can and getting the next data word ready to build the pixel.

NOTE

You might be wondering where the 25.175 MHz clock is generated on the Chameleon? Well, the Propeller chip is generating the signal and it has internal counters and PLLs that generate the 25.175 MHz signal for us.

10.3.1 VGA Horizontal Timing

Referring to Figure 2.13(b), each of the 480 lines of video are composed of the following standard named regions:

- A (31.77 μ s) Scanline time.
- B (3.77 μ s) HSync pulse.
- C (1.89 μ s) Back porch.
- D (25.17 μ s) Active video time.
- E (0.94 μ s) Front porch.

As you can see even the names are similar to NTSC. However, unlike NTSC, VGA is very unforgiving you must follow the spec more closely otherwise the monitor's will typically ignore the signal. The next interesting thing is that all the signals are riding on different lines. For example, the hsync and vsync both have their own signal lines, and the R, G, and B signals do as well, so its much easier to generate all the signals in a large state machine and then route them to the output ports as a single byte wide stream of data.

TIP

Even though, VGA is 640x480 that doesn't mean that you have to put out 640x480 pixels and stream video at that rate. The Propeller chip for example only has 32K bytes of memory, thus a full frame buffer with one byte per pixel would require 307,200 bytes! Therefore, the idea is to use tile graphics modes, or reduced resolution modes, but still conforming to the VGA timing constraints.

Therefore, to generate a line of video (with a negative sync polarity), you start by turning off all the R, G, B channels with a 0.0V, and simply hold the hsync line LOW for **3.77 μ s** (B), then you wait **1.89 μ s** (C) and the VGA is ready to take R, G, B data, now you clock out 640 pixels at 25.175 MHz for a total time of **25.17 μ s** (D) for the active video portion. You then turn the video lines R, G, B off, and wait **0.94 μ s** (E) and then start the process again. And that's all there is too it. Perform this process 480 times then its time for a vertical sync and retrace, let's look at that.

10.3.2 VGA Vertical Timing

The vertical sync and retrace is much easier than the horizontal timing since there is no video to interleave in the signal. Referring to Figure 8.3 (c) the various named timing regions of the vertical timing are:

- O (16.68 ms) Total frame time
- P (64 μ s) Vsync pulse.
- Q (1.02 ms) Back porch
- R (15.25 ms) Active video time.
- S (0.35 ms) Front porch.

The meaning of each is similar to that in the horizontal sync period, except the "event" they are focused around are the 480 lines of active video, so the 480 lines are encapsulated in (R). The most important thing to realize is that these times are measured in milliseconds for the most part except for the Vsync pulse. So once again, the timing of an entire frame starts off with the Vsync pulse (P) for **64 μ s**, after which comes the back porch (Q) for **1.02 ms**, followed by the active video (R) for **15.25 ms**. During the active video you don't need to worry about the Vsync line since you would be

generating 480 lines of video, when complete, back to the Vsync timing region (S) the front porch for 0.35 ms and then the frame is complete.

The thing to remember is that unlike composite video, a VGA signal needs to be driven by multiple signals for each of the constituent controls; R, G, B, Hsync, and VSync which in my opinion is much easier than modulating and mixing them all together as in NTSC which is a blood bath of crosstalk and noise! Now, that we have the horizontal and vertical timing for VGA covered, let's review the actual video data portion of the signal during the active video and see what that's all about.

Table 10.4 – The RGB signals and their relationship to VGA and the Propeller I/O pins.

Prop Port Bit	Chameleon Signal	Description
P23	VGA_RED_B1	Bit 1 of RED channel.
P22	VGA_RED_B0	Bit 0 of RED channel.
P21	VGA_GREEN_B1	Bit 1 of GREEN channel.
P20	VGA_GREEN_B0	Bit 0 of GREEN channel.
P19	VGA_BLUE_B1	Bit 1 of BLUE channel.
P18	VGA_BLUE_B0	Bit 0 of BLUE channel.
P17	VGA_HSYNC	VGA HSYNC TTL level.
P16	VGA_VSYNC	VGA VSYNC TTL level.

10.3.3 Generating the Active VGA Video

Generating the actual RGB video pixels is trivial on any system, there is no look up, no math, just send out bytes or words that represent the RGB values and that's it. Of course the Propeller will do this for us via the VSU and a driver, but there is no reason you can't drive the I/O pins manually with pure software as well. Either way, let's take a look at the mechanics of the signal. On the Chameleon AVR for example, there are 2-bits per channel, so the encoding of the VGA data byte is as simple as generating bytes in the format shown in Figure 10.4.

Figure 10.4 – VGA data byte encoding.

I/O	P23	P22	P21	P20	P19	P18	P17	P16
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	R1	R0	G1	G0	B1	B0	VSYNC	HSYNC
	Red Signal		Green Signal		Blue Signal		Sync Signals	

Let's say that we have a BYTE buffer called **vga_byte[]** and a function **Write_VGA(value, time_us)** that we use to send data to the VGA port, given that, we can write all kinds of functions that send out different signals as long as we stream the bytes to the port at the 25.175MHz rate everything will work out fine. For example, say that we are working with a positive sync system, that is a VGA monitor that wants a TTL HIGH for sync, then to generate a Hsync pulse we could use some pseudo-code like this:

```
vga_byte = 0b00000010;
write_VGA(vga_byte, 3.77);
```

Ok, now consider we want to draw 640 pixels from an array **video_buffer[]** that is storing the pixel data in byte format already in the proper format for our hardware, then all we would do is this:

```
UCHAR video_buffer[640]; // pre-initialized video buffer
for (pixel = 0; pixel < 640; pixel++)
    write_VGA(video_buffer[pixel], 0.039721946);
```

Of course, you would need some might fast hardware to delay at a resolution of **39 ns**, but you get the idea. This is a "model" of the algorithm for a line of video, this coupled with the model for the horizontal timing, vertical timing, and put it all together as a state machine and you have a VGA generator! The VGA drivers on the Propeller are more complex than this of course, but that's the idea.

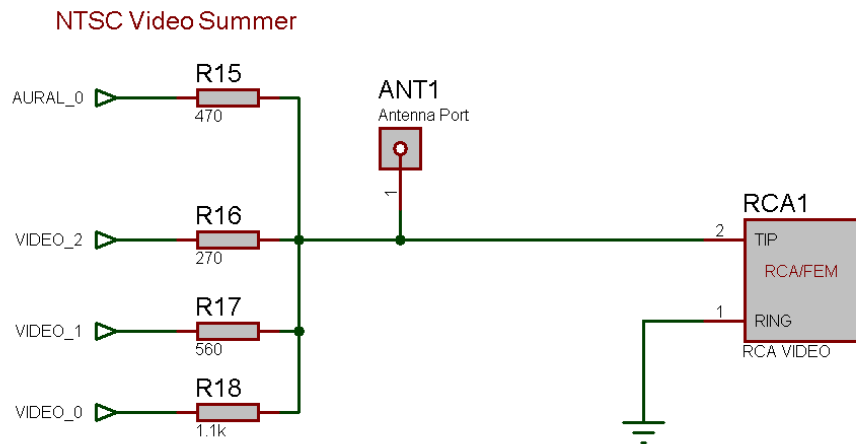
11.0 NTSC/PAL Composite Video Hardware

To begin with there are many online and written sources for NTSC, the primary online resource is located here:

<http://www.ntsc-tv.com/>

At the end of the discussion on NTSC there is a list of other resources as well.

Figure 11.1 – The Chameleon AVR composite video hardware.



11.1 Video Hardware Interface

Referring to Figure 11.1, the composite video generation hardware consists nothing more than a crude 3-bit D/A converter (summing circuit) along with a single signal for audio summed with the node as well. The real magic of the video generation is up to the Propeller chip, it not only generates the luma (brightness), but the chroma (color) in the proper format and timing for whatever the desired protocol is (NTSC or PAL).

The Propeller chip video hardware is designed to be interfaced to a simple 3-bit DAC with the resistance values chosen and shown above in the figure. Table 11.1 shows the signal relationship to the Propeller I/O signals.

Table 11.1 – The Video Hardware.

Prop Port Bit	Chameleon Signal	Description
P12	VIDEO_0	Bit 0 of the overall composite video signal (contains both LUMA and CHROMA).
P13	VIDEO_1	Bit 1 of the overall composite video signal (contains both LUMA and CHROMA).
P14	VIDEO_2	Bit 2 of the overall composite video signal (contains both LUMA and CHROMA).
P15	AURAL_0	Single "aural" audio bit used when transmitting broadcast video.

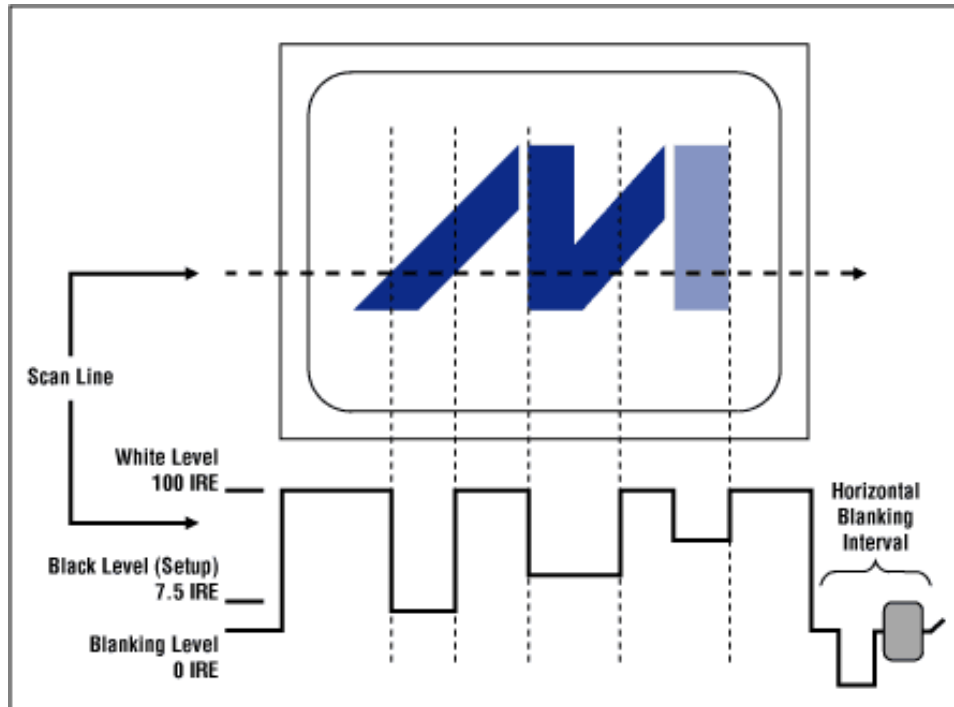
There are 3-bits of D/A conversion for the overall composite video signal **VIDEO_0-3** thus giving a total of 8 different overall LUMA values; however, a number of them at the low scale of the range must be used to jump from "sync level" 0.0V to "black level" 0.25 - 0.3V. For example if the entire D/A range is 1.0V then $0.3 / 1.0 = 30\%$, 30% of 7 is 2, thus the 3-bit value of 0 is sync while the 4-bit value of 2 is black, leaving us with only the values 3-7 or 5 different intensity levels. Additionally, since the chroma signal rides on top of the luma signal there is yet another constraint since the chroma need to have a peak to peak of at least 2 units, this in fact, we can only drive the video signal to a value of 6, so the Propeller's chroma hardware can twiddle the luma ± 1 to obtain the chroma. However, in fact most drivers simply use 0 as sync, 1 as black, then values 1...6 as shades of gray. This gives us a total luma range of 6 values and still allows color to work which the Propeller is designed to be able output 16 phase shifts of the color burst. This gives a total color/luma range of $16 \times 6 = 96$ colors without resorting to tricks. If you are really interested in how to write graphics drivers you should peruse the source code of the various graphics drivers written by various authors for the Propeller. Additionally, the only reference book that explains in detail how the graphics system works is "**Game Programming for the Propeller Powered HYDRA**" which you can find on Amazon.com, Parallax.com, and XGameStation.com. There is also some minimal

coverage of the Propeller's graphics hardware (transcribed from the HYDRA book) in the Propeller Manual itself located on the DVD here:

DVD-ROM: \ CHAM_AVR \ DOCS \ PROPELLER \ Propeller_Manual_WebPM-v1.1.pdf

Now let's discuss a little bit about NTSC video signal generation.

Figure 11.2 - Horizontal scan versus display brightness



NOTE

"IRE" is an arbitrary unit that relates to the voltage of the video signal, 100 IRE = 1.0V these days; however, previously 140 IRE 1.0V. There is an EXACT spec for broadcast, but many video generation hardware units assume a total of 1.0V for the total SYNC -> WHITE video amplitude peak-peak voltage.

11.2 Introduction to NTSC Video

A video image is "**drawn**" on a television or computer display screen by sweeping an electrical signal (that controls a beam of electrons that strike the screen's phosphor surface) horizontally across the display one line at a time. The amplitude of this signal versus time represents the instantaneous brightness at that physical point on the display. Figure 11.2 illustrates the signal amplitude relationship to the brightness on the display.

At the end of each line, there is a portion of the waveform (horizontal blanking interval) that instructs the scanning circuit in the display to retrace to the left edge of the display and then start scanning the next line. Starting at the top, all of the lines on the display are scanned in this way. One complete set of lines makes a frame. Once the first complete frame is scanned, there is another portion of the waveform (vertical blanking interval, not shown) that tells the scanning circuit to retrace to the top of the display and start scanning the next frame, or picture. This sequence is repeated at a fast enough rate so that the displayed images are perceived to have continuous motion.

11.2.1 Interlaced versus Progressive Scans

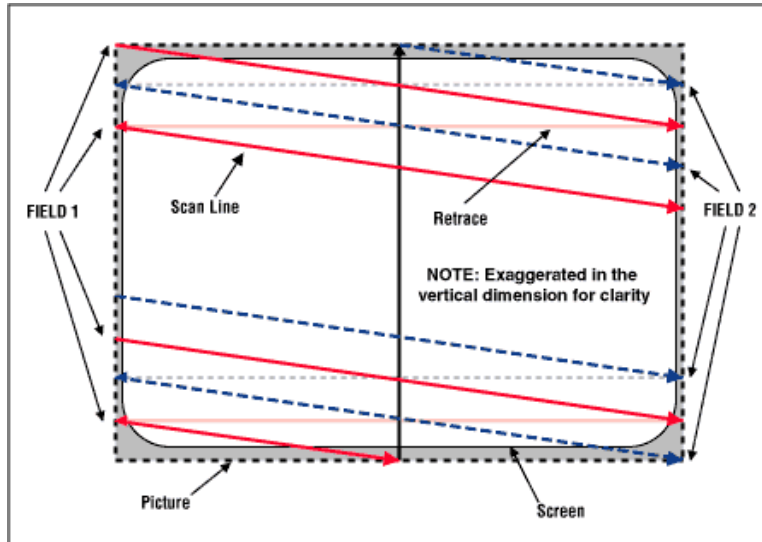
These are two different types of scanning systems. They differ in the technique used to "render" the picture on the screen. Television signals and compatible displays are typically **interlaced**, and computer signals and compatible displays are typically **progressive** scan (non-interlaced). These two formats are incompatible with each other; one would need to be converted to the other before any common processing could be done. Interlaced scanning is where each picture, referred to as a frame, is divided into two separate sub-pictures, referred to as fields.

Two fields make up a single frame. An interlaced picture is painted on the screen in two passes, by first scanning the horizontal lines of the first field and then retracing to the top of the screen and then scanning the horizontal lines for the second field in-between the first set. Field 1 consists of lines 1 through 262 1/2, and field 2 consists of lines 262 1/2 through 525. The interlaced principle is illustrated in Figure 11.3. Only a few lines at the top and the bottom of each field are shown.

NOTE

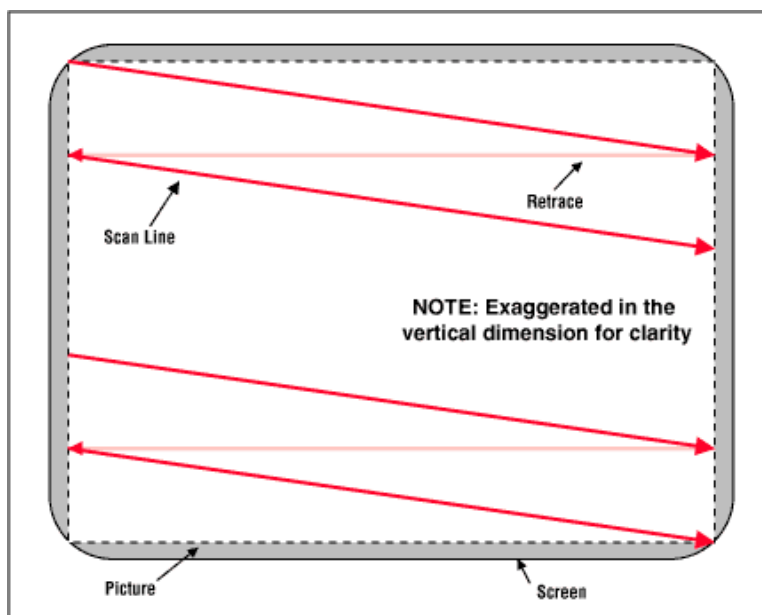
Most computer displays that output to TVs are progressive scans, that is, they draw a single frame 60 times a second without interlacing. However, some computers and game systems do put out interlaced video. The old Amiga computer was one such example that had an interlaced video mode option to double the number of scanlines.

Figure 11.3 - Interlaced scanning system.



A progressive, or non-interlaced, picture is painted on the screen by scanning all of the horizontal lines of the picture in one pass from the top to the bottom. This is illustrated in Figure 11.4.

Figure 11.4 - Progressive (non-interlaced) scanning system.



11.2.2 Video Formats and Interfaces

There are many different kinds of video signals, which can be divided into either two classes; those for television and those for computer displays. The format of television signals varies from country to country. In the United States and Japan, the NTSC format is used. NTSC stands for National Television Systems Committee, which is the name of the organization that developed the standard. In Europe, the PAL format is common. PAL (phase alternating line), developed after NTSC, is an improvement over NTSC. SECAM is used in France and stands for sequential couleur avec memoire (color with memory). It should be noted that there is a total of about 15 different sub-formats contained within these three general formats. Each of the formats is generally not compatible with the others. Although they all utilize the same basic scanning system and represent color with a type of phase modulation, they differ in specific scanning frequencies, number of scan lines, and color modulation techniques, among others. The various computer formats (such as VGA, SVGA, XGA) also differ substantially, with the primary difference in the scan frequencies and resolutions. These differences do not cause as much concern, because most computer equipment is now designed to handle variable scan rates (multisync monitors are now the de-facto standard). This compatibility is a major advantage for computer formats in that media, and content can be interchanged on a global basis. Table 11.2 lists some of the more popular formats along with comparisons of their specifications.

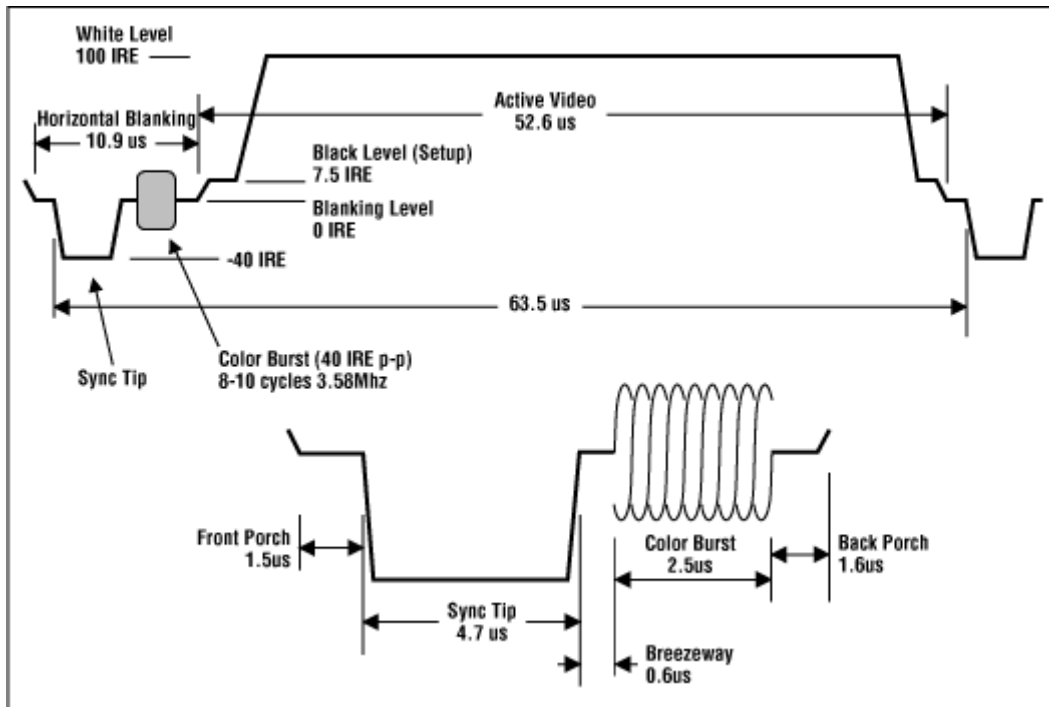
Table 11.2 - Typical Frequencies for Common TV and Computer Video Formats.

Video Format	NTSC	PAL	HDTV/SDTV	VGA	XGA
Description	Television Format for North America and Japan	Television Format for Most of Europe and South America	High Definition/Standard Definition Digital Television Format	Video Graphics Array (PC)	Extended Graphics Array (PC)
Vertical Resolution Format (visible lines per frame)	Approx 480 (525 total lines)	Approx 575 (625 total lines)	1080 or 720 or 480; 18 different formats	480	768
Horizontal Resolution Format (visible pixels per line)	Determined by bandwidth, ranges from 320 to 650	Determined by bandwidth, ranges from 320 to 720	1920 or 704 or 640; 18 different formats	640	1024
Horizontal Rate (kHz)	15.734	15.625	33.75-45	31.5	60
Vertical Frame Rate (Hz)	29.97	25	30-60	60-80	60-80
Highest Frequency (MHz)	4.2	5.5	25	15.3	40.7

Of course, not listed in the table are HDMI video modes. These are a whole other ballgame, but are very similar to XGA modes and higher.

11.2.3. Composite Color Video Blanking Sync Interface

Composite signals are the most commonly used analog video interface for NTSC/PAL. Composite video is also referred to as CVBS, which stands for color, video, blanking, and sync, or composite video baseband signal. It combines the brightness information (luma), the color information (chroma), and the synchronizing signals on just one cable. The connector is typically an RCA jack. This is the same connector as that used for standard line level audio connections. S-video of course is slightly cleaner since the analog mixing of chroma and luma does **NOT** take place. A typical waveform of an all-white NTSC composite video signal is shown in Figure 11.5 (a).

Figure 11.5 (a) - NTSC composite video waveform.

This figure depicts the portion of the signal that represents one horizontal scan line. Each line is made up of the active video portion and the horizontal blanking portion. The active video portion contains the picture brightness (luma) and color (chroma) information. The brightness information is the instantaneous amplitude at any point in time. The unit of measure for the amplitude is in terms of an IRE unit. IRE is an arbitrary unit where 140 IRE = 1Vp-p (or sometimes 1.4Vp-p). From the figure, you can see that the voltage during the active video portion would yield a bright-white picture for this horizontal scan line, whereas the horizontal blanking portion would be displayed as black and therefore not seen on the screen. Please refer back to Figure 11.3.4 for a pictorial explanation. Some video systems (NTSC only) use something called "setup," which places reference black a point equal to 7.5 IRE or about 54mV above the blanking level.

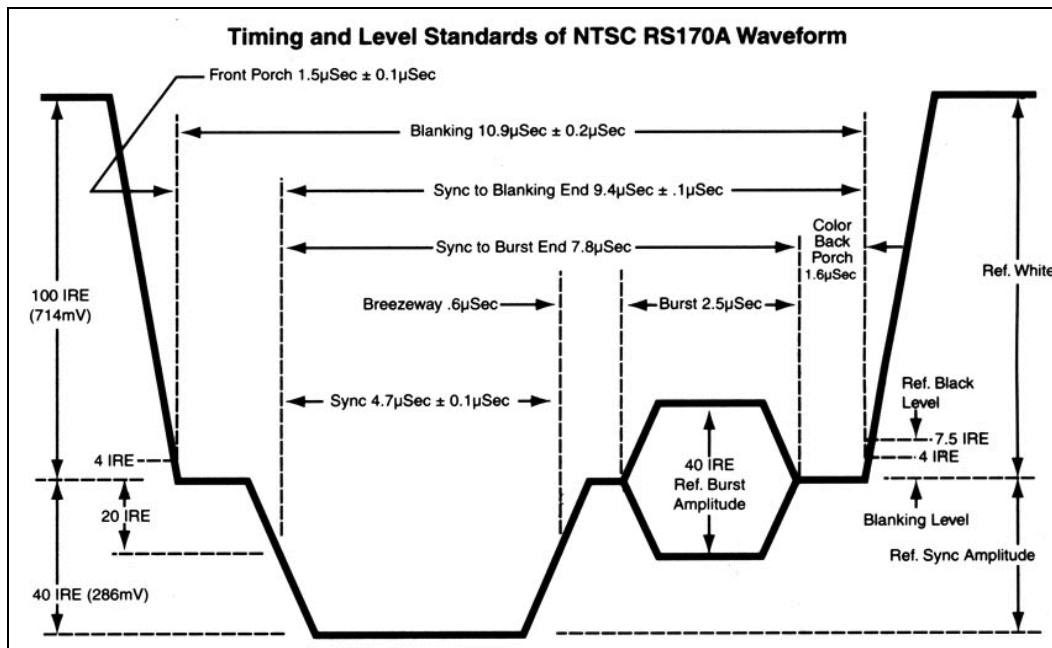
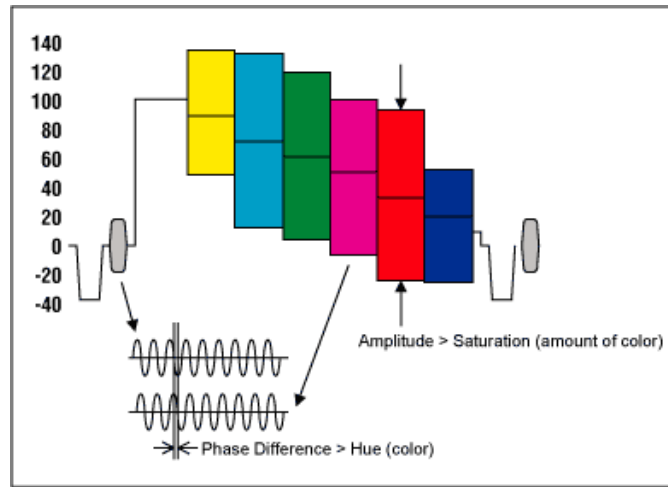
Figure 11.6 (b) - NTSC composite video waveform.

Figure 11.6 (b) depicts yet another timing drawing the NTSC video signal, this time, more formally labeled at the RS-170A color standard created in the early 1950's as the successor to the B/W only RS-170 standard created 1941.

11.2.4 Color Encoding

Color information is superimposed on top of the luma (brightness) signal and is a sine wave with the colors identified by a specific phase difference between it and the color-burst reference phase at the beginning of each scanline. This can be seen in Figure 11.6(c), which shows a horizontal scan line of color bars.

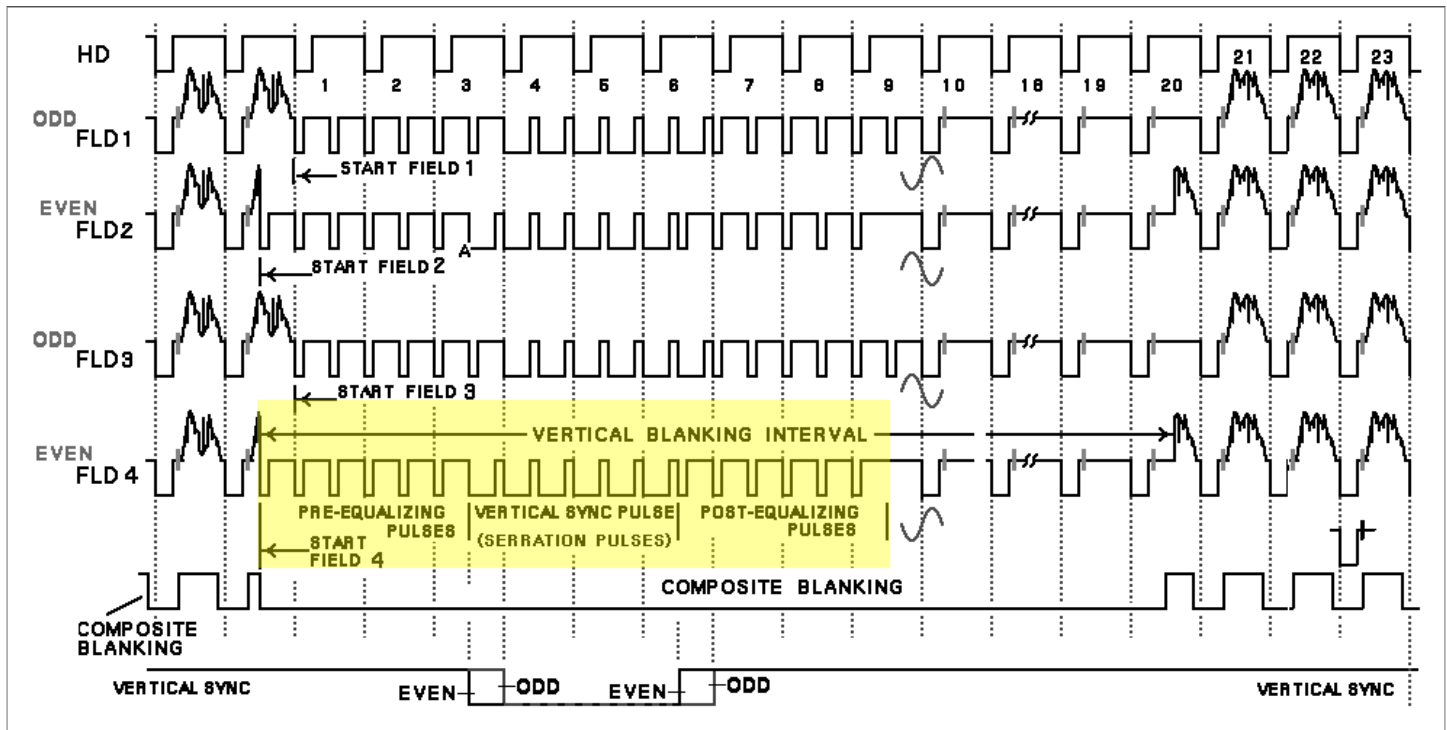
Figure 11.6 (c) - Composite Video Waveform of the Color Bars.



The amplitude of the modulation is proportional to the amount of color (or saturation), and the phase information denotes the tint (or hue) of the color. The horizontal blanking portion contains the horizontal synchronizing pulse (sync pulse) as well as the color reference (color burst) located just after the rising edge of the sync pulse (called the "back porch"). It is important to note here that the horizontal blanking portion of the signal is positioned in time such that it is not visible on the display screen.

11.2.5 Putting it All Together

Generating a NTSC (or PAL) signal means controlling the signal into the TV such that all the timing specifications are met. There are a number of aspects that must be taken into consideration to achieve this successfully. However, one important detail is that the video signal does not have to be perfect, that is, the timings can be slightly off per line, per frame, per color burst, etc. nevertheless, the amount of "**slack**" must be consistent. In general here are the steps to generating an NTSC signal.

Figure 11.7 – NTSC “Super Frame” video signal reference for NTSC frame construction.

11.2.5.1 Frame Construction

Referring to Figure 11.7. This diagram represents the NTSC color **"Super Frame"**. The complete NTSC spec actually has 4 fields per super frame (if you want to follow the spec and interlace 100%). We aren't going to follow the spec 100%, but we are going to follow some of the more important parts of the spec, one such area is in the **"vertical sync"** and **"equalization pulses"** area, please concentrate your attention on this area of Figure 2.20, for the description below.

A Frame is composed of 262.5 lines (non-interlaced), 262 will suffice. Of those 262 lines, many of them are invisible due to overscan, and some must be used for the vertical retrace pulse. Therefore, a first attempt to generate a video signal frame is the following algorithm:

```
// Step 1: Draw the top over scan
For line = 1 to 8
  Begin
  Generate the next black scanline
  End

// Step 2: Draw the active region of the scan, 240 lines
For line = 1 to 240
  Begin
  Generate the next scanline
  End

// Step 3: Draw the bottom over scan
For line = 1 to 10
  Begin
  Generate the next black scanline
  End

// Step 4: Generate a vertical sync pulse, basically apply 0v to the signal for
// the time duration of scanlines, note, no HSYNC pulse is sent, no serrations
Video = 0.0v
Delay(4-6 Scanlines)
```

If you add up the lines, you will get a total of 262 lines including active, inactive, and retrace or vertical sync. Now, due to overscan, you probably won't be able to see more than 200-220 of the active scan video lines, they will typically be beyond view. Thus, most games systems don't waste draw them, and simply increase the overscan regions. So a good rule of thumb is that active video should be from 192-224 lines with equal amount of top and bottom overscan.

Now, the above algorithm to draw a frame will work, and is very simple, but it doesn't comply with the RS-170A spec completely. It's missing a very important signal component called the **vertical serrations** in the vertical sync pulse. If you look at the rough draft algorithm above, we have a **vertical sync** period, where we simply generate sync signal for 4-6 scanlines. This is fine, and will work, but you will notice an effect called "flagging" at the top of the display which is ugly. A better approach is to continue to generate hsync pulses during the vsync. This keeps the horizontal sync timing hardware tracking. To accomplish this, the hsync is simply **inverted**. Both, the hsync and vsync filters will still detect the edges of the pulses, so the inversion doesn't hurt anything, so both the hsync and vsync occurs and the horizontal timing circuitry doesn't lose sync. Now, you might ask, "why wouldn't you always perform this inversion of the hsync during the vsync?". Well, the answer is that you might want to have once big continuous block of 4-6 lines that you set a single bit (sync) and then you can do work without interruption, but the video looks quite ugly, so it's a compromise.

In any event, considering this adjustment to our algorithm, all we have to do is generate 6 hsync pulses at the end of the bottom overscan that are inverted, thus, this new algorithm will suffice:

```
// Step 1: Draw the top over scan
For line = 1 to 8
  Begin
  Generate the next black scanline
End

// Step 2: Draw the active region of the scan, 240 lines
For line = 1 to 240
  Begin
  Generate the next scanline
End

// Step 3: Draw the bottom over scan
For line = 1 to 10
  Begin
  Generate the next black scanline
End

// Step 4: Generate a vertical sync pulse with serrated (inverted) hsync pulses,
Video = 0.0V for entire video line, expect for BLACK during hsync pulse
Output 6 blank lines with inverted hsync, this will keep horizontal tracking and cause a vsync to occur
```

11.2.5.2 Line Construction

Generating each line is a little more tricky than generating a frame since the real work must be done on each line. The basic idea is that you must generate the video signal which controls the luma (brightness), the chroma (color), and synchronization all within the same signal. This is accomplished by mimicking the signals you see in Figures 11.5 and 11.6. Each line consists of a total of 63.5 us where 52.6 us is actual video data and the other 10.9 us is sync and setup data. However, you can slightly alter them if you wish. For example, if you want to make the video portion of a line 50 us rather than 52.6 us then it will work since you can just stuff black into non-active region; however, the more you alter the spec (especially in terms of the length of the sync and color burst) the higher the chances are that the signal will not work on some older (or newer sets such as LCD and plasma with digital filtering).

To generate the actual video signal the Propeller chip gives you total control over the actual output voltage of the composite video line, therefore, you program the voltages as a function of time to create each video line using the VSU (video streaming unit). For example, each line consists of the following areas:

"Front Porch" - The spec calls for a "front porch" of 1.5 us consisting of black, therefore you would tell the Propeller hardware to send out black, then you would delay for 1.5 us (taking into consideration the amount of time to execute the actual instruction that turns black on).

"Sync Tip" - The next part of the spec is the horizontal sync or HSYNC, this should be approximately 4.7 us, therefore, you would tell the video hardware to output a 0.0V for 4.7 us.

The next portion of the video signal is the "Color Burst" which consists of a pre-burst phase called the "Breezeway", the burst itself called the "Color Burst" and finally the post-burst called the "Back Porch".

"Breezeway" - This part of the spec says to output black for 0.6 us.

"Color Burst" - This is the most complex part of the specification and the one that would be nearly impossible to do without the extra hardware support of the Propeller's VSU. In any case, we will explain how this works shortly, but for now, you need to know that you must generate 8-10 cycles of color burst tone. This is a 3.579594 MHz "tone" signal that the TV locks onto and uses as a phase reference for the remainder of the video line. The color of each pixel is determined by the

phase difference from the color burst reference at the beginning of each line and the color burst tone of each pixel. In any case, 8-10 clocks must be sent, I usually send 9-10 cycles of color burst. Each cycle at 3.579594 MHz takes 279.36 ns, therefore, if you want 10 cycles of color burst, you must turn the color burst hardware on for $279.36 \text{ ns} * 10 = 2.79 \text{ us}$ approximately.

“Back Porch” - Immediately following the color burst is the final part of the setup for the actual pixel data, this is called the “back porch” and should last 1.6 us.

11.2.6 Generating B/W Video Data

The remainder of the video information is 52.6 us, this is where you insert your pixel data. Now, if you wanted a B/W only signal then you would modulate the video signal from BLACK (0.3V) to WHITE (1.0V) for the remainder of the line and be done with it. For example, each line could rasterize a line buffer, or a sprite and different values would map to different voltages from BLACK to WHITE. With this approach most TVs have an input bandwidth wide enough to display about 320 luminance changes per active line, that means that no matter how fast you try to change the luminance signal only 320 times a line will you see anything. Let's see how we roughly estimate this.

The line length is 52.6 us, we want to make 320 changes in that time, that means that we need to send data at a rate of:

$$52.6 \text{ us} / 320 = 164.375 \text{ ns per change}$$

Inverting this gives us the frequency which is $1/164.375 \text{ ns} = 6.0 \text{ MHz}$ roughly! Ouch, that means that the input to the TV's luminance has to be 6.0 MHz or greater. Sorry to say, it's not. In most cases, you are lucky if you get 4.5 – 5.0 MHz input bandwidth, this 320 is a definitely upper limit on B/W luminance transitions per line.

11.2.7 Generating Color Video Data

Generating color video is much more complicated than B/W, however, if we take a practical approach rather than a mathematical, it's quite easy. Forgoing the complex quadrature encoding of color and luminance and the encoding and decoding of the signals, creating a color is very easy. For each color clock on the active scan line, you must generate a 3.579594 MHz sine wave, this must ride on top or be superimposed on the luminance signal (the Chameleon AVR hardware does all this). The overall amplitude of the signal is the brightness or luminance just as it was with B/W, but the color signal's saturation is simple the peak-peak (p-p) value of the color signal 3.579595 MHz signal as shown in Figure 7.6. The actual color that is displayed has nothing to do with the amplitude of the video signal, but only the PHASE difference from the reference burst from the original color burst reference at the beginning of each line.

To re-iterate, to generate color, we simply produce a 3.579594 MHz signal, superimpose or add it to the overall luminance signal, and the phase difference between our color signal and the reference is the color on the screen! Cool.

Taking this further, let's break up the line into pixels once again and see how many can be displayed each line. There are 52.6 us of active video time. We have to generate a 3.589594 MHz signal and stuff it into each pixel; given this how many pixels can fit into a line?

Calculation of Color Clocks Per Line

$$52.6 \text{ } \mu\text{s} / (3.579594 \text{ MHz}^{-1}) = 188.$$

This means that at best case you can have 188 colored pixels per active scanline, but it gets worst! That doesn't necessarily mean that you can have 188 DIFFERENT colored pixels across the screen, it just means you can request that from the poor TV with limited bandwidth. Again, this is a give take world, and in many cases, you would never have 188 different colors on the same line, it would look like a rainbow. In most cases, objects have constant color for a few pixels at a time. In any case, many video system over drive the video to 224, 240, or 256 virtual pixels (but, the color will not change that fast), you can do this if you wish, however, I suggest using a nice 160 x 192 display or thereabouts which will always look pretty good, has enough resolution, and you will almost get a 1:1 color change per pixel even if you change each pixel's color. However, give everything a try and see what you get.

11.2.8 NTSC Signal References

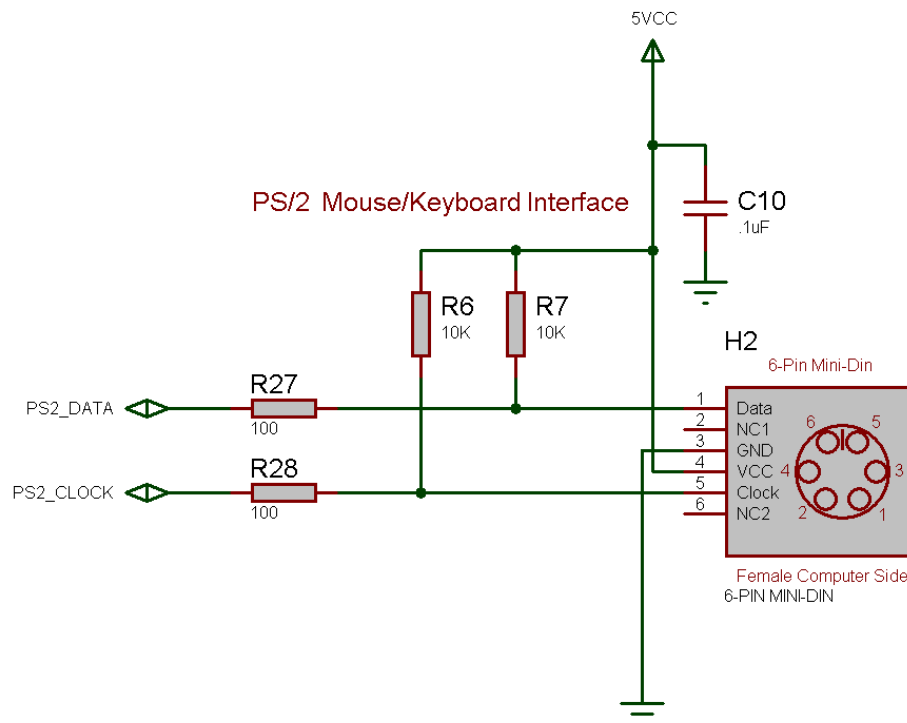
Here are a number of web sites and documents to help you understand the NTSC / PAL video formats:

<http://www.ntsc-tv.com/>
<http://www.sxlist.com/TECHREF/io/video/ntsc.htm>
<http://www.bealecorner.com/trv900/tech/RS170A.jpg>
<http://www.bealecorner.com/trv900/tech/>
http://www.maxim-ic.com/appnotes.cfm/appnote_number/734/In/en
<http://pdfserv.maxim-ic.com/en/an/AN734.pdf>

12.0 Keyboard & Mouse Hardware

PS/2 keyboards and mice are very similar from a hardware interface point of view. The interface is a simple serial system with two lines; **DATA** and **CLOCK**. Both lines are open collector, thus each end point typically can drive the line LOW, but there is a pull up that pulls the line(s) HIGH. Also, PS/2 interfaces are 5V, so when interfacing to a 3.3V system then some voltage translation must be performed. Additionally, we are interfacing the PS/2 port directly to the Propeller chip and since it's a 3.3V device care must be taken.

Figure 12.1 – The Chameleon AVR keyboard and mouse interface hardware.



As noted above, the keyboard (and mouse) are controlled via two bi-directional data lines; **CLOCK** and **DATA**. To interface to the Propeller, a pair of I/O lines are all that is needed. It would be nice if the serial protocol for keyboards and mice was identical to RS-232A then things would be a little easier and we could use one of the Propeller serial drivers, but the hardware and protocol is slightly different, so we have to use I/O pins and write our own driver. Or use one of the mouse/keyboard drivers already written for the Propeller which is the tactic we use for the Chameleon. More or less every single device interfaced uses a standard pre-written driver and all we do is make calls to it from the AVR chip. But, its nice to know how these things work, so let's take a look.

Table 12.1 below lists the pins and I/O map for the hardware interface.

Table 12.1 – PS/2 connector interface to Chameleon AVR.

Prop Port Bit/Pin	Chameleon Signal	Description
P27	PS2_CLOCK	Keyboard clock signal, open collector.
P26	PS2_DATA	Keyboard data signal, open collector.

NOTE

As noted, the PS/2 interface works the same for keyboards and mice, the protocols are different, but the low level serial and hardware interfaces are the same. Thus, if you plug a mouse into the port, it would work the same, the clock and data lines would be in the same place. Thus, all you would have to do is write the driver.

In the section below general keyboard and mouse operation is described, so you can write your own drivers if you wish. The driver we used for the Propeller driver supports both keyboard and mouse, but its not integrated thus a core is needed for each driver object – a bit wasteful.

12.1 Keyboard Operation

The keyboard protocol is straightforward and works as follows; for every key pressed there is a **"scan code"** referred to as the **"make code"** that is sent, additionally when every key released there is another scan code referred to as the **"break code"** that in most cases is composed of \$E0 followed by the original make code scan value. However, many keys may have multiple make codes and break codes. Table 12.2 lists the scan codes for keyboards running in default mode (startup).

Table 12.2 – Default scan codes.

KEY	MAKE	BREAK		KEY	MAKE	BREAK		KEY	MAKE	BREAK
A	1C	F0,1C		9	46	F0,46		[	54	F0,54
B	32	F0,32		`	0E	F0,0E		INSERT	E0,70	E0,F0,70
C	21	F0,21		-	4E	F0,4E		HOME	E0,6C	E0,F0,6C
D	23	F0,23		=	55	F0,55		PG UP	E0,7D	E0,F0,7D
E	24	F0,24		\	5D	F0,5D		DELETE	E0,71	E0,F0,71
F	2B	F0,2B		BKSP	66	F0,66		END	E0,69	E0,F0,69
G	34	F0,34		SPACE	29	F0,29		PG DN	E0,7A	E0,F0,7A
H	33	F0,33		TAB	0D	F0,0D		U ARROW	E0,75	E0,F0,75
I	43	F0,43		CAPS	58	F0,58		L ARROW	E0,6B	E0,F0,6B
J	3B	F0,3B		L SHFT	12	F0,12		D ARROW	E0,72	E0,F0,72
K	42	F0,42		L CTRL	14	F0,14		R ARROW	E0,74	E0,F0,74
L	4B	F0,4B		L GUI	E0,1F	E0,F0,1F		NUM	77	F0,77
M	3A	F0,3A		L ALT	11	F0,11		KP /	E0,4A	E0,F0,4A
N	31	F0,31		R SHFT	59	F0,59		KP *	7C	F0,7C
O	44	F0,44		R CTRL	E0,14	E0,F0,14		KP -	7B	F0,7B
P	4D	F0,4D		R GUI	E0,27	E0,F0,27		KP +	79	F0,79
Q	15	F0,15		R ALT	E0,11	E0,F0,11		KP EN	E0,5A	E0,F0,5A
R	2D	F0,2D		APPS	E0,2F	E0,F0,2F		KP .	71	F0,71
S	1B	F0,1B		ENTER	5A	F0,5A		KP 0	70	F0,70
T	2C	F0,2C		ESC	76	F0,76		KP 1	69	F0,69
U	3C	F0,3C		F1	05	F0,05		KP 2	72	F0,72
V	2A	F0,2A		F2	06	F0,06		KP 3	7A	F0,7A
W	1D	F0,1D		F3	04	F0,04		KP 4	6B	F0,6B
X	22	F0,22		F4	0C	F0,0C		KP 5	73	F0,73
Y	35	F0,35		F5	03	F0,03		KP 6	74	F0,74
Z	1A	F0,1A		F6	0B	F0,0B		KP 7	6C	F0,6C
0	45	F0,45		F7	83	F0,83		KP 8	75	F0,75

1	16	F0, 16		F8	0A	F0, 0A		KP 9	7D	F0, 7D
2	1E	F0, 1E		F9	01	F0, 01		]	5B	F0, 5B
3	26	F0, 26		F10	09	F0, 09		;	4C	F0, 4C
4	25	F0, 25		F11	78	F0, 78		'	52	F0, 52
5	2E	F0, 2E		F12	07	F0, 07		,	41	F0, 41
6	36	F0, 36		PRNT SCRN	E0, 12, E0, 7C	E0, F0, 7C, E0, F0, 12		.	49	F0, 49
7	3D	F0, 3D		SCROLL	7E	F0, 7E		/	4A	F0, 4A
8	3E	F0, 3E		PAUSE	E1, 14, 77, E1, F0, 14, F0, 77	-NONE-				

The keyboard hardware interface is either an old style male **5-pin DIN** or a new PS/2 male **6-pin mini-DIN** connector. The 6-pin mini DIN's pin out is shown in Figure 12.2 (referenced looking at the computer's female side where you plug the keyboard into, notice the staggering of the pin numbering).

Figure 12.2 - Female PS/2 6-Pin Mini Din Connector at Chameleon AVR socket.

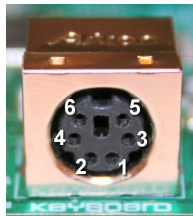


Table 12.3 lists the signals for reference, the descriptions of the signals are as follows:

DATA	Bi-directional and used to send and receive data.
CLOCK	Bi-directional; however, the keyboard nearly always controls it. The host can pull the CLOCK line LOW though to inhibit transmissions, additionally during host -> keyboard communications the CLOCK line is used as a request to send line of sorts to initiate the host -> keyboard transmission. This is used when commands or setting need to be sent to the keyboard.
VCC/GND	Power for the keyboard (or mouse). Specifications state no more than 100mA will be drawn, but I wouldn't count on it and plan for 200mA for feature rich keyboards with lots of lights etc.

When both the keyboard and the host are **inactive** the **CLOCK** and **DATA** lines should be **HIGH** (inactive).

Table 12.3 – Pin out of PS/2 6-Pin Mini Din.

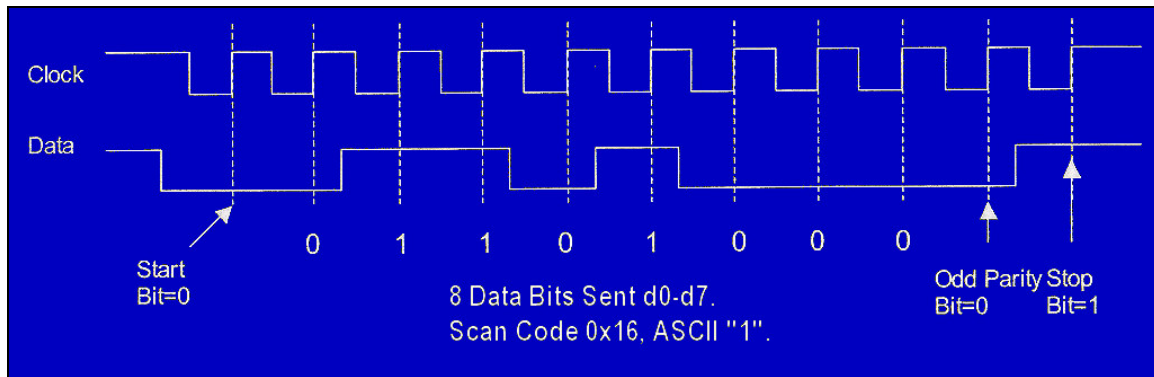
Pin	Function
1	DATA (bi-directional open collector).
2	NC.
3	GROUND.
4	VCC (+5 @ 100 mA).
5	CLOCK.
6	NC.

12.1.1 Communication Protocol from Keyboard to Host

When a key is pressed on the keyboard, the keyboard logic sends the **make scan code** to the host computer. The scan code data is clocked out by the keyboard in an 11-bit packet, the packet is shown in Figure 12.3. The packet consists of a **single LOW start bit** (35 μ s) followed by **8 data bits** (70us each), a **parity bit**, and finally a **HIGH stop bit**. Data should

be sampled by the host computer on the data line on the falling edge of the CLOCK line (driven by keyboard). Below is the general algorithm for reading the keyboard.

Figure 12.3 – Keyboard Serial Data Packet.



12.1.2 Keyboard Read Algorithm

The read algorithm makes the assumption that the main host has been forcing the keyboard to buffer the last key. This is accomplished by holding **CLOCK LOW**. Once the host releases the keyboard, then the keyboard will start clocking the clock line and **drop the DATA line** with a **start bit** if there was a key in the buffer, else the **DATA line will stay HIGH**. So the following steps are **after** the host releases the keyboard and is trying to determine by means of polling if there is a key in the keyboard buffer.

Step 1: Delay 5 μ s to allow hold on CLOCK line to release and keyboard to send buffered scan code.

Step 2 (Start of frame): If both CLOCK and DATA are low (start bit) then enter into read loop, else return, no key present.

Step 3 (Start of data): Wait until clock goes high...

Step 4 (Read data): Read data bits loop.

```
// pseudo code
for (t = 0; t <= 7; t++) {
    wait for CLOCK to go low...
    delay 5  $\mu$ s to center sample
    bit(t) = DATA;
} // end for
```

Step 5: (Read parity and Stop Bits): Lastly the parity and Stop Bits must be read.

And that's it! Of course, if you want to be strict then you should read the parity and stop bit, but you don't need to unless you want to perform error correction.

NOTE

Both the keyboard and mouse use an "**odd parity**" scheme for error detection. Odd parity is HIGH when the number of 1's in a string is ODD, LOW otherwise. Even parity is HIGH when the number of 1's in a string is EVEN, LOW otherwise. Note that parity only tells us there is an error, but not how to correct it, or where it was. You simply have to re-transmit the data.

12.1.3 Keyboard Write Algorithm

The process of sending commands to the keyboard or "**writing**" to the keyboard is identical to that when reading. The protocol is the same except the host initiates the conversation. Then the keyboard will actually do the clocking while the host pulls on the data line at the appropriate times. The sequence is as follows:

Step 1: Wait for the keyboard to stop sending data (if you want to play nice). This means that both the DATA and CLOCK lines will be in a **HIGH** state.

Step 2 (Initiate transmission): The host drives the CLOCK line **LOW** for **60us** to tell the keyboard to stop all transmissions. This is redundant if you waited for Step 1; however, the keyboard might not want to shut up, therefore, this is how you **force** it to stop and listen.

Step 3 (Data ready for transmission): Drive the DATA line **LOW**, then release the CLOCK line (by release we mean not to put a HIGH or a LOW, but to set the CLOCK into a tristate or input mode, so the keyboard can control it). This puts the keyboard into the "receiver" state.

Step 4 (Write data): Now, the keyboard will generate a clock signal on the CLOCK line, you sample the DATA line when the CLOCK line is **HIGH** and send it. The host program should serialize the command data (explained momentarily) made up of 8-bits, a parity bit, and a stop bit for a total of **11-bits**.

Step 5 (End transmission): Once the last data bit is sent then the parity (odd parity) and stop bit is sent then the host drives the DATA line (Stop bit) and releases the DATA line.

12.1.4 Keyboard Commands

Table 10.4 illustrates some of the commands one can send to a generic keyboard based on the 8042 keyboard controller originally in the IBM PC spec. This table is not exhaustive and only a reference, many keyboards don't follow the commands 100% and/or have other commands, so be weary of the commands.

Table 12.4 – Keyboard Commands.

Code Description

\$ED Set/Reset Mode Indicators, keyboard responds with ACK then waits for a following option byte. When the option byte is received the keyboard again ACK's and then sets the LED's accordingly. Scanning is resumed if scanning was enabled. If another command is received instead of the option byte (high bit set on) this command is terminated. Hardware defaults to these indicators turned off.

```

|7-3|2|1|0| Keyboard Status Indicator Option Byte
|  |  |  | `--- Scroll-Lock indicator   (0=off, 1=on)
|  |  |  | `----- Num-Lock indicator   (0=off, 1=on)
|  |  |  | `----- Caps-Lock indicator   (0=off, 1=on)
|  |  |  | `----- reserved              (must be zero)

```

\$EE Diagnostic Echo, keyboard echoes the \$EE byte back to the system without an acknowledgement.

\$F0 PS/2 Select/Read Alternate Scan Code Sets, instructs keyboard to use one of the three make/break scan code sets. Keyboard responds by clearing the output buffer/typematic key and then transmits an ACK. The system must follow up by sending an option byte which will again be ACK'ed by the keyboard:

```

00 return byte indicating scan code set in use.
01 select scan code set 1 (used on PC & XT).
02 select scan code set 2.
03 select scan code set 3.

```

\$F2 PS/2 Read Keyboard ID, keyboard responds with an ACK and a two byte keyboard ID of 83AB.

\$F3 Set Typematic Rate/Delay, keyboard responds with ACK and waits for rate/delay byte. Upon receipt of the rate/delay byte the keyboard responds with an ACK, then sets the new typematic values and scanning continues if scanning was enabled.

```

Typematic Rate/Delay Option Byte
|7|6|5|4|3|2|1|0|
|  |  |  | |---+---+---+--- typematic rate indicator (see INT 16,3)
|  |  |  | | `----- A in period formula (see below)
|  |  |  | | `----- B in period formula (see below)
|  |  |  | | `----- typematic delay
|  |  |  | | `----- always zero

```

$\text{delay} = (\text{rate} + 1) * 250$ (in milliseconds)

$\text{rate} = (8 + A) * (2^{**}B) * 4.17$ (in seconds, ñ 20%)

Defaults to 10.9 characters per second and a 500ms delay. If a command byte (byte with high bit set) is received instead of an option byte this command is cancelled.

- \$F4 Enable Keyboard, cause the keyboard to clear its output buffer and last typematic key and then respond with an ACK. The keyboard then begins scanning.
- \$F5 Default w/Disable, resets keyboard to power-on condition by clearing the output buffer, resetting typematic rate/delay, resetting last typematic key and setting default key types. The keyboard responds with an ACK and waits for the next instruction.
- \$F6 Set Default, resets to power-on condition by clearing the output buffer, resetting typematic rate/delay and last typematic key and sets default key types. The keyboard responds with an ACK and continues scanning.
- \$F7 PS/2 Set All Keys to Typematic, keyboard responds by sending an ACK, clearing its output buffer and setting the key type to Typematic. Scanning continues if scanning was enabled. This command may be sent while using any Scan Code Set but only has effect when Scan Code Set 3 is in use.
- \$F8 PS/2 Set All Keys to Make/Break, keyboard responds by sending an ACK, clearing its output buffer and setting the key type to Make/Break. Scanning continues if scanning was enabled. This command may be sent while using any Scan Code Set but only has effect when Scan Code Set 3 is in use.
- \$F9 PS/2 Set All Keys to Make, keyboard responds by sending an ACK, clearing its output buffer and setting the key type to Make. Scanning continues if scanning was enabled. This command may be sent while using any Scan Code Set but only has effect when Scan Code Set 3 is in use.
- \$FA PS/2 Set All Keys to Typematic Make/Break, keyboard responds by sending an ACK, clearing its output buffer and setting the key type to Typematic Make/Break. Scanning continues if scanning was enabled. This command may be sent while using any Scan Code Set but only has effect when Scan Code Set 3 is in use.
- \$FB PS/2 Set Key Type to Typematic, keyboard responds by sending an ACK, clearing its output buffer and then waiting for the key ID (make code from Scan Code Set 3). The specified key type is then set to typematic. This command may be sent while using any Scan Code Set but only has effect when Scan Code Set 3 is in use.
- \$FC PS/2 Set Key Type to Make/Break, keyboard responds by sending an ACK, clearing its output buffer and then waiting for the key ID (make code from Scan Code Set 3). The specified key type is then set to Make/Break. This command may be sent while using any Scan Code Set but only has effect when Scan Code Set 3 is in use.
- \$FD PS/2 Set Key Type to Make, keyboard responds by sending an ACK, clearing its output buffer and then waiting for the key ID (make code from Scan Code Set 3). The specified key type is then set to Make. This command may be sent while using any Scan Code Set but only has effect when Scan Code Set 3 is in use.
- \$FE Resend, should be sent when a transmission error is detected from the keyboard
- \$FF Reset, Keyboard sends ACK and waits for system to receive it then begins a program reset and Basic Assurance Test (BAT). Keyboard returns a one byte completion code then sets default **Scan Code Set 2**.

12.2 Communication Protocol from Mouse to Host

The mouse protocol is exactly the same as the keyboard protocol as far as sending and receiving bytes with the 11-bit packet. The only difference of course is the data format the mouse sends to the host and the commands the host (Propeller chip) can send the mouse. Again, we will cover real programming examples in the *Programming Manual*, but let's just review the technical details briefly to get acquainted with the commands and data.

12.2.1 Basic Mouse Operation

The standard PS/2 mouse interface supports the following inputs:

- X (right/left) movement.
- Y (up/down) movement.
- Left, Middle, and Right buttons.

The mouse has an **internal microcontroller** that translates the motion of either a **mechanical ball** or **optical tracking** system. The inputs along with the buttons are scanned at a regular frequency and updated are made to the internal "state" of the mouse via various counters and flags that reflect the movement and button states. Obviously there are mice these days with a lot more than 3 buttons and 2-axes, but we are not going to concern ourselves with these (extensions) we just need to read the X,Y position along with the state of the buttons.

The standard PS/2 mouse has **two** internal **9-bit 2's complement counters** (with an overflow bit each) that keep track of movement in the X and Y axis. The X and Y counters along with the state of the three mouse buttons are sent to the host in the form of a **3-byte data packet**. The movement counters represent the amount of movement that has occurred since the last movement data packet was sent to the host; therefore they are relative positions, not absolute.

Each time the mouse reads its inputs (controlled internally by the microcontroller in the mouse), it updates the state of the buttons and the delta's in the X, Y counters. If there is an overflow, that is if the motion from the last update is so large it can't fit in the 9-bits of either counter then the overflow flag for that axis is set to let the host know there is a problem.

The parameter that determines the amount by which the movement counters are incremented/decremented is the **resolution**. The default resolution is **4 counts/mm** and the host may change that value using the **"Set Resolution" (\$E8)** command. Additionally, the mouse hardware can do a scaling operation to the sent data itself to save the host the work. The **scaling** parameter controls this. By default, the mouse uses **1:1** scaling, which means that the reported motion is the same as the actual motion. However, the host may select **2:1** scaling by sending the **"Set Scaling 2:1" (\$E7)** command. If 2:1 scaling is enabled, the mouse applies the following mapping as shown in Table 12.5 the counters before sending their contents to the host in the data packet.

Table 12.5 – Mouse scaling reported data mapping when in 2:1 mode.

Actual Movement (delta)	Reported Movement
0	0
1	1
2	1
3	3
4	6
5	9
delta > 5	delta*2

So the scaling operation only takes affect when the actual movement delta is greater than 1, and for delta > 5, the reported movement is always **(delta*2)**. Now, let's look at the actual data packet format for the mouse state.

12.2.2 Mouse Data Packets

The PS/2 mouse sends the movement information to the host which includes the position counters, button state, overflow flags and sign bits in the format show in Table 12.6.

Table 12.6 – Mouse data packet format.

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 1	Y overflow bit	X overflow bit	Y sign bit	X sign bit	Always 1	Middle Button	Right Button	Left Button
Byte 2	X Movement (delta)							
Byte 3	Y Movement (delta)							

The movement counters are 9-bit 2's complement integers, since the serial protocol only supports 8-bits at a time, the upper most sign bit for each 9-bit integer is stored in Byte 1 of the overall movement packet. Bit 4 holds the 9th bit of the X movement and Bit 5 holds the 9th bit of the Y movement. Typically, you don't need to worry about the 9th bits since that would be a lot of motion in one update, so just use byte 2 and 3 to track motion.

The motion counters are updated when the mouse reads its input and movement has occurred. The movement counters once again only record differential or delta motion rather than absolute. With a 9-bit value recording each counter, a total amount of -255 to +256 can be represented in 9-bit 2's complement. If this range is exceeded, the appropriate overflow bit is set for either the X or Y counter. Note that the movement counters are reset whenever a movement data packet is successfully sent to the host. The counters are also reset after the mouse receives any command from the host other than the "Resend" (\$FE) command. Next, let's discuss the different mouse operation modes.

12.2.3 Modes of Operation

There are four standard modes of operation which dictate how the mouse reports data to the host, they are:

- **RESET** - The mouse enters **Reset mode** at power-up or after receiving the "**Reset**" (\$FF) command. For this mode to occur both the **DATA** and **CLOCK** lines must be **HIGH**.
- **STREAMING** - This is the default mode (after Reset finishes executing) and is the mode in which most software uses the mouse. If the host has previously set the mouse to Remote mode, it may re-enter Stream mode by sending the "**Set Stream Mode**" (\$EA) command to the mouse.
- **REMOTE** - Remote mode is useful in some situations and may be entered by sending the "**Set Remote Mode**" (\$F0) command to the mouse.
- **WRAP** - This diagnostic mode is useful for testing the connection between the mouse and its host. Wrap mode may be entered by sending the "**Set Wrap Mode**" (\$EE) command to the mouse. To exit Wrap mode, the host must issue the "**Reset**" (\$FF) command or "**Reset Wrap Mode**" (\$EC) command. If the "Reset" (\$FF) command is received, the mouse will enter Reset mode. If the "Reset Wrap Mode" (\$EC) command is received, the mouse will enter the mode it was in prior to Wrap mode.

RESET Mode - The mouse enters Reset mode at power-on or in response to the "**Reset**" (\$FF) command. After entering reset mode, the mouse performs a diagnostic self-test referred to as BAT (Basic Assurance Test) and sets the following default values:

- Sample Rate = 100 samples/sec.
- Resolution = 4 counts/mm.
- Scaling = 1:1.
- Data Reporting Disabled.

After Reset, the mouse sends a BAT completion code of either **\$AA** (BAT successful) or **\$FC** (Error). The host's response to a completion code other than \$AA is undefined. Following the BAT completion code of \$AA (ok) or \$FC (error), the mouse sends its **device ID** of **\$00**. This distinguishes the standard PS/2 mouse from a keyboard or a mouse in an **extended mode**.

After the mouse has sent its device ID of \$00 to the host, it will enter **Stream mode**. Note that one of the default values set by the mouse is "**Data Reporting Disabled**". This means the mouse will **not** issue any movement data packets until it receives the "**Enable Data Reporting**" command. The various modes of operation for the mouse are:

STREAM Mode - In stream mode, the mouse sends movement data when it detects movement or a change in state of one or more mouse buttons. The rate at which this data reporting occurs is the **sample rate** (defaults to **100 samples/sec** on Reset). This parameter can range from **10** to **200** samples/sec on most drivers. The default sample rate value is **100** samples/sec, but the host may change that value by using the **"Set Sample Rate"** command. Stream mode is the default mode of operation following reset.

REMOTE Mode - In this mode the mouse reads its inputs and updates its counters/flags at the current sample rate, but it **does not** automatically send data packets when movement occurs, rather the host must **"poll"** the mouse using the "Read Data" command. Upon receiving this command the mouse sends back a single movement data packet and resets its movement counters.

WRAP Mode - This is an **"echoing"** mode in which every byte received by the mouse is sent back to the host. Even if the byte represents a valid command, the mouse will not respond to that command -- it will only echo that byte back to the host. There are two exceptions to this: the **"Reset"** command and **"Reset Wrap Mode"** command, this is obviously the only way to get the mouse back out of the Wrap mode! The mouse treats these as valid commands and does not echo them back to the host. Thus Wrap mode is a good diagnostic mode to test if a mouse is connected and if its working.

12.2.4 Sending Mouse Commands

A mouse command similar to a keyboard command is sent using the standard **11-bit** serial protocol outlined in the Keyboard Write section. The commands supported are shown in Table 12.7.

Table 12.7 - Lists the set of command accepted by the standard PS/2 mouse.

Code	Description
\$FF	Reset - The mouse responds to this command with "acknowledge" (\$FA) then enters Reset Mode.
\$FE	Resend - The host can send this command whenever it receives invalid data from the mouse. The mouse responds by resending the last packet it sent to the host. If the mouse responds to the "Resend" command with another invalid packet, the host may either issue another "Resend" command, issue an "Error" command, cycle the mouse's power supply to reset the mouse, or it may inhibit communication (by bringing the Clock line low). The action taken depends on the host.
\$F6	Set Defaults - The mouse responds with "acknowledge" (\$FA) then loads the following values into its driver: - Sampling rate = 100. - Resolution = 4 counts/mm. - Scaling = 1:1. - Disable Data Reporting. The mouse then resets its movement counters and enters Stream mode.
\$F5	Disable Data Reporting - The mouse responds with "acknowledge" (\$FA) then disables Data Reporting mode and resets its movement counters. This only effects data reporting in Stream mode and does not disable sampling. Disabled Stream mode functions the same as Remote mode.
\$F4	Enable Data Reporting - The mouse responds with "acknowledge" (\$FA) then enables Data Reporting mode and resets its movement counters. This command may be issued while the mouse is in Remote mode (or Stream mode), but it will only effect data reporting in Stream mode.
\$F3	Set Sample Rate - The mouse responds with "acknowledge" (\$FA) then reads one more byte from the host which represents the sample rate in unsigned 8-bit magnitude format. The mouse saves this byte as the new sample rate. After receiving the sample rate, the mouse again responds with "acknowledge" (\$FA) and resets its movement counters. Most mice accept sample rates of 10, 20, 40, 60, 80, 100 and 200 samples/sec.

- \$F2 Get Device ID** - The mouse responds with "acknowledge" (\$FA) followed by its device ID (\$00 for the standard PS/2 mouse.) The mouse also resets its movement counters in most cases.
- \$F0 Set Remote Mode** - The mouse responds with "acknowledge" (\$FA) then resets its movement counters and enters Remote mode.
- \$EE Set Wrap Mode** - The mouse responds with "acknowledge" (\$FA) then resets its movement counters and enters Wrap mode.
- \$EC Reset Wrap Mode** - The mouse responds with "acknowledge" (\$FA) then resets its movement counters and enters the mode it was in prior to Wrap mode (Stream Mode or Remote Mode.)
- \$EB Read Data** - The mouse responds with acknowledge (\$FA) then sends a movement data packet. This is the only way to read data in Remote Mode. After the data packet has been successfully sent, the mouse resets its movement counters.
- \$EA Set Stream Mode** - The mouse responds with "acknowledge" then resets its movement counters and enters Stream mode.
- \$E9 Status Request** - The mouse responds with "acknowledge" then sends the following 3-byte status packet (then resets its movement counters) as shown below:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 1	Always 0	Mode	Enable	Scaling	Always 0	Left Button	Middle Button	Right Button
Byte 2	Resolution							
Byte 3	Sample Rate							

Right, Middle, Left button = 1 if button pressed; 0 if button is not pressed.

Scaling = 1 if scaling is 2:1; 0 if scaling is 1:1 (Refer to commands \$E7 and \$E6).

Enable = 1 if data reporting is enabled; 0 if data reporting is disabled (Refer to commands \$F5 and \$F4).

Mode = 1 if Remote Mode is enabled; 0 if Stream mode is enabled (Refer to commands \$F0 and \$EA).

- \$E8 Set Resolution** - The mouse responds with acknowledge (\$FA) then reads the next byte from the host and again responds with acknowledge (\$FA) then resets its movement counters. The byte read from the host determines the resolution as follows:

Byte Read from Host	Resolution
\$00	1 count/mm
\$01	2 count/mm
\$02	4 count/mm
\$03	8 count/mm

- \$E7 Set Scaling 2:1** - The mouse responds with acknowledge (\$FA) then enables 2:1 scaling mode.

- \$E6 Set Scaling 1:1** - The mouse responds with acknowledge (\$FA) then enables 1:1 scaling (default).

Lastly, the only **commands** the standard PS/2 mouse will send **to the host** are the **"Resend" (\$FE)** and **"Error" (\$FC)**. They both work the same as they do as host-to-device commands. Other than that the mouse simply sends 3-byte data motion packets in most cases.

NOTE

If the mouse is in Stream mode, the host should disable data reporting (command \$F5) before sending any other commands. This way, the mouse won't keep trying to send packets back while the host is trying to communicate with the mouse.

12.2.5 Mouse Initialization

During the mouse power up **both** the **DATA** and **CLOCK** lines must be pulled **HIGH** and/or released/tristated by the host. The mouse will then run its power up self test or BAT and start sending the results to the host. The host watches the **CLOCK** and **DATA** lines to determine when this transmission occurs and should look for the code **\$AA** (ok) followed by **\$00** (mouse device ID). However, you can forgo this step if you like and just wait 1-2 seconds and "assume" the mouse was plugged in properly. You do not have to respond to this code in other words. If you wish to **Reset** the mouse, you can at any time send the command code **\$FF**, and the mouse should **respond** with a **\$FA** to acknowledge that everything went well.

Once the mouse sends the **\$AA**, **\$00** startup codes then the mouse enters its standard default mode as explained in the sections above. The only thing we need to do to get the mouse sending packets is to tell the mouse to start reporting movement. This is done by sending the command "**Enable Data Reporting**" (**\$F4**), the mouse responds with **\$FA** to acknowledge the command worked and then will start streaming out 3-byte movement packets at the default rate of 100 samples/sec. This step is necessary, since on start up if the mouse simply started streaming data packets the host could lose important data, thus the mouse "**waits**" to be told to start streaming the data and reporting the motion.

The movement data packets are formatted as shown in Table 12.5. You simply need to read these packets and send them upstream to your application.

12.2.6 Reading Mouse Movement

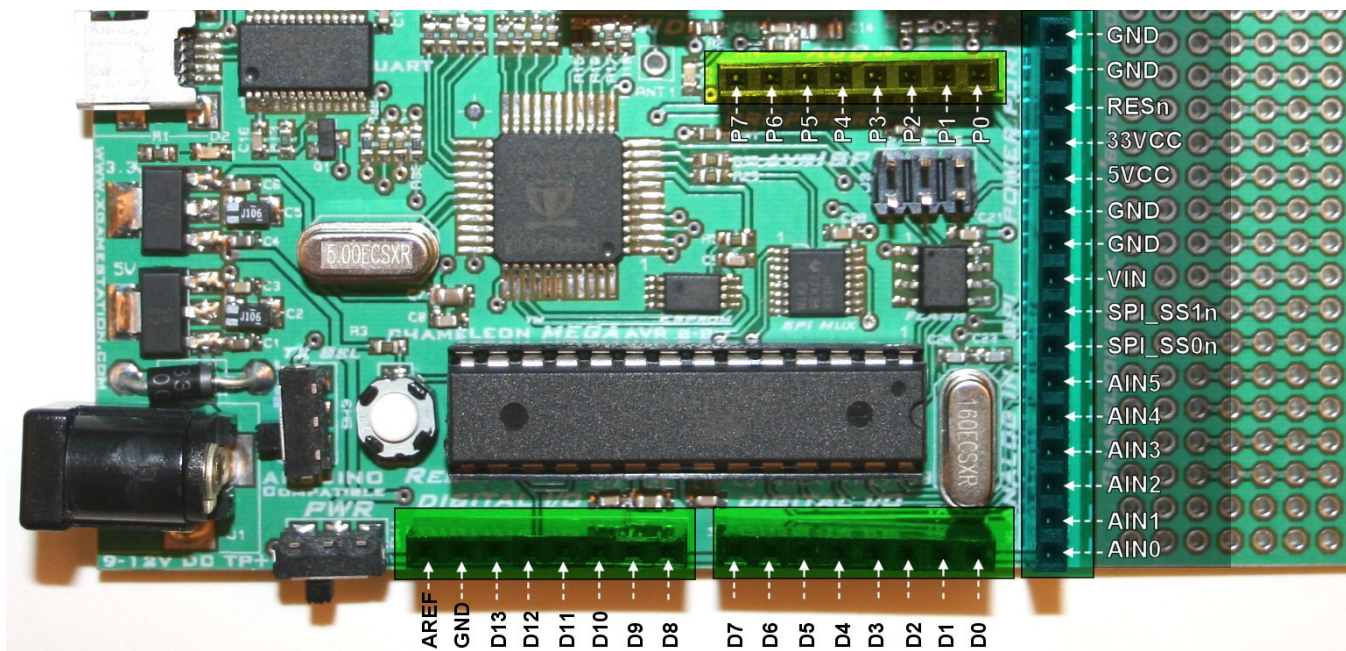
Assuming that the mouse has been initialized and is in **Streaming** mode with **Reporting** mode enabled, then you simply loop and read each 3-byte movement packet. As you read the first byte you use it to determine the state of the buttons as well as any overflows with the counters. Then the next two bytes, the X and Y deltas should be added to running counters (16-bit) that track the absolute position of the mouse cursor. That's it!

Of course, this is only for your edification, the Propeller driver does all this for you, so all you have to do is send a simple command from the AVR to the Propeller to read the keyboard or mouse.

13.0 The I/O Headers

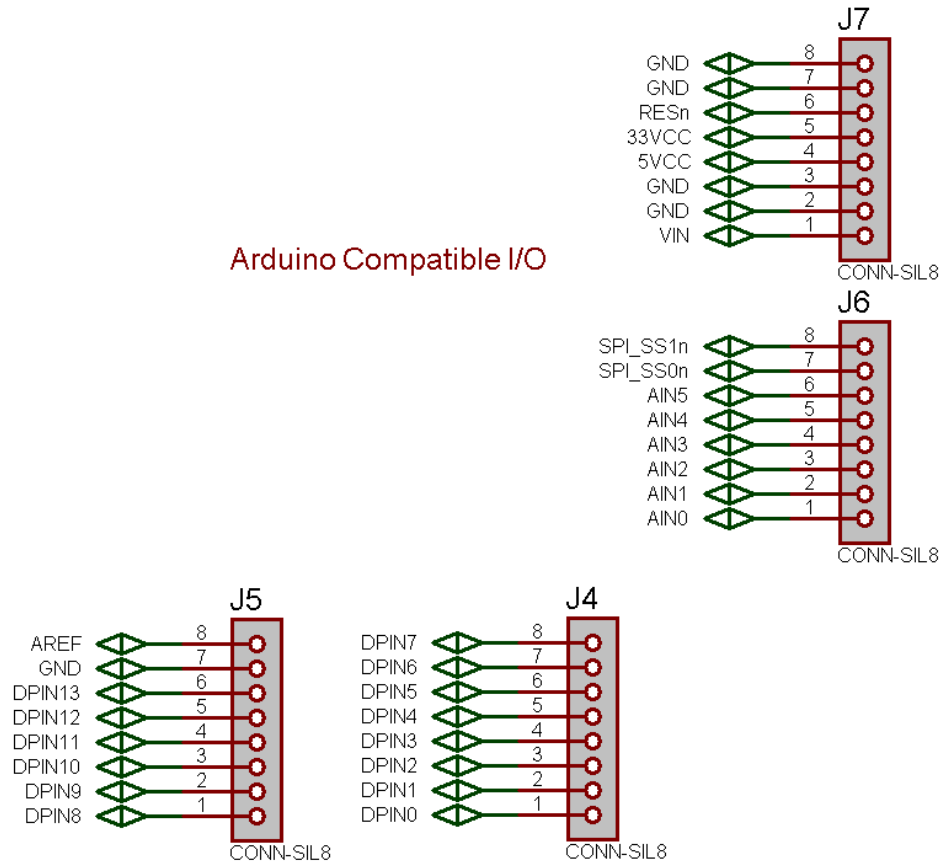
The I/O headers on the Chameleon AVR consist of a number signals that interface to both the AVR and Propeller chip. The Propeller chip has its own 8-bit local I/O port located right under the composite video and PS/2 ports. The other headers surrounding the Chameleon PCB interface mostly to the AVR chip. Let's take a look at the PCB for a moment for reference as shown in Figure 13.1.

Figure 13.1 – The Chameleon's annotated I/O headers.



Referring to the figure, there are two 8-pin I/O headers on the bottom of the board. A single 16-pin header on the right, and finally the 8-bit Propeller local port. The Propeller local port is discussed elsewhere, so let's concentrate on the other 3 headers. First off, on the Chameleon AVR version specifically we tried to map the Arduino compatible signal names to the same pins on the 328P chip, so programs written for Arduinos that use digital I/O would work with little modification. Similarly, we used the name naming conventions for the signals such as Dn for digital I/Os and AINn for analog inputs. However, the Chameleon has a few extra signals mostly on the larger 16-pin header on the right. Signals such as the multiple power sources as well as the SPI multiplexer chip selects **SPI_SS1n** and **SPI_SS0n** respectively. Figure 13.2 below illustrates the electrical design of the headers themselves laid out as they are on the PCB.

Figure 13.2 – I/O headers electrical design.



Also, the pins on the headers obviously connect to multifunction pins such as the serial UARTS, SPI, and so forth. Table 13.1 below maps the header pin names to the physical pin names and numbers for your convenience.

Table 13.1 – I/O header pinout map.

I/O Header Designator	I/O Header Pin Name/#	AVR Pin #	AVR Pin Name/Extra Function
J5	AREF	21	AREF
J5	GND ¹	8, 22	GND
J5	DPIN13	19	PB5 (SCK / PCINT5)
J5	DPIN12	18	PB4 (MISO / PCINT4)
J5	DPIN11	17	PB3 (MOSI / OC2A / PCINT3)
J5	DPIN10	16	PB2 (SSn / OC1B / PCINT2)
J5	DPIN9	15	PB1 (OCA1 / PCINT1)
J5	DPIN8	14	PB0 (PCINT0 / CLK0 / ICP1)
J4	DPIN7	13	PD7 (PCINT23 / AIN1)
J4	DPIN6	12	PD6 (PCINT22 / OC0A / AIN0)
J4	DPIN5	11	PD5 (PCINT21 / OCB0 / T1)
J4	DPIN4	6	PD4 (PCINT20 / XCK / TO)

J4	DPIN3	5	PD3 (PCINT19 / OC2B / INT1)
J4	DPIN2	4	PD2 (PCINT18 / INT0)
J4	DPIN1	3	PD1 (PCINT17 / TXD)
J4	DPIN0	2	PD0 (PCINT16 / RXD)
J6	SPI_SS1n	NA	NA
J6	SPI_SS0n	NA	NA
J6	AIN5	28	PC5 (ADC5 / SCL / PCINT13)
J5	AIN4	27	PC4 (ADC4 / SDA / PCINT12)
J6	AIN3	26	PC3 (ADC3 / PCINT11)
J6	AIN2	25	PC2 (ADC2 / PCINT10)
J6	AIN1	24	PC1 (ADC1 / PCINT9)
J6	AIN0	23	PC0 (ADC0 / PCINT8)
J7	GND ¹	8,22	GND
J7	GND ¹	8,22	GND
J7	RESn	1	PC6 (PCINT14 / RESETn)
J7	33VCC ¹	NA	NA
J7	5VCC ¹	20	VCC
J7	GND ¹	8,22	GND
J7	GND ¹	8,22	GND
J7	VIN ²	NA	NA

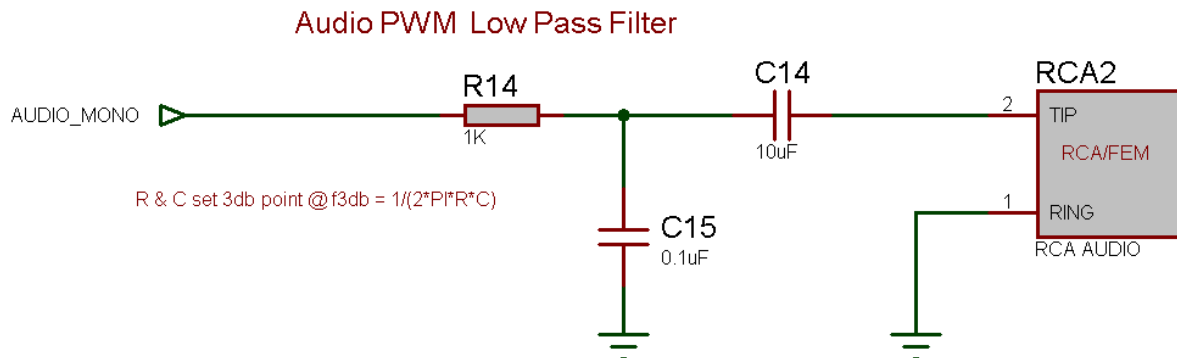
Note 1: Power pins 5VCC, 33VCC, GND don't actually connect to the AVR, but to the system power lines which in-turn are electrically connected to the AVR's AVCC, VCC, and GND respectively.

Note 2: VIN is connected to the 5V regulator's input pin. This signal reflects the input analog voltage plugged into the 9V DC power jack. If only the USB power is plugged in this signal will be floating or near ground.

14.0 Audio Hardware

There is no dedicated audio hardware inside the AVR chip nor the Propeller chip, therefore we have to rely on "**software techniques**" to create sound for the system. However, knowing that software is going to be used to generate sound we can add a little analog circuitry to "help" the audio software out and allow us to use certain common techniques for generating sounds. The Propeller does have counters on board and can generate PWM signals, thus the audio drivers typically leverage these hardware elements. Thus, for most audio applications that connect to a Propeller chip all you need is a typical PWM "integrator" or "low pass" filter. The Chameleon employs such a hardware design as shown in Figure 14.1.

Figure 14.1 – The analog audio hardware on the Chameleon AVR.



Referring to the circuit, we see that the signal **AUDIO_MONO** (I/O P24, pin 31 on the Propeller) is an input into the network. This signal is passed thru a low pass filter consisting of a resistor (**R14 @ 1K ohm**) and a capacitor (**C15 @ 0.1uF**). Note these reference designators may change in the future, but the point is there is an R/C network here. Moving

on there is also another AC coupling capacitor (**C14 @ 10uF**) to the final RCA output. The circuit has two sections and they each serve different purposes. Refer to Figure 14.1 for the following explanation.

14.1 A Little Background on Low Pass Filters (EE stuff)

The signal $V_{in}(t)$ comes in at the port and the **low pass filter** made up of **R14** and **C15** pass low frequencies, but block high frequencies. This is what's called a "**single pole RC filter**". The "**gain**" of the filter or the "**transfer function**" describes the relationship between the output and the input signal, thus a signal with a gain of 1 at the output, call it $V_{out}(t)$ would be equal to the input (with some possible phase variance, but this is irrelevant for our discussion). The single pole RC filter acts like a voltage divider, but we have to use more complex math to describe it based on imaginary numbers and/or differential equations. This is a tough, but there is a trick based on the **Laplace Transform** which transforms differential equations into algebraic equations (called the **S-Domain**) then we can work with the circuit as if it were made of resistors then transform back when done. This is all not that important, but its fun to know a little about analog stuff, so let's continue.

So given all that, we want to know what the voltage or signal is at the top of C15 (V_{out}) since this signal or voltage will then affect the remainder of the circuit. Using a voltage divider made of R14 and C15, we need to write a relationship between the input voltage at **AUDIO_MONO**, call it $V_{in}(t)$ and the output voltage of the RC filter at the top of C15, call it $V_{out}(t)$. Ok, here goes a few steps:

$$V_{out}(s) = V_{in}(s) * [(1/sC) / (R + 1/sC)]$$

Then dividing both sides by $V_{in}(s)$ we get,

$$\text{Gain} = H(s) = [(1/sC) / (R + 1/sC)]$$

Simplifying a bit, results in,

$$\text{Gain} = H(s) = 1 / (1 + sRC)$$

Then transforming back from the S-domain to the frequency domain we get,

$$\text{Gain} = H(f) = 1 / (1 + 2\pi fRC)$$

Note: Technically there is another term in there relating to phase, but it's not important for this discussion. In any event, now this is interesting, this equation:

$$\text{Gain} = H(f) = 1 / (1 + 2\pi fRC)$$

Describes the amplification or more correctly the attenuation of our signal as a function of frequency f . This is very interesting. Let's try something fun. Let's plug in some values really quickly and see what happens, let's try 0 Hz, 1 Hz, 100 Hz, 1000 Hz, 1 MHz and see what we get. Table 14.1 shows the results.

Table 14.1 – The results of passing various frequencies thru a low pass single pole RC filter.

Frequency f Hz	Gain	Comments
0	1/1	DC gain is 1.0 or no attenuation
1	$1/(1+2\pi fRC)$	
10	$1/(1+2\pi f10RC)$	
100	$1/(1+2\pi f100RC)$	
1000	$1/(1+2\pi f1000RC)$	
1,000,000	$1/(1+2\pi f1,000,000RC)$	

This is very interesting, ignoring for a moment the actually values of RC, we see that very quickly larger values of f (frequency) very quickly dominate the denominator and the quotient goes to 0. This is why this filter is called a "single pole", as the term in the denominator goes to infinity the quotient goes to zero. Ok, so how does this help us? Well, if we select values of RC, we can tune the frequency that this "attenuation" gets really strong. This is typically called the 3dB point, that is the point where the signal is attenuated by 3dB (decibels), not really important, to know what a decibels is, it's a measure of power or ratio of signals more or less, but what is important is that 3dB is about 70% of the signal, so if you want to filter out everything above 10KHz you would set the 3dB point for about 10KHz (maybe a bit more) and you

would see the signal get filtered. Also, note that at DC, frequency $f=0$, the right hand term in the denominator sum $(1 + 2\pi f RC) = 1$, thus the gain is $1/1$ or 1.0 which is exactly what it should be! Cool huh!

Filters can be a lot of fun since you can chain them together; low pass, high pass to make a band pass, or multiple low and high pass to make them fall off faster and so forth. Here's a cool tool on the web to get a "feel" for filters:

Low Pass: http://www.st-andrews.ac.uk/~www_pa/Scots_Guide/experiment/lowpass/lpf.html

High Pass: http://www.st-andrews.ac.uk/~www_pa/Scots_Guide/experiment/highpass/hpf.html

In any event, playing with the math, the 3dB point for our circuit is:

$$f_{3dB} = 1/(2\pi RC)$$

Notice, f is not in there since we solved for it. Therefore, we have everything we need to use the circuit and the equation. Let R and C in our circuit be $1K\ \Omega$ and $0.1\mu F$ respectively, plugging them in we get:

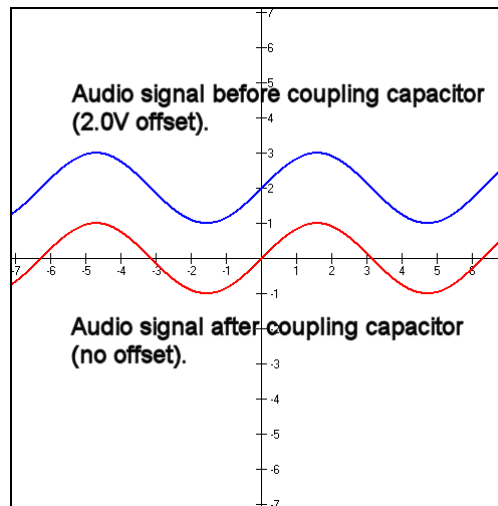
$$f_{3dB} = 1/(2\pi \cdot 1K \cdot 0.1\mu F) = 1.59\text{ KHz}$$

Which might be a little low, to loosen this up, let's make the resistor smaller – $200\ \Omega$:

$$f_{3dB} = 1/(2\pi \cdot 200 \cdot 0.1\mu F) = 7.95\text{ KHz}$$

Which is a rather high frequency about 50% the max range of most human ears which top out at **15-20KHz**.

Figure 14.2 – Sine wave riding on a constant DC offset as its passed thru a coupling capacitor.



Why is this important? Well, first off if you send a signal thru our little low pass filter to the output (connected to the audio port on the TV) then the signal is going to attenuate at high frequencies. We will get back to this momentarily, let's move on to the second stage in the audio circuit which is based on **C15**. This stage of the circuit is a AC pass filter, that means that it will only pass AC and the DC component will be blocked. In other words, say that the input was a sine wave or square or square wave with a peak to peak voltage of $1V$, but it was riding on a $2V$ signal, this would look like Figure 14.2 (top graph in blue). However, after going thru the AC coupling capacitor, the signal would look like that shown in Figure 14.2 (bottom graph in red). So all C15 does is block the DC. Now, let's talk about the some more hardware related concepts about making sounds with a couple specific techniques used for many of the Propeller audio drivers. Of course, you don't need to know this material since you are simply going to use a driver someone else writes (like the default drivers we provide), however, it can't hurt to have an understanding of how the sounds are generated typically by these drivers.

14.1.1 Pulse Code Modulation (PCM)

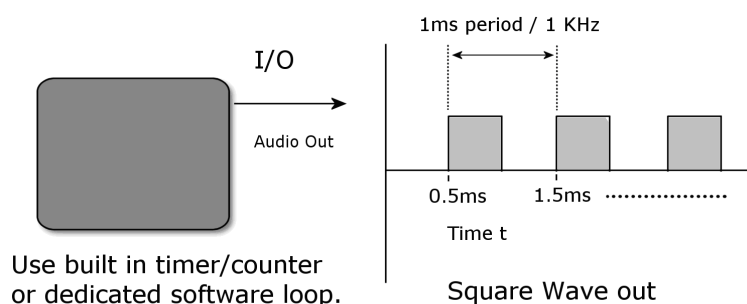
The most common method to make sound on a game console is to use **PCM** or **Pulse Code Modulation**. This is really nothing more than storing a series of samples of the sound in some resolution; $4, 8, 12, 16$ bit along at some playback rate. A .WAVE file for example is PCM data, you simply output the data at the proper rate to a D/A converter with an amplifier

connected to a speaker and you will hear the sound. There are a number of issues with this; first it takes huge amounts of memory (even with compression), secondly you need a D/A (digital to analog) converter on the hardware and there is no "synthesis" opportunities really (of course you can always synthesis the samples). PCM is a viable option for the Chameleon AVR, but there is so little FLASH memory (relatively speaking) you would have to stream off and external SD card. Thus, instead of digitizing sound effects, the technique used to save memory is to synthesize sound effects with various techniques. Still, you can definitely do PCM with the Chameleon AVR and or develop a "Mod" player if you wish, but at the output stage you only have a single bit, thus you have to use PWM to output your PCM data! More on this later.

14.1.2 Frequency Modulation (FM)

Frequency modulation with a fixed waveform is very easy to do and the Chameleon AVR can do this no problem. The idea here is to output a square wave or sine wave directly to the output device and then modulate the frequency. So if we use the Chameleon AVR's internal timers we can do this (more on this later) or we can write a software loop to do it. For example, if you wanted to hear a 500KHz, 1KHz, and 2KHz signal you just write a software loop that toggles the output **AUDIO_MONO** at that rate and you will hear the sound on the attached output device. Figure 14.3 shows this graphically.

Figure 14.3 – Synthesizing a single frequency.



Now, there are a couple problems with this; first it's not readily apparent how to "add" signals and play more than one sound at once. In fact, it's nearly impossible directly using this technique. Secondly, the only signal you can output is a square wave, you can't output sine waves. This tends to make the sounds have a "textured" sound since harmonics are in the square wave. That is if you output a square wave at frequency f then you will find that there are harmonics at $3f$, $5f$, etc. all these sine wave are what really make up a square wave. Take a look at **Fourier** transform theory to see this:

http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/OWENS/LECT4/node2.html

Of course our little low pass filter is going to filter most of these harmonics, but you aren't going to hear a pure sine until you increase the frequency to about the 3dB cutoff which maybe desired.

In any event, if all you need is a single channel, some noise, pure tones, then FM with the **AUDIO_MONO** port at (P24) is more than enough. You can use software or the built in timer in PWM mode to accomplish this as well, but you won't have any control over amplitude since you would be running the output at audio frequencies and not at PWM frequencies.

14.1.3 Pulse Width Modulation (PWM)

Pulse width modulation or PWM is a very clever way to create **any** kind of sound with a single bit of output! The bad news is that the generation of the output and the algorithms needed are fairly involved and take a little bit of numerical analysis, but once you get it working its not an issue and you can build libraries to create any sound you wish – in fact many advanced synthesizers use PWM systems, so it's very powerful.

To start with you might want to "Google" for PWM techniques and read up a bit. Watch out since most PWM references are about using PWM for motor control. Here are a few PWM articles to get your started:

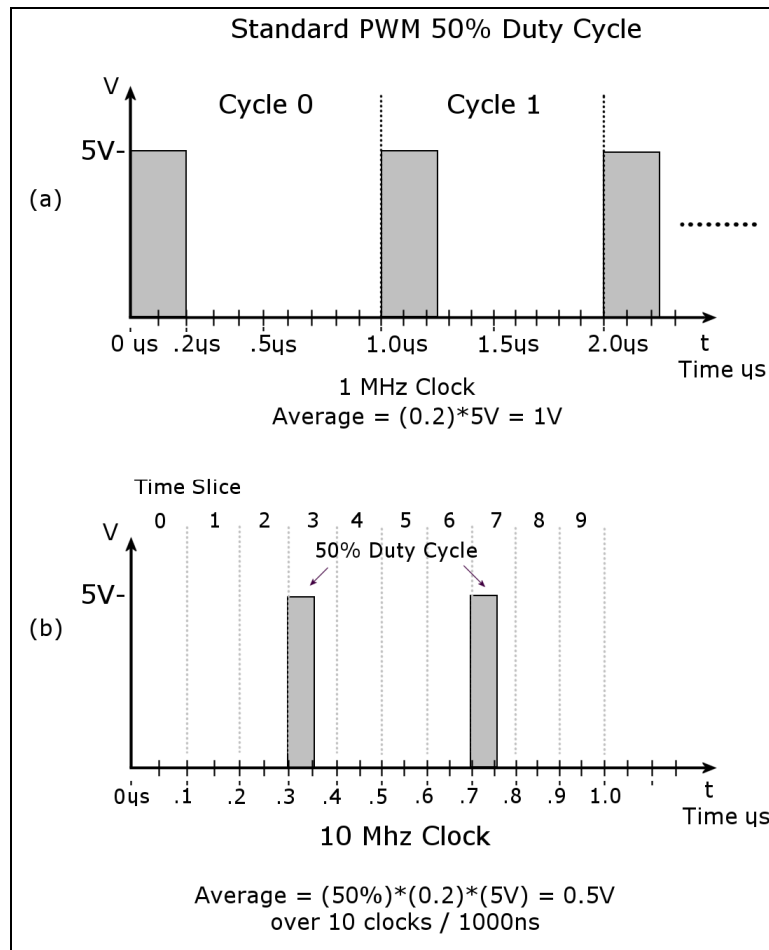
http://www.freescale.com/files/32bit/doc/app_note/MC68EZ328PWM.pdf

http://www.freescale.com/files/32bit/doc/app_note/MC68EZ328DTMF.pdf

<http://www.intel.com/design/mcs96/technote/3351.htm>

<http://ww1.microchip.com/downloads/en/AppNotes/00655a.pdf>

http://archive.chipcenter.com/knowledge_centers/embedded/todays_feature/showArticle.jhtml?articleID=10100668

Figure 14.4 - Duty cycle modes.

After reading all these documents you should have a fairly good grasp of PWM techniques. More or less, PWM is really a method of digital to analog conversion using a single bit output along with a low-pass filter that acts as an “**averaging**” or “**integration**” device. A PWM signal is at fixed output frequency usually many times the highest frequency you want to synthesis, for example a good rule of thumb is that the PWM signal should be 10-100x greater than the frequencies you want to synthesis. Also, the PWM period is **fixed**, the modulation of information in the PWM signal is in the duty cycle of the signal. Recall, duty cycle is defined as:

Duty Cycle = Time Waveform is HIGH / Total Period of Waveform

For example, say we had a 1KHz signal, this means the period is $1/1\text{KHz} = 1\text{mS} = 1000 \mu\text{s}$. Now, let's say we are talking about square waves in a digital system with a HIGH of 5V and a LOW of 0V. Furthermore, let's say that the duty cycle is 20%, what would the final waveform look like? Figure 9.4(a) depicts this. As you can see the waveform is HIGH 20% of the total period or 200 μs, and the waveform is LOW 800 μs. Therefore, if you were to average this signal $f(t)$ over time you would find that the average is simply the area under the HIGH part divided by the total AREA of the waveform which is:

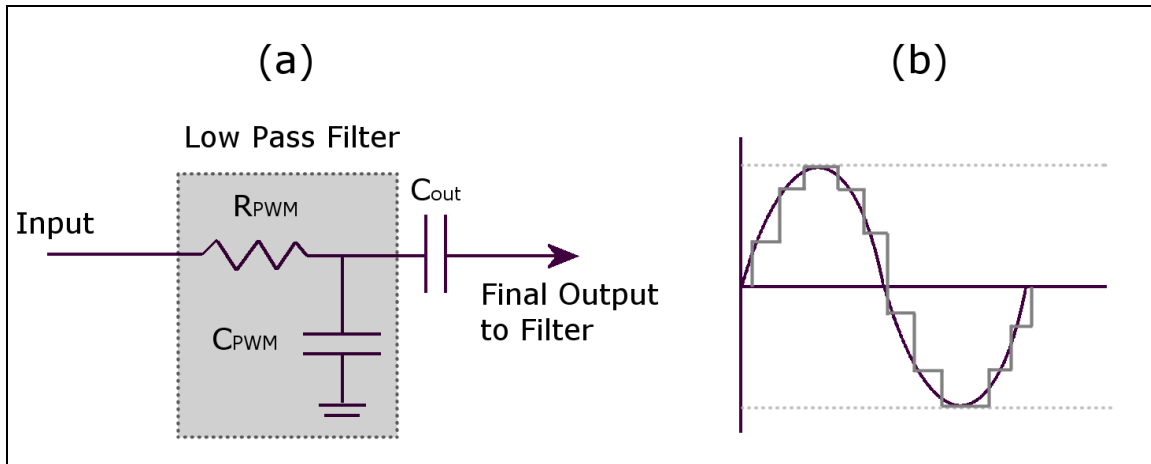
$$\text{Average Signal @ 20\% duty cycle} = (5V) * [(200 \mu\text{s}) / (1000 \mu\text{s})] = 1.00V.$$

This is a VERY interesting result, by just varying the time we pulse a digital output HIGH we can create an analog voltage! Thus, we can modulate the analog voltage in any shape we wish to create a final waveform of any shape we want; sounds, explosions, voices, etc. plus since this is all digital we can synthesis as many channels as we wish since at the end of the day we need only a single bit as the output. Figure 14.4(b) shows another PWM like technique where instead of modulating the duty cycle, instead complete 50% duty cycle pulses are sent, but they are interspersed with 0% duty cycles as a function of time. If the rate of these “pulses” is high enough, they too create an “average” voltage over time. In Figure 14.4(b) there are 10 total cycles and in 2 of them we have 50% duty cycles for a total analog voltage per 10 clocks of:

$$\text{Average Signal @ 20\% duty cycle} = (5V) * (50\%) * [(100 \text{ ns}) / (1000 \text{ ns})] = 0.5V.$$

Notice we are dealing in nanoseconds in the second example, this technique needs to run at a higher frequency to give the "average" enough time to change at the highest frequency rate you want to synthesize in this method of D/A.

Figure 14.5 – A low pass "averaging" filter and the "stair step" results.



Now, the only mystery component to the D/A PWM converter is "averaging" of the signal. Well, fortunately for us, a low pass filter as shown in Figure 14.5 acts as an averaging circuit, in fact, those of you with an EE background know that $1/S$ is integral in the S-Domain and a low pass filter has a $1/S$ term in it. Intuitively it makes sense as well since a low pass filter as shown in Figure 15.5(a) is really just a charging circuit and as we send these pulses to it, the circuit charges a bit, then another pulse comes and it charges a bit more, when a pulse doesn't come or the duty cycle is smaller then the RC circuit discharges, so the PWM modulated pulse train gets turned into a signal by an "**averaging**" process and looks like a stair step where you can see the averaging period superimposed on the signal. Figure 14.5(b) shows an example of this. Here we see a PWM signal running at frequency f_0 , and a sine wave being synthesized at 1KHz before and after the averaging circuit. We will get more into the programming of PWM when we discuss sound generation in the programming section of the manual. But, for now realize that the **low pass filter** (LPF) that the Chameleon AVR uses on the audio circuit acts as a LPF as well as an "averaging" circuit for PWM, so it has two uses – pretty cool.

14.1.3.1 Selecting the Filtering Frequency

Also, there are a couple of concerns with the actual values of the LPF, so that you don't get too much noise. Noise is going to come from the PWM frequency itself. For example, say you use a PWM frequency of 1MHz then your filter obviously needs to cut this frequency out, but it also needs to cut in at the highest frequency you plan to synthesize, for example, if you plan to have sounds all the way up to CD quality at 44KHz then you might set the 3dB point at 50KHz for example.

Also, we are using an "**passive**" or "**inactive**" single pole filter. You might want a "**sharper**" response in a more high end sound system like a 2 or 4 pole system. You can achieve this by daisy chaining our little LPFs together OR you can use an "**active filter**" that consumes power based on a transistor or better yet simple **Operational Amplifier** (Op Amp) like the **LM 741**. Or better yet pick an 8-pole switched capacitor filter by National, TI, or Linear Tech and the signal will hit a "brick wall" at the cutoff ☺

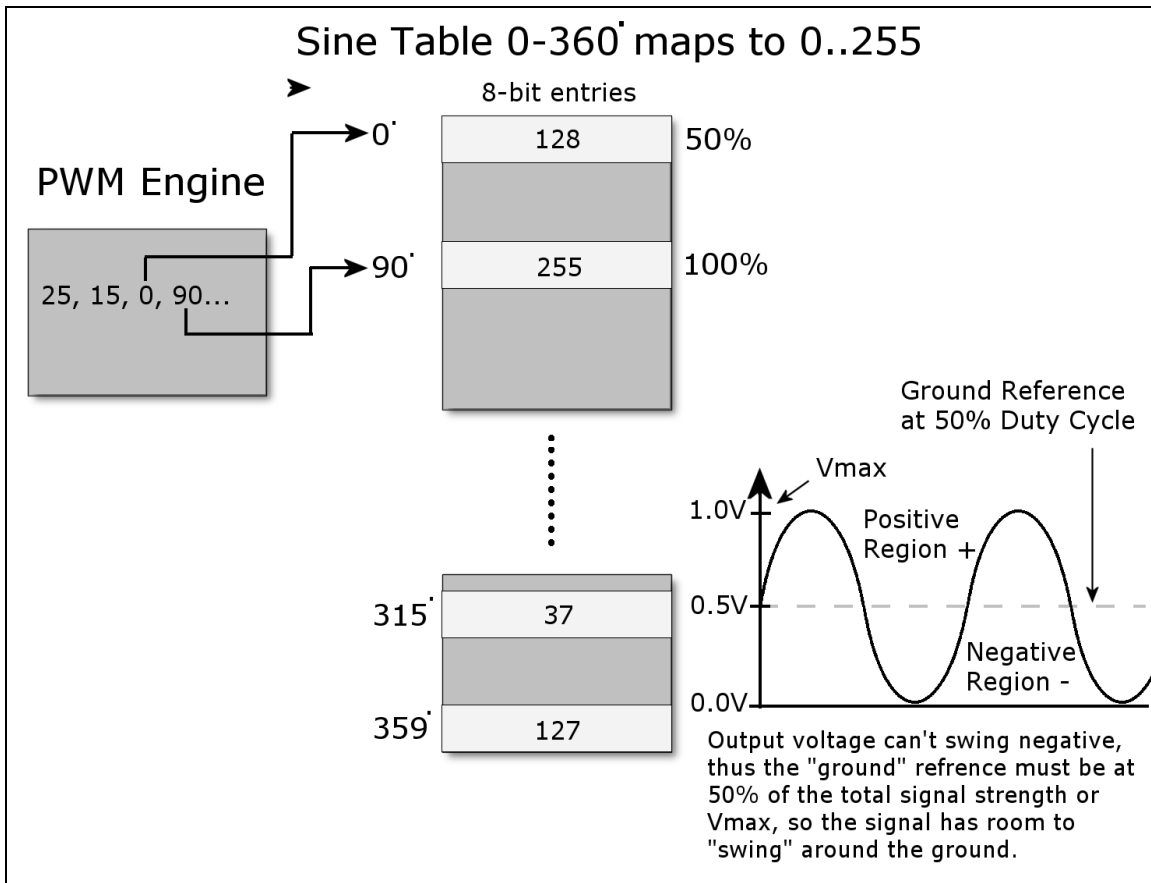
But, for our purposes; games, simple music, audio warnings, sound effects, etc. a single pole passive filter will do fine, if there is one thing I know, most people can't tell the difference between 16-bit, 96 KHz sound and 8-bit, 11 KHz, so sound generation can be a little rough!

14.1.3.2 PWM Setup Procedure

As mentioned, we will write some real code in the software section of this manual, but just to get you thinking about it, let's briefly digress a moment and work up an example with real numbers, so you can see this all in your head. Many of you are already sound experts and half of us have written MOD players and done DOS games for years, so this is old news, simple stuff, but a couple of you may never have played with sound, so this review can't hurt.

So to PWM modulate a "signal" we encode the signal onto the PWM carrier by means of encoding the "information" or analog value of the signal onto the PWM carrier by modulating the period or the duty cycle of the fixed frequency PWM carrier. This is a VERY important concept to understand – the PWM frequency/period NEVER changes, only the duty cycle of any particular cycle. Thus by modulating the information onto the duty cycle we can then later demodulate the signal by integrating or averaging the PWM signal with a RC circuit and presto we have an analog voltage!

Figure 14.6 – Indexing into a Sine look up table to synthesize a signal at a given PWM rate.



The first thing we need to do is decide what kind of waveforms we want to synthesis, remember they can be ANYTHING, they can be periodic and simple like sine, square wave (redundant), triangle, sawtooth, noise, or even digitized speech. But, to start simple, let's synthesis a single sine wave. So first we need a look up table for sine wave, let's call it **sinetable[]** and assume it has **256** entries and we generate it such that a single cycle has a low of 0 and a high of 255 and is encoded in 8-bits (similar to an example in the references). Now, the algorithm is fairly simple, we need to index thru the sine table a rate such that we complete a single cycle of our sine wave at the desired output signal frequency. As a quick and dirty example, let's say that we use a PWM frequency of 256 KHz and we want to play a 1 KHz sine wave, then this means that each second there are 256,000 PWM pulses, we want to index into our table and play 1000 iterations of our data which has a length of 256 entries, thus we need to index into our table at a rate of:

$$256,000 / (1000 * 256) = 1$$

Interesting, so every cycle we simply increment a counter into the sine table and output the appropriate duty cycle in the table lookup. This is shown in Figure 14.6. As another example, say we want to synthesis a 240 Hz signal then let's see what we would need:

$$256,000 / (240 * 256) = 4.16$$

In this case, we would increment a counter until it reached 4 then we would increment our index into our sine table. But, this example as well as the last should bring something to your attention, there is a max frequency we can synthesis and our signal synthesis is going to be inaccurate for anything, but integral divisors, so that's an issue we need to address in a moment. First, let's look at the maximum signal frequency, if you want to play back all 256 sine samples then the maximum "signal" frequency is always:

Maximum Synthesis Frequency = PWM frequency / Number of Samples per Cycle

Which in this example is,

$$256,000 / 256 = 1000 \text{ Hz}$$

So, you have two options; either increase the PWM frequency *or* index into your sample table at larger intervals. This problem along with the lack of precision can be handled by use of fixed point arithmetic and a little numerical trick. We are going to scale all the math by 8-bits or in essence multiply by 256 then we are going to create two variables one called a phase accumulator (**PHASE_ACC**) and one called a phase increment (**PHASE_INC**). Then instead of using count down algorithms, we are going to simply add the phase increment to the phase accumulator and use the phase accumulator as an index into the sine table. Therefore, what we have done is turned out 256 element sine table into a "virtual" $256 \times 256 = 65536$ element sine table for math purposes to give us more accuracy for slower, non-integral waveforms as well as allow fast waveforms to index and skip elements in the look up table, so a possible setup is something like this:

The Phase Accumulator 16-bit

b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
b15-b8 used as index								value							

Now, we have two interesting properties; first, no matter what the index in the upper 8-bits can never exceed 255, so we can never go out of bounds of our table, secondly, we can put very small numbers into the phase accumulator via that phase increment variable.

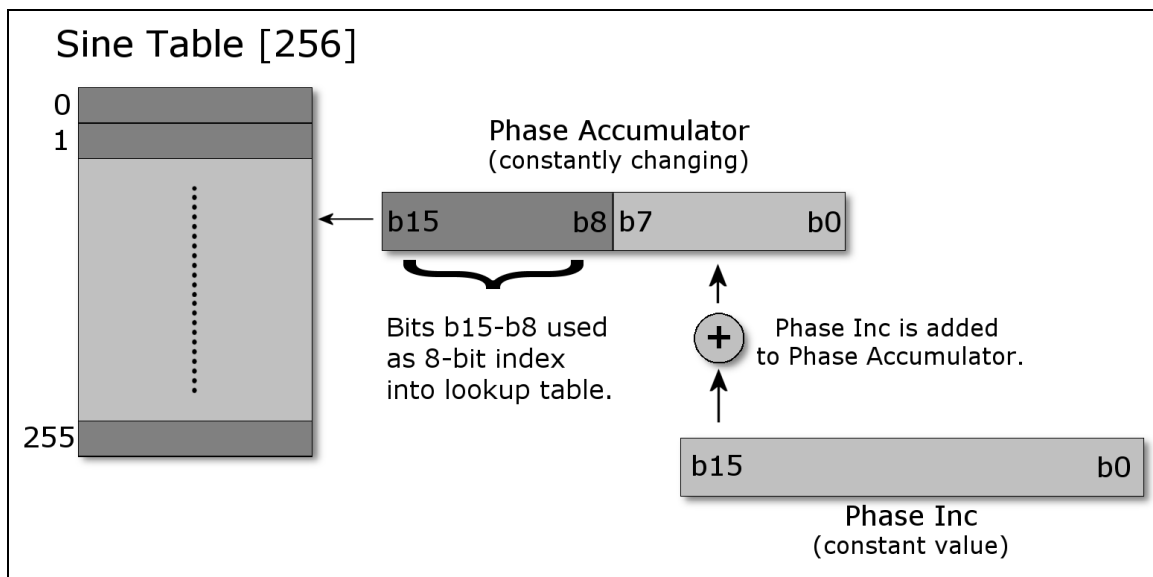
So now the algorithm for PWM sound is simple:

Step 1: Run the PWM at the output frequency 256,000 Hz.

Step 2: Calculate the Phase Increment (PHASE_INC) to add to the Phase Accumulator (PHASE_ACC) such that the final desired "signal" frequency is output via the lookup into the sine or waveform table.

Step 3: Continually add the Phase Increment to the Phase Accumulator and every cycle use the upper 8-bits of the Phase Accumulator as the index into the 256 element sine table.

Figure 14.7 – Our final PWM setup.



Now, to recap, we still have a table that has only 256 elements, but we are going to pretend it has 65536 elements, that is, 256×256 to improve our accuracy, this is nothing more than using a shift $\ll 8$ and create a fixed point value in 8.8 format. Next, we are going to call out a couple vars to make the algorithm easy, one is an accumulator called **PHASE_ACC** that simply accumulates the current count then we convert it back to an integer by shifting it $\gg 8$ times OR just using the upper 8-bits at the index into our 256 element sine table (the later is preferred). Then we simply need the magic number

PHASE_INC that for a given PWM frequency (256,000 in this case) and a desired output signal frequency along with a specific number of data table elements will make the whole thing work. Here's the final math:

Given,

1. A complete sine waveform has 65536 data entries in our virtual table and 256 in the real table.
2. The PWM frequency is 256K.

NUM_INCREMENTS = The number of times that the PWM signal increments thru the final signal table in ONE complete wave cycle. This is shown in Figure 14.7.

In other words,

$$\begin{aligned}\text{NUM_INCREMENTS} &= \text{signal period} / \text{PWM period} \\ &= \text{PWM frequency} / \text{signal frequency}\end{aligned}$$

Now, let's compute **PHASE_INC**,

$$\text{PHASE_INC} = 65536 / \text{NUM_INCREMENTS}$$

That is, the **PHASE_INC** is the rate at which we need to increment thru the data table to maintain the relationship so that the output is changing at the rate of the signal frequency. Now, plugging this all in together and moving things around a bit:

$$\text{PHASE_INC} = 65536 * \text{signal frequency} / \text{PWM frequency}.$$

And of course **PHASE_ACC** is simply set to 0 to begin with when the algorithm starts. As an example, let's compute some values and see if it works. First let's try a 1KHz signal and see what we get:

$$\text{PHASE_INC} = 65536 * 1\text{KHz} / 256,000 = 256.$$

So this means that we add 256 to the phase accumulator each PWM cycle, then use the upper 8-bits of the phase accumulator as the index, let's try it a few cycles as see of it works:

Table 14.2 – Test of PWM algorithm at 1KHz with PHASE_INC of 256.

Iteration	PHASE_INC	PHASE_ACC	PHASE_ACC (upper 8-bits)
0	256	0	0
1	256	256	1
2	256	512	2
3	256	768	3
4	256	1024	4
5	256	1280	5
6	256	1536	6
7	256	1792	7
.			
.			
255	256	65280	255

Referring to Table 14.2, we see that each cycle of the PWM the **PHASE_ACC** is incremented by 1 and the next element in the 256 element sine table is accessed, thus the table is cycled thru at a rate of $256,000 / 256 = 1,000$ Hz! Perfect! Now, let's try another example where the frequency is higher than what we could sustain with our more crude approach at the beginning of the section with only a counter and not using the accumulator and fixed point approach. Let's try to synthesize a 2550 Hz signal.

$$\text{PHASE_INC} = 65536 * 2550 / 256,000 = 652.8$$

Now, this is a decimal number which will be truncated during the fixed point math to 652, but this is fine, that's only an error or:

$$\text{Truncation error} = 100 * (0.8 / 652.8) = 0.12\%$$

I think I can live with that! Table 14.3 shows the results of the synthesis algorithm running once again.

Table 14.3 – Test of PWM algorithm at 2500 Hz with PHASE_INC of 652.

Iteration	PHASE_INC	PHASE_ACC	PHASE_ACC (upper 8-bits)
0	652	0	0
1	652	652	2
2	652	1304	5
3	652	1956	7
4	652	2608	10
5	652	3260	12
6	652	3912	15
7	652	4564	17
.			
.			
100	652	65000	253

As you can see in this case, each cycle the sine table is accessed by skipping an entry or two causing a bit of distortion, but since the table has 256 entries the amount of distortion is 1-2% at most, so once again we see that the technique works perfectly. Also, notice that it takes only 100 cycles of the PWM clock to iterate thru one complete cycle of the sine table which makes sense as well.

In conclusion, the software is where the magic is here. The PWM audio circuitry couldn't be simpler as shown in Figure 14.5. Additionally, you can synthesize multiple channels by simply having multiple phase accumulators and phase increments, in fact to synthesize a 64 channel system you would just need something like:

```
int phase_inc[64], phase_acc[64];
```

And you simply run a loop over everything and at sum up all the phase accumulators each cycle and use the sum as the index into the sine table. Of course, the sine table itself has to scaled in the amplitude axis and you must do a auto-scale, so that the sum doesn't overflow, but you get the idea. Moreover, you can store other waveforms like square, triangle, sawtooth, and so on and then mix various waveforms. Finally, you can easily overlay a **ADSR** (**a**ttack – **d**ecay – **s**ustain – **r**elease) envelope to a note based system and create just about anything. The only downside to PWM is that it must be VERY accurate, your ears are VERY sensitive to frequency shifts, so the timing loops must be perfect, thus a hardware timer or a tight loop needs to be used that doesn't vary otherwise, you will hear it.

In conclusion, that's a taste of what goes into generating sounds on the Propeller chip. Then there is the whole other process of creating a sound engine on top of these techniques to play "music" such as MIDI or MOD.

Part II – Programming Manual

15.0 Installing the Tool chains: AVRStudio, Arduino, and Propeller IDE

The first step in developing applications for the Chameleon AVR is to install the tool chains and all their components. Since the Chameleon is compatible with the **Arduino** tool chain as well as the standard **AVR Studio**, we are going to cover both methods with the emphasis on the more complex toolchain – AVR Studio. The Arduino tool is relatively easy to install, so we will cover that at the end of this section. Also, the Chameleon's AVR is **pre-loaded** with the Arduino bootloader in its flash memory. So if you don't have a AVR ISP MKII programmer, or you simply don't want to deal with AVRStudio you can work with the Arduino toolchain alone. However, I still recommend you skim the discussion of AVRStudio since it's a much more powerful tool than the Arduino tool.

Additionally, the Chameleon has a Parallax Propeller chip on it, so we have to install that tool as well, so there is a lot to do. We are going to show the installations of the tool in the following order:

- AVR Studio IDE
- Arduino Development System
- Propeller IDE

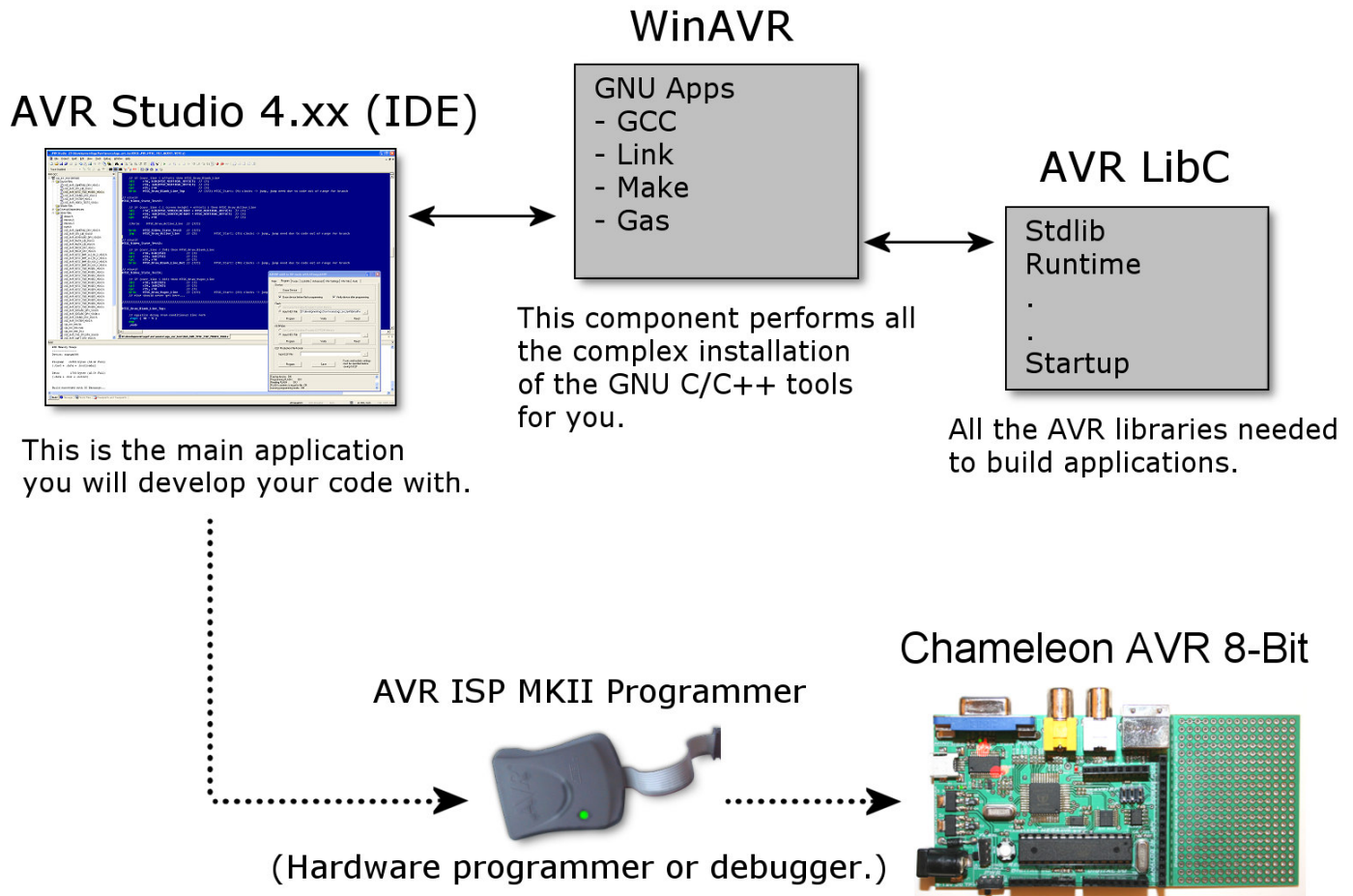
If you do not own a AVR ISP MK II or similar in-circuit-programmer then you can skim the first section.

15.1 Atmel's AVR Studio Toolchain Overview

Atmel's AVR Studio is the premier development tool for AVR processors. The tool is 100% free, support assembly language natively and now with the integration of WinAVR, the tool supports C/C++ seamlessly.

Like all tool chains this can be quite complex and a single incorrect checkbox can wreak havoc on many installation making them useless. For this reason, we will cover the installation of all main tools in the chain step by step. For those of you reading that are already expert in installation of AVR tools then you can skim this section and jump to the Arduino tool installation.

Figure 15.1 – The relationship of the various AVR programming tools in the tool chain.



Referring to Figure 15.1, you can see that there are three main components of the tool chain for the Chameleon AVR and AVR processors in general, they are:

- AVR Studio 4.xx
- AVRISP MKII (sold separately)
- WinAVR

AVR Studio 4.xx – This is the primary IDE developed by Atmel for programming the AVR line of processors. The tool comes complete with everything you need to program in assembly language, use simulators, debuggers and various programming tools developed by Atmel as well as 3rd party vendors. AVR Studio 4.xx does **NOT** have a built in compiler (only assembler). For this we will use a free **GNU GCC** compiler targeted toward AVR processors. Normally, this would be a very complex task to set up; however, there is another tool called **WinAVR** that eases this process for us.

AVRISP MKII – This is the actual hardware programmer that is recommended for the Chameleon AVR 8-Bit. The programmer only supports programming and not debugging, but is **low cost** and very versatile. The programmer must be installed as well and its firmware potentially updated. This installation will be done after the other tools are installed, so AVR Studio 4.xx specifically can update any drivers needed for the tool, since AVR ISP is integrated into the AVR Studio IDE itself.

WinAVR – This is a GNU GCC compiler installation that works on Windows that integrates the GNU GCC compiler and appropriate tools such as the **GNU Assembler**, **Linker**, and **Make** tools all in a single Windows installation package. Thus, after we install AVR Studio 4.xx, we will install WinAVR to give the IDE C/C++ support. In addition to the GCC compiler, the 3rd party run-time C/C++ library named "**AVR Libc**" is installed which you can leverage for your C/C++ programs. It more or less implements the standard C/C++ library, but additionally has a multitude of AVR and embedded system specific functions and libraries along with it.

AVR Studio 4.xx and WinAVR are located on the DVD-ROM at these locations:

DVD-ROM:\ CHAM_AVR \ TOOLS \ AVR \ AvrStudio417Setup.exe
DVD-ROM:\ CHAM_AVR \ TOOLS \ AVR \ WinAVR-20090313-install.exe

NOTE

AVRISP MK II (AVR ISP) is simply a piece of hardware and Windows will detect it and install the drivers from the AVR Studio install, thus there is no software per se that we need to worry about. However, if you wanted to install AVR ISP by itself then there are stand alone programming applications from Atmel. But, we aren't going to consider them since we will use AVR Studio which has programming and debugging built in.

15.1.1 Installing AVR Studio 4.xx (Optional)

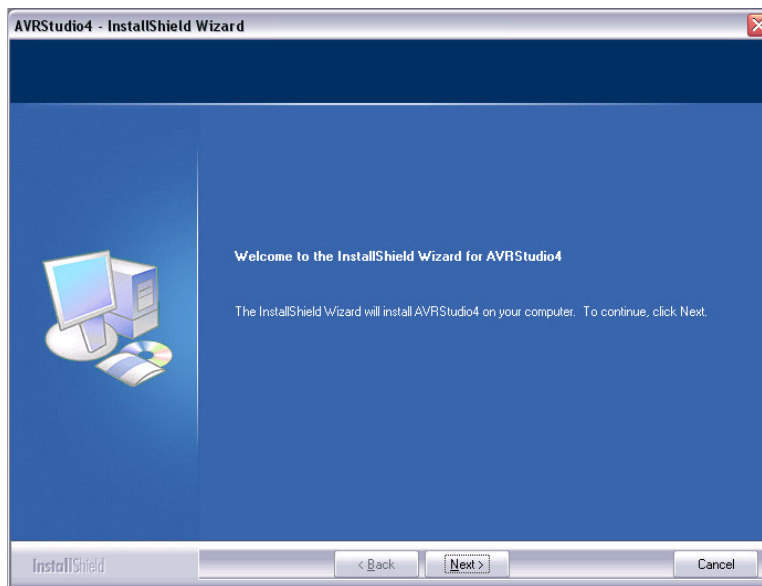
Installing AVR Studio is relatively straightforward as long as you know what you're doing to begin with. This is the problem with many IDEs and installation programs; you have to install the tool to figure out what **not** to do and then in many cases un-install the tool then re-install it. To save you all this grief, we are going to install the tool together step by step with a complete set of screen shots of each step along with explanations. Thus, without further ado, let's get this done. The first step is to locate and launch the AVR Studio 4.xx installer. The file is located on the DVD here:

DVD-ROM:\ CHAM_AVR \ TOOLS \ AVR \ AvrStudio417Setup.exe

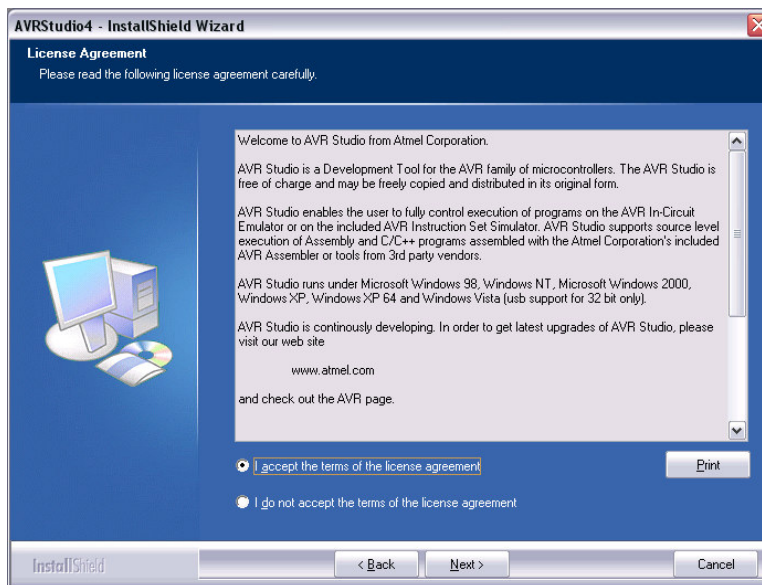
TIP

You can always download the latest copy of the install from **Atmel** directly; however, small changes in the installation and setup may de-synchronize what you see here in these page, thus, let's just use the same installation packages on the DVD. You can always update later **after** you have installed the complete tool chain and have it working.

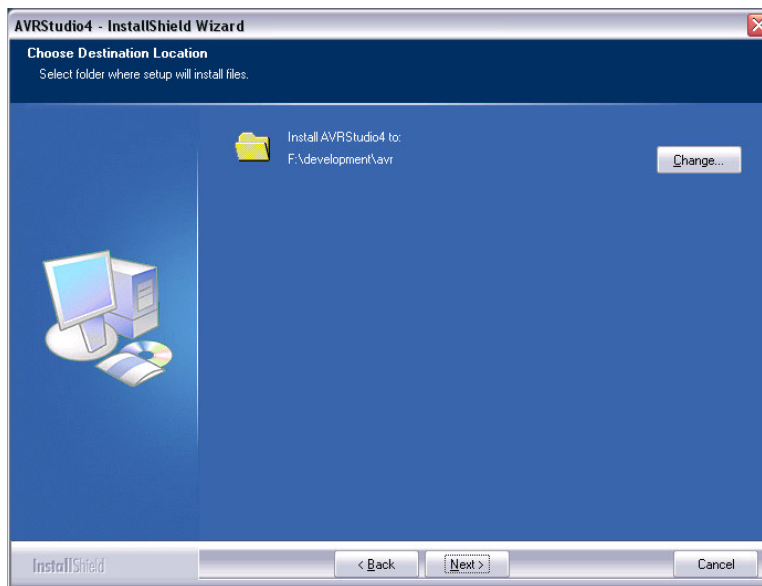
At this point, please launch the AVR Studio installation program. You should see the initial splash screen as show in Figure 15.2.

Figure 15.2 – Initial splash screen for AVR Studio 4 installer.

Press **<Next>** this will take you to the licensing screen shown in Figure 15.3 below.

Figure 15.3 – AVR Studio License Agreement.

The license screen is the usual you would expect, simply press **<Next>** to continue. This brings up the installation target directory as shown in Figure 15.4 below.

Figure 15.4 – AVR Studio installation directory selection.

I highly recommend you select a **“shallow”** installation directory for the files. In other words, don't put the installation in a long path with many sub-directories. I would install the software right off the root of one of your drives. This way, it's easy to type the path for command line applications. Moreover, use **“AVR”** in the suffix of the path for convention purposes, makes it easier to find the installation. Finally, avoid spaces in your directory names and I suggest not using drive **C:**, the less you put on your OS drive the better. You ever wonder why Windows gets slow after a while? The reason is that there are so many files on the install drive, DLLs in the Windows directory, etc. that the performance of Windows grinds to a halt. So anytime you can put apps on another drive or partition that's your best bet. **“My Programs”** is cute for soccer moms, but to be avoided as an installation target for serious programmers. Once you have selected the desired installation directory go ahead and press **<Next>** to continue the installation process.

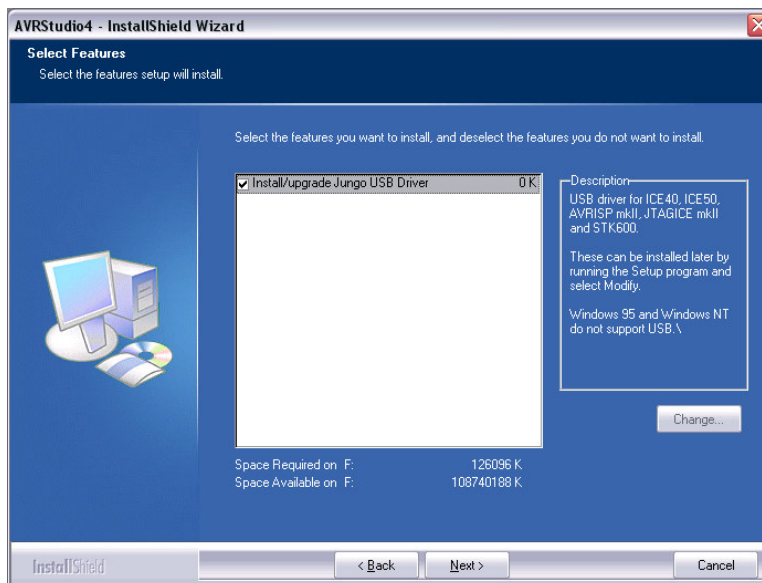
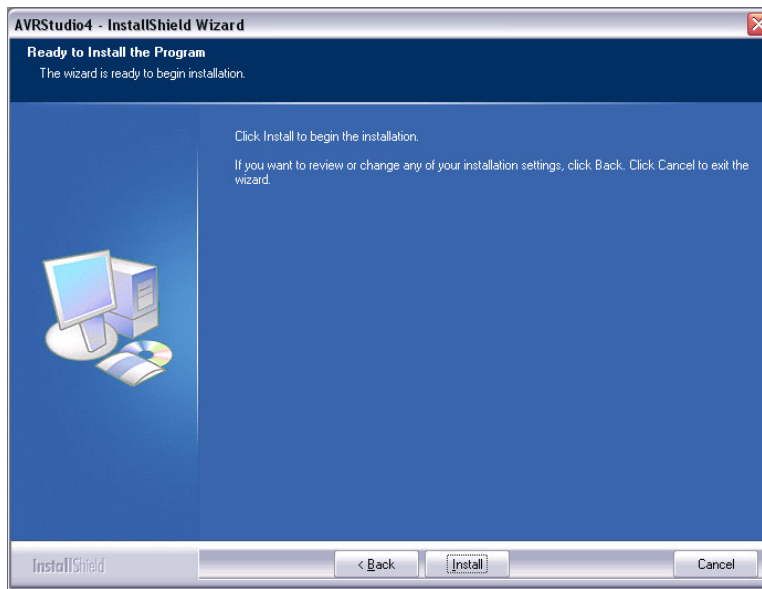
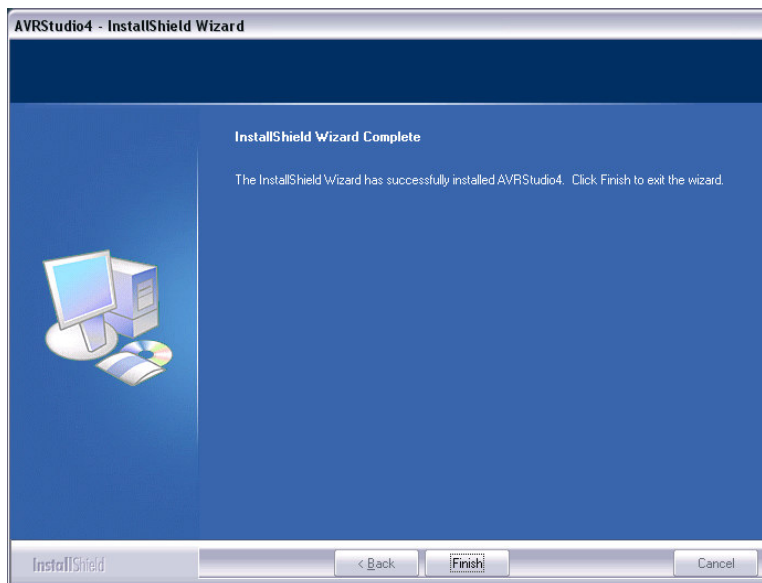
Figure 15.5 – AVR Studio extra installation features dialog.

Figure 15.5 shows the AVR Studio features installation dialog. As you can see, there's not many features. Actually, they all come by default. In any event, select the **USB driver upgrade** to save you the time of installing it later. This option includes the needed driver for any external USB based programmer/debugger which we need. Press **<Next>** and the final installation confirmation dialog will display as shown in Figure 15.6.

Figure 15.6 – AVR Studio final installation dialog.

Unless you want to go back and make changes, simply press **<Install>** and the installation will begin. If all goes well, then the installation complete message box will be displayed as shown in Figure 15.7.

Figure 15.7 – AVR Studio installation complete message box.

You're probably thinking, "that was easy!". I agree, but installation isn't really the hard part, it's the **"setup"** of the tools after installation! With that in mind, let's keep moving on with the installation of **AVR ISP MKII**.

TIP

Make sure to reboot your machine to allow the system to configure and all the environment variables to get set correctly. The WinAVR step of the installation depends on the previous installation of AVR Studio, so we want to make sure everything is clean.

15.1.2 Installing the AVR ISP MKII Hardware (Optional)

Now that AVR Studio is installed, we could install WinAVR next, but I find that installing the programming hardware is a better idea, so the drivers can load and it can register itself, so that WinAVR will **find** it during its installation. This way we don't get into a "**chicken and the egg**" problem and have to install something two times.

TIP

Of course, if you do not have a **AVR ISP MKII** then you can skip this section since you are probably planning on using the pre-flashed bootloader in Arduino mode over the USB serial connection.

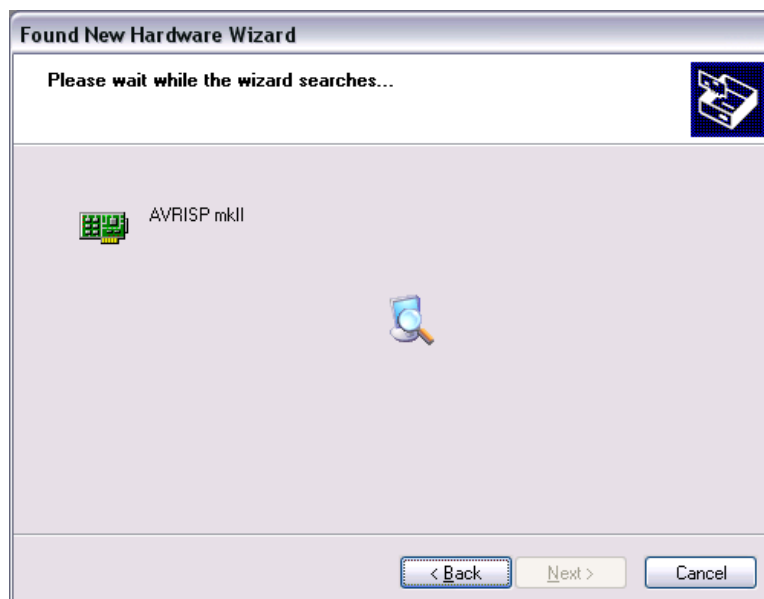
Installing the AVR ISP MKII programmer consists of nothing more than plugging it into your PC with a USB cable. Go ahead and do this now as. Once you plug it in, the PC should detect it and begin the installation process as shown in Figure 15.8 below.

Figure 15.8 – Window's detecting the Atmel AVR ISP MKII.



Select the radio button to install the software automatically and then click **<Next>**. Unless something went wrong with the AVR Studio installation you should see the hardware being installed as shown in Figure 15.9 below.

Figure 15.9 – Windows installation AVR ISP hardware and drivers.



When complete simply press **<Next>**, then the drivers should install. If all went well, you should see the final splash screen shown in Figure 15.10.

Figure 15.10 – AVR ISP drivers successfully installed.



Press **<Finish>** and the programmer should be ready to go.

15.1.3 Installing WinAVR™

WinAVR is a complete open source Windows based installation of the GNU GCC compiler and tool chain that integrates into the installation of AVR Studio. You can find out more here:

<http://winavr.sourceforge.net/>

We are very lucky to have this, without it, we would have to use a 3rd party compiler which costs lots of \$\$\$\$. On the other hand, it's GNU-GCC based which means very little support and documentation written by paid professionals, so it's a compromise. As mentioned before WinAVR integrates into AVR Studio giving AVR Studio C/C++ support (along with the GCC assembler named "**GAS**"). Before WinAVR you could still get AVR processor GCC support, but you would have to do a lot of work to accomplish this, thus the installer for WinAVR performs all kinds of really complex and tedious setup tasks and hides all the complexity from you.

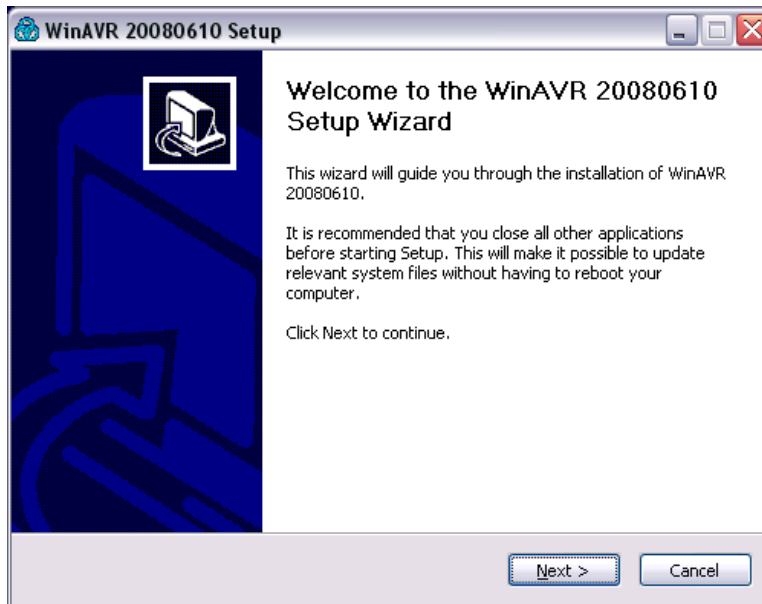
With that in mind, let's begin the installation. The latest WinAVR installer is located on the DVD here:

DVD-ROM:\ CHAM_AVR \ TOOLS \ AVR \ WinAVR-20090313-install.exe

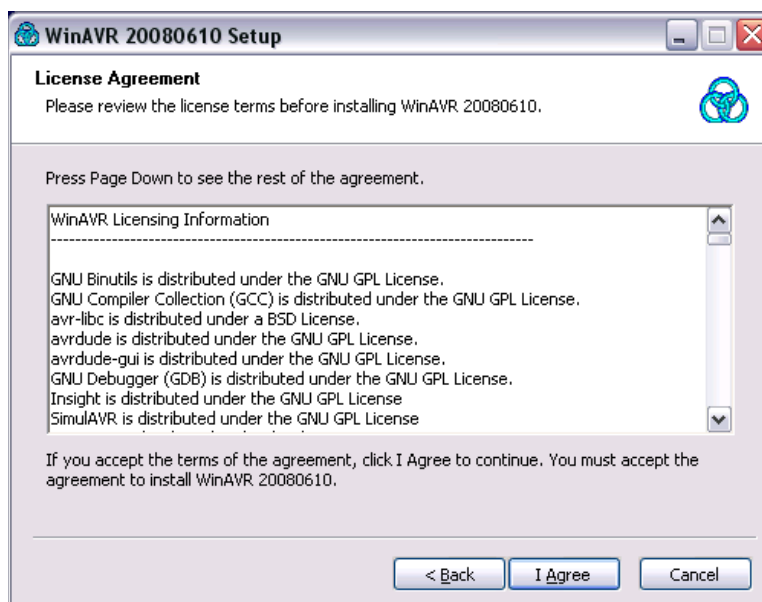
Of course, you can always download the very latest copy to the minute from the **Source Forge** site, but once again, let's just use the one on the DVD-ROM, so we both have the exact same software. You can update in the future once you have everything working. Once you have located the installation file, go ahead and launch it, you will see the language splash screen as shown in Figure 15.11 below (ignore the version differences from our screen shots to the latest copy on the DVD).

Figure 15.11 – WinAVR language selection.

Select English and click **<Ok>**, this will bring you to the main splash screen shown below in Figure 15.12.

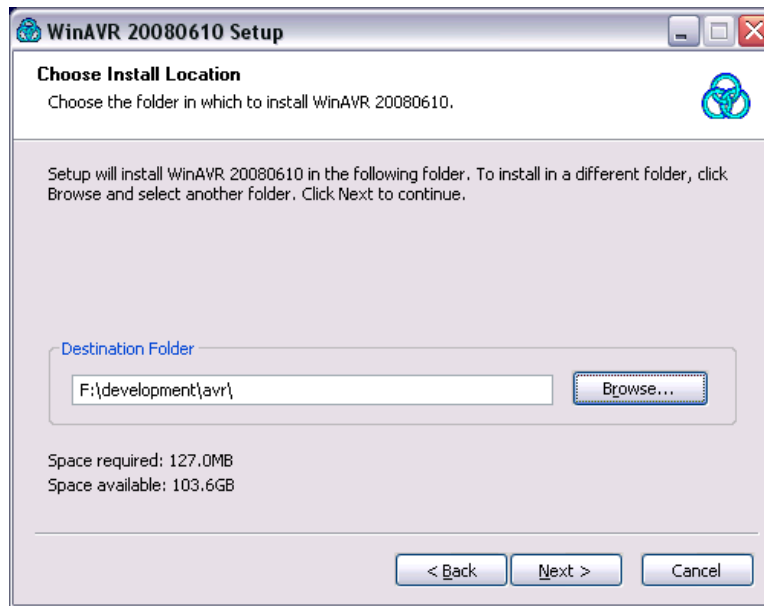
Figure 15.12 – WinAVR main splash screen.

Click **<Next>** and the famous GNU GPL license agreement should display on your screen. Read it and then click **<I Agree>** to continue as shown in Figure 15.13 below.

Figure 15.13 – WinAVR's GPL license.

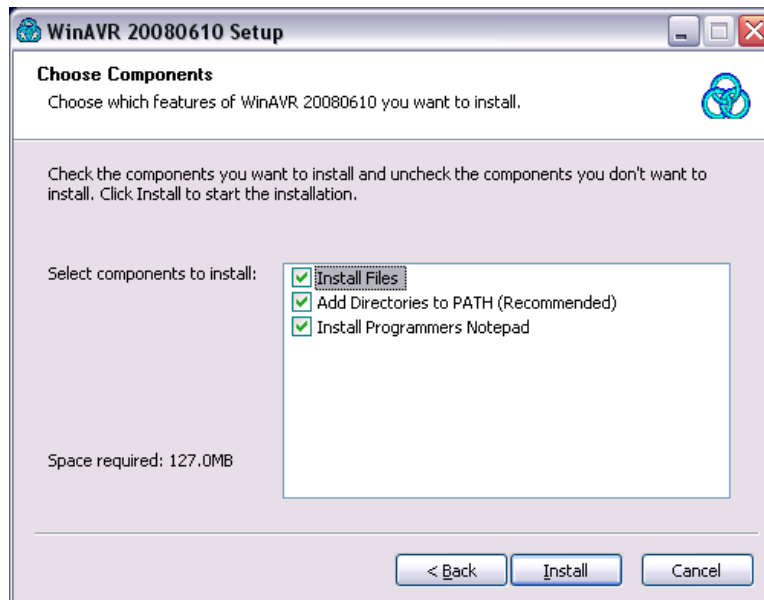
The next step in the installation is the selection of the installation file directory. Browse and select the **same root directory** you previously installed AVR Studio into. By doing this, everything will be conveniently located in a single directory. Figure 15.14 illustrates this.

Figure 15.14 – Selection of WinAVR's installation directory – should be the same as AVR Studio.



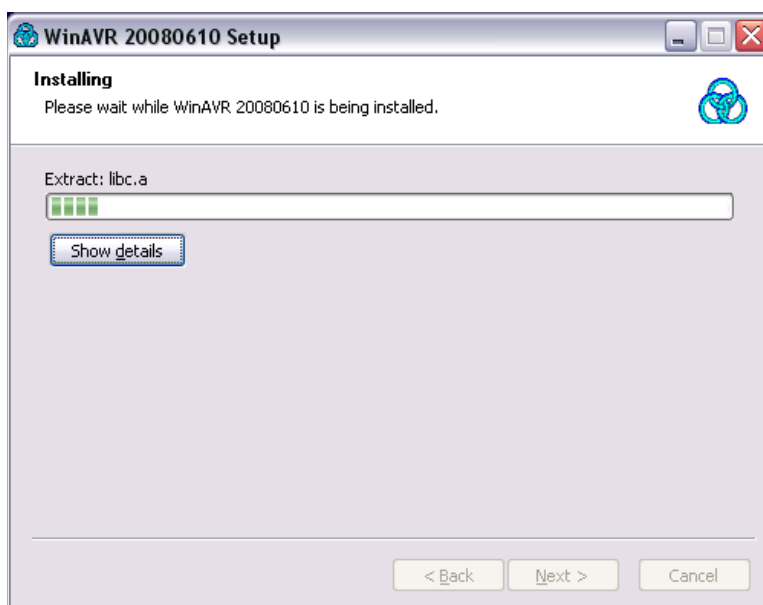
Once you are satisfied with the installation directory location, click **<Next>** for the installation components dialog as shown in Figure 15.15 below.

Figure 15.15 – WinAVR software components selection dialog.



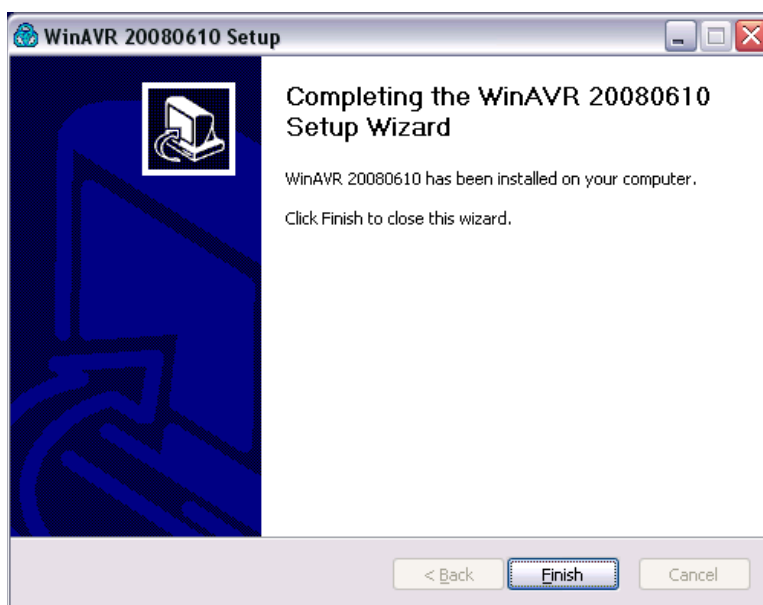
The WinAVR components selection dialog is straightforward and mostly redundant (of course I want to install the files!). In any event, simply check all checkboxes and then click **<Install>** to initiate the installation process. The installation progress dialog should display as shown in Figure 15.16 below.

Figure 15.16 – WinAVR 20080610 installation progress dialog.



The installation is rather quick (under a minute). When complete, the installation finished message box will display, click **<Finish>** to complete the process and you can stop holding your breath ☺

Figure 15.17 – WinAVR successfully installed!



15.1.4 Building a Project and Testing the Tool Chain

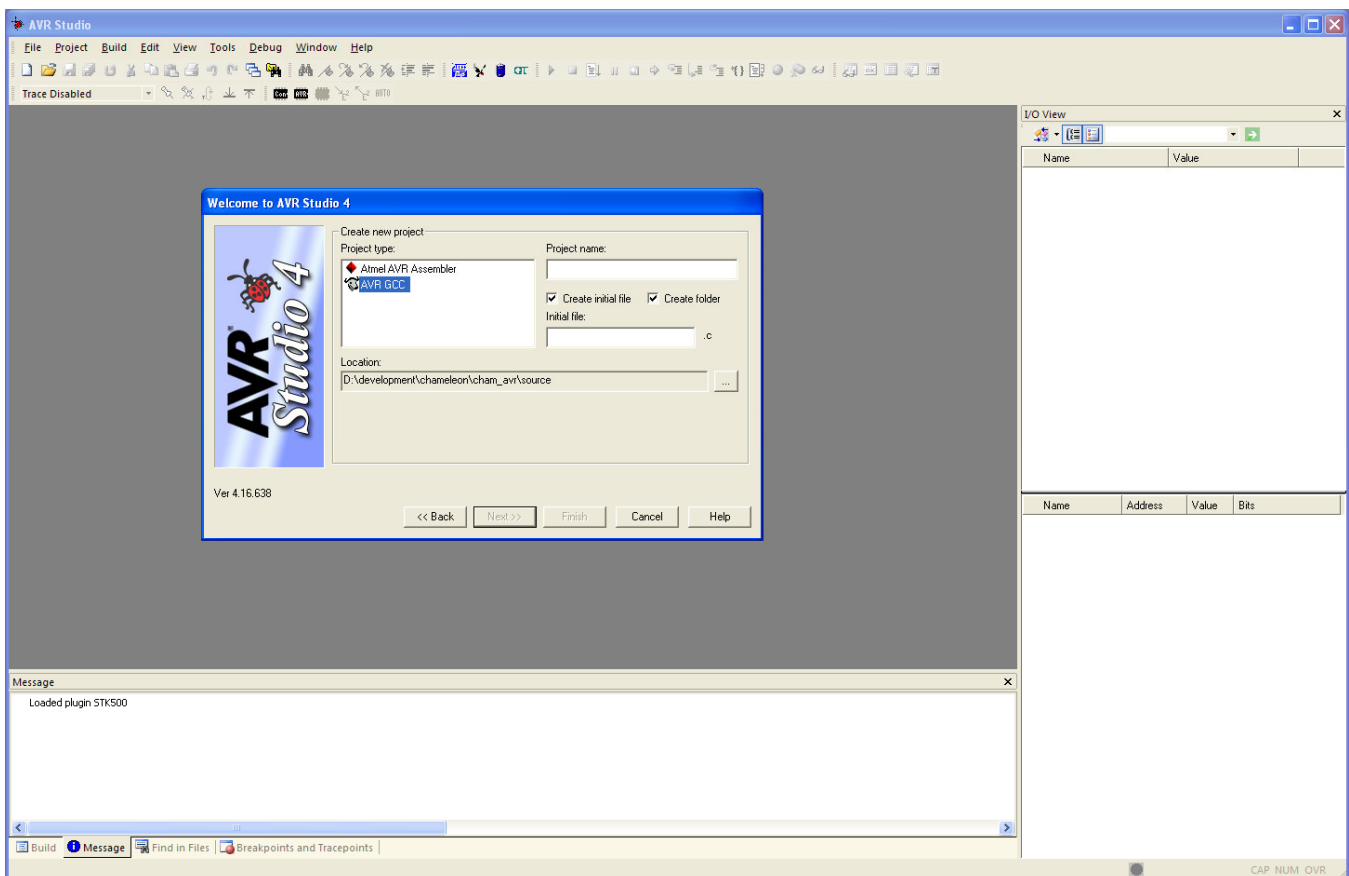
Congratulations, you have installed the tool chain and if you are still with me then everything must have went correctly! At this point, we have to set up the tools themselves which includes a lot of detail. I highly recommended you first skim over this section then return back to this point. This way you have an idea of what's coming. If you make mistakes during the setup process they are easy to fix if you know you made them, but very hard if you don't. Thus, take a few moments to skim the screen shots then return here and we set things up for real...

Alright, we are going to approach all development with the Chameleon AVR is as simple as possible. To that end, we need to create a **single** project to work with then for each demo we will simply include the **source files** in the source tree that the demo needs and remove ones it doesn't. This way we don't have dozens of projects, hundreds of directories and a big mess. The primary steps that we must follow are:

- Create an initial project and directory within the AVR installation directory.
- Set up the GCC tool chain properly; the compiler, linker, etc.
- Set up the AVR ISP MKII programmer itself (if you have one).
- Copy all the source files from the DVD with all the demos and drivers into the **Source** directory locally on your hard drive.

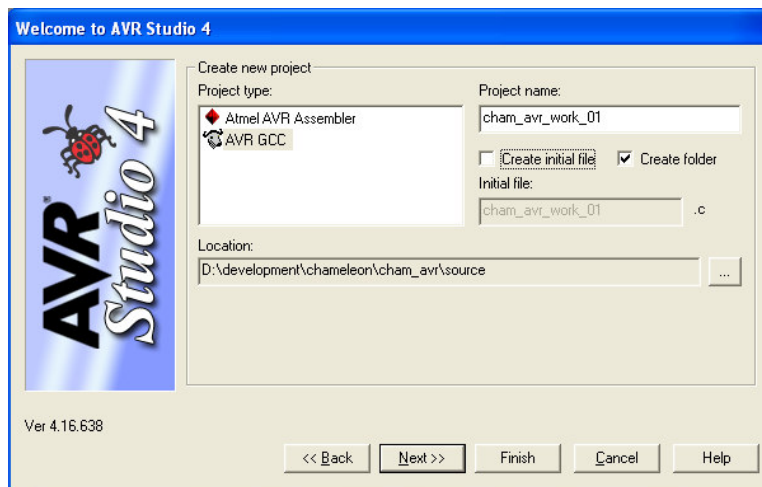
Let's begin by launching **AVR Studio 4**. You should have an icon on your **Desktop** or link in the **Start Menu**, locate AVR Studio 4 and launch it. You should see something like the image depicted in Figure 15.18.

Figure 15.18 – AVR Studio starting up for the first time.



Hopefully, you see something similar to this. However, it's possible that some of the tools in different locations or not at all (eg. the I/O view and Processor state). Any of these differences can be remedied at a later time from the **Main Menu** under **<View→Toolbars>**. Now, let's get to work.

The first step to setting up the tool is to create an initial project. The **New Project** dialog is shown in Figure 15.19 below.

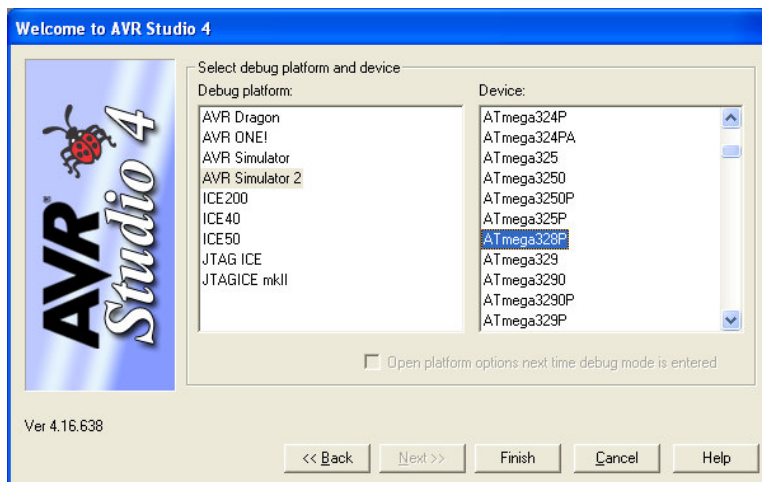
Figure 15.19 – Setting up the New Project dialog correctly.

Make the following changes to the default dialog, so we have the same project:

- Select AVR GCC – This tells the tool that we want to use C/C++ and GCC ASM.
- Check “**Create Folder**” and uncheck “**Create Initial File**”.
- Type in “**cham_avr_work_01**” under “**Project Name**”.
- Under “Location” browse to your install directory’s “**source**” sub-directory (or wherever you want to put projects), we will put all projects in there.

After you make these changes, you should have something that looks like Figure 15.19. Also, it’s important you use the project name “**cham_avr_work_01**” since I will refer to that from time to time and this will be the directory we will copy data to. However, in the future, you can create as many projects as you wish with whatever names you wish. Just make sure to place them all in their **own** directories, so you don’t have file collisions.

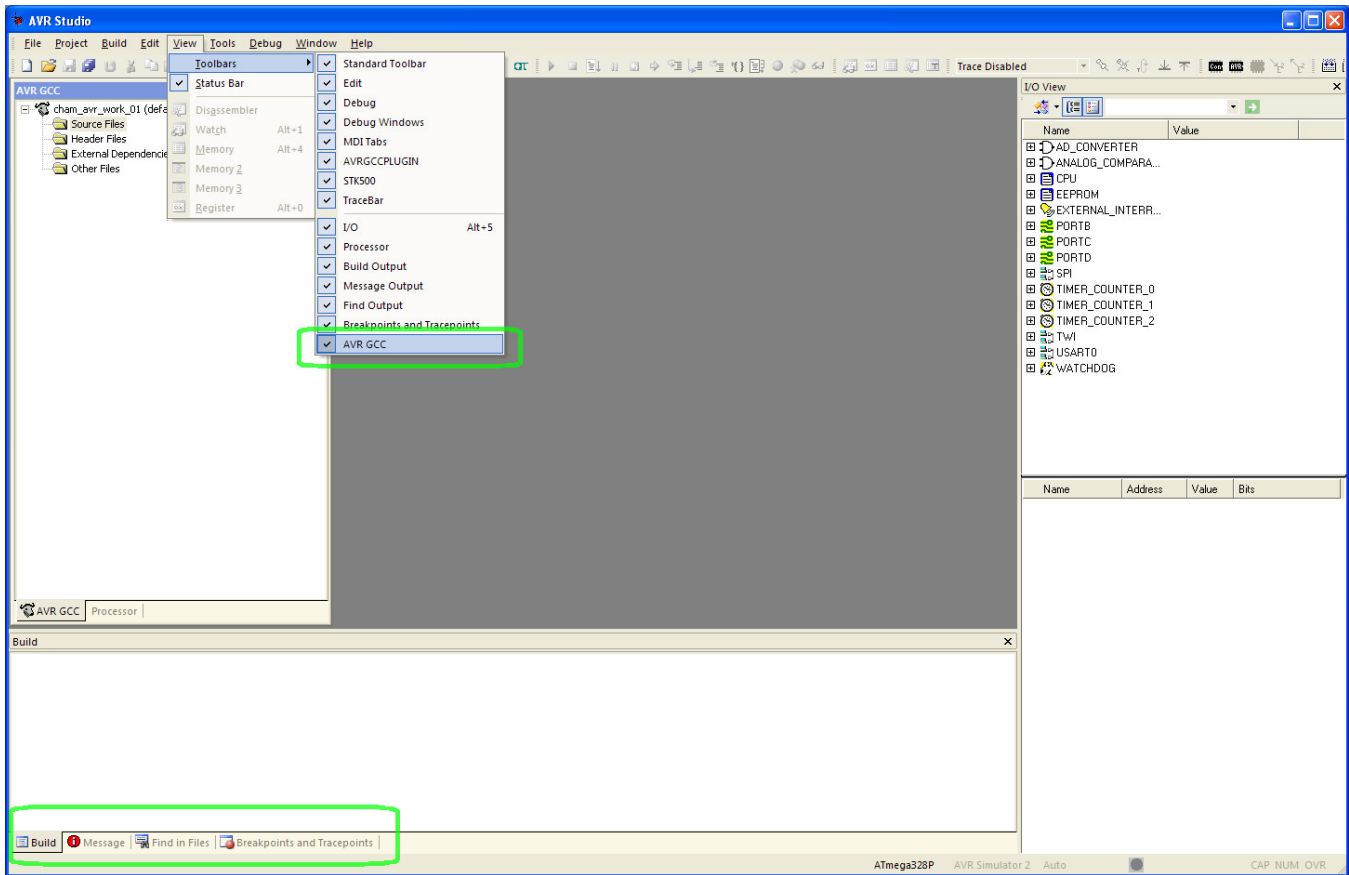
Once complete, go ahead and click **<Next>** to move onto the section of the “**Debug Platform**” as shown in Figure 15.20.

Figure 15.20 – AVR Studio selection of the Debug Platform.

AVR Studio supports all kinds of programmers, debuggers; **hardware** and **software** tools. This dialog helps you select the programming device along with the target device you will be working with. In our case, under **Debug Platform** select “**AVR Simulator 2**” along with “**ATmega328P**” under **Device**. The software simulator is of some use, but you will find its capabilities limited. Moreover, it only simulates the processor not any the hardware connected to it. Thus, in the future, you might want to purchase the **AVR JTAG ICE** debugger or other similar 3rd party tool.

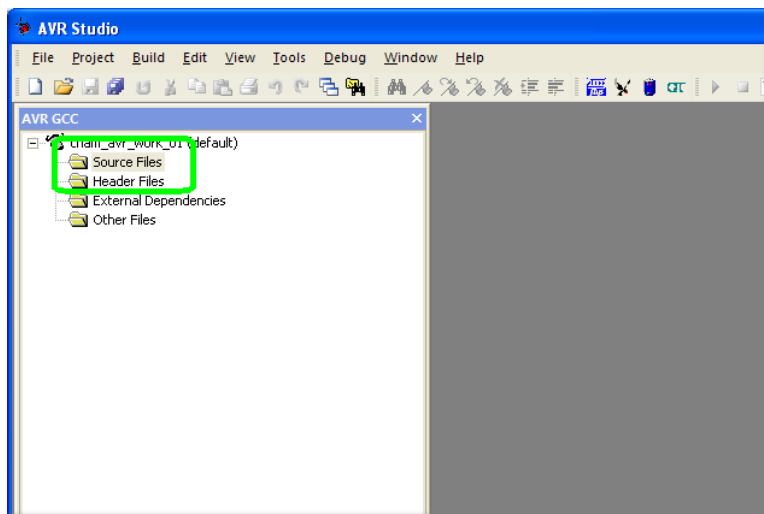
Moving on, click **<Finish>** once you have the **Debug Platform** filled out appropriately. This should open AVR Studio up to its default view which we might need to make adjustments to, so the correct tools are displayed. Take a look at Figure 15.21 below to accomplish this.

Figure 15.21 – Selecting the active tools in AVR Studio.



Referring to Figure 15.21 and selecting **<View→Toolbars>** from the **Main Menu**, you can select/de-select various tools that you wish enabled in the IDE. More or less, enable all of them especially **"AVR GCC"** shown last in the list, we need this tool, so we have access to the source tree of our project. You might want to do without the **"Processor"** and **"I/O"** displays since they really only matter for simulation and debugging support, but it's up to you. When you select the AVR GCC tool, you should see the file list control on the left side of the IDE as shown in the figure, this is important. The reason of course, this file list is where we will add/remove files for the current project in the **"Source Files"** folder as shown in Figure 15.22 below.

Figure 15.22 – The AVR Studio GCC Source Files folder.



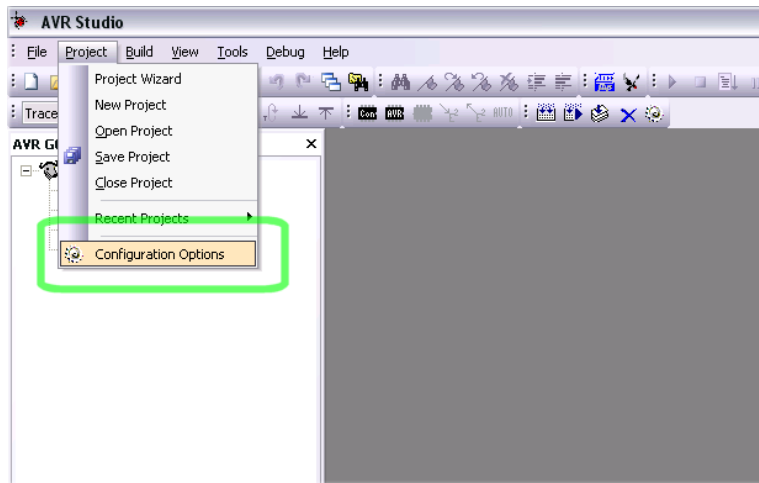
WARNING!

You will notice that I have also highlighted the **tabs** at the bottom of the tool interface's primary window in Figure 15.21. It consists of the following tabs; **Build**, **Message**, **Find in Files**, and **Breakpoints and Tracepoints**. This GUI control has a bug in it that you must watch out for. Sometimes during compilation, it will switch from **Build** view to **Message** view, this is very confusing since **YOU**, the **USER** did **not** request it. So you are minding your business, compiling an application and all of a sudden you look down at the output window and get the most bizarre errors? But, what you don't realize is that the window tab has switched from **Build** to **Message**. So the point is, watch out for this. And make sure when you are compiling and you get warnings, errors, etc. that the tab is set to **Build** view, so you can see the real problems and not some cryptic GCC tool error.

15.1.4.1 Setting up the Project Options

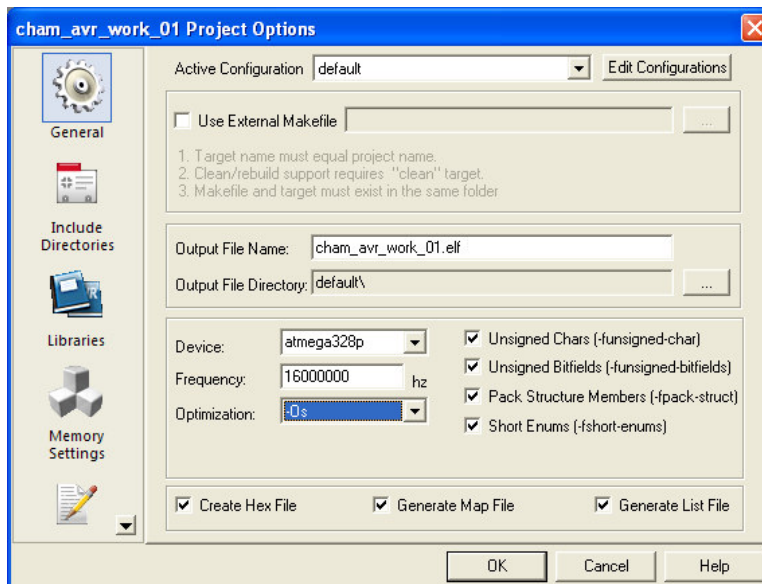
We will get to adding files to the project in a moment, right now, we need to perform more setup to get the tool chain where we need it. The next step is to launch the "**Configuration Options**" dialog from the **Main Menu** via the **<Project→Configuration Options>** as shown in Figure 15.23 below.

Figure 15.23 – Locating the Configuration Options dialog.



Once you launch the tool then you should see the very complex interface shown in Figure 15.24 below.

Figure 15.24 – The Project Options tool which sets up the entire tool chain.



WARNING!

This is the **most** important tool of the entire application. It's here that everything is configured. If the settings are not correct here you will have nothing, but trouble. Alas, in the next few paragraphs and screen shots we are going to painstakingly setup every single little detail. The number one problem with using tools like this is the setup, so take your time, it will pay you back in hours or days of frustration!

Referring to Figure 15.24, there is a lot going on with this tool interface. First, there are a number of categories on the left that can be selected such as;

General	- General settings across the compiler.
Include Directories	- Sets various search directories for the compiler.
Libraries	- Add/remove libraries that the linker might need.
Memory Settings	- Manipulate the initial memory configurations of the AVR.
Custom Options	- Set flags and controls for the tools like GCC and MAKE.

We aren't going to need to make changes to all the tool settings, but nonetheless, I highly recommended that you view all of them and read about them in the documentation and help since these settings control everything. In any event, let's begin with the **General** settings as shown in Figure 15.24. Make sure the **General** tab is set on the left hand side and highlighted.

First off, at the top of the tool under "**Active Configuration**" we are altering the "**default**" configuration, so make sure that's selected properly. Next, make sure "**Use External Makefile**" is **un-checked**. For those that aren't familiar with command line tools, **Make** is a tool that reads in a "**Makefile**" containing commands for everything from the compiler to the linker to the debugger. Make is typically used by Unix/Linux programmers and of course is part of the GCC tools we are using. Thus, you can use an external "**makefile**" if you wish. Luckily, WinAVR takes care of this for us and generates the make file, so we don't have to.

Next, the under "**Output File Name:**" should be the name of the project itself with the extension "**.elf**". Elf is the final output of the compiler and linker which is ready for conversion to the binary image downloaded into the AVR. Elf stands for "**Executable and Linkable Format**" and is a very common file format for executable files, binary objects, and other objects that are ready to be loaded or executed on various platforms. The WinAVR uses it as an intermediate format before the final binary AVR hex image is generated that we will download into the Chameleon each time.

Under the "**Output File Directory:**", you can set anything you wish, but for now set it to "**default**" as shown in the figure. This is where the final output will go and the binaries will reside with the data ready for the programmer itself to download to the Chameleon AVR target.

The next settings panel has a lot of settings that need to be addressed, let's talk briefly about each:

Device	- Sets the actual target device, in our case set to " atmega328p ".
Frequency	- Sets a constant for the compiler equal to F_CPU , set to main frequency of 16,000,000 Hz.
Optimization	- Sets the level of optimization, for our use, " -Os, -O1, or -O2 " are a safe settings.

You can experiment later with various optimization settings, but the **Device** and **Frequency** should be equal to the values above.

To the right of these settings are number of checkboxes, these control some of the GCC compiler's options which you can read about in the GCC documentation. However, they are pretty straightforward and similar to every other compiler's settings. They are only a subset of the dozens of compiler settings that you might want to set with GCC, but are some of the most common embedded programmers like to play with. In our case, the only one we want checked is "**Pack Structure Members**". All this does is pack structures tight rather than pad them to put them on **WORD** or other various boundaries. In essence, saves memory.

Finally, there are (3) checkboxes at the bottom of the page; "**Create Hex File**", "**Create Map File**", "**Generate List File**". These are important and should be checked. You should be familiar with the map and list file if you're a C/C++ programmer, but for those that don't here are some basic definitions:

Map File	- A "Map" file generated by a compiler typically lists the memory organization, usage, and symbol allocation as well as bindings. Very useful.
-----------------	--

List File - This is the most useful output from a compiler; it contains the actual ASM code generated by the compiler along with the C/C++ code along side. This way you can "see" how the compiler works and what it's doing in various situations. This is one of the best ways to learn optimization. I am always shocked at how many programmers know nothing about this tool! So don't be another statistic!

Hex File - This is embedded system specific and contains the final AVR compatible binary hex file that the programmer tool needs to FLASH the AVR's memory. In 99% of the cases we want to generate this file, so we can program the AVR with our code.

Since these files are so important and understanding what's in them is equally important, let's take a quick peek at each. I explored my latest compile, and extracted some bits of each file, so you can see what's in them.

Map File Example (.map)

The .MAP file is typically rather large and has all kinds of interesting information. By reading it, you will see all the details and get a real respect for what goes on behind the scenes of a linker/loader. Here's a small excerpt of one section of a .MAP file:

```
Allocating common symbols
Common symbol      size      file
g_uart_buffer_tx   0x20      CHAM_AVR_UART_DRV_V010.o
g_uart_buffer_rx   0x20      CHAM_AVR_UART_DRV_V010.o

Memory Configuration

Name      Origin      Length      Attributes
text      0x00000000    0x00020000    xr
data      0x00800060    0x0000ffa0    rw !x
eeprom    0x00810000    0x00010000    rw !x
fuse      0x00820000    0x00000400    rw !x
lock      0x00830000    0x00000400    rw !x
signature 0x00840000    0x00000400    rw !x
*default* 0x00000000    0xffffffff

Linker script and memory map

Address of section .data set to 0x800100
LOAD d:/development/xgs2/avr/winavr/bin/./lib/gcc/avr/4.3.2/./.././../avr/lib/avr5/crtm328p.o
LOAD CHAM_AVR_VGA_DRV_V010.o
LOAD CHAM_AVR_FLASH_DRV_V010.o
LOAD CHAM_AVR_GFX_DRV_V010.o
LOAD CHAM_AVR_KEYBOARD_DRV_V010.o
LOAD CHAM_AVR_MOUSE_DRV_V010.o
LOAD CHAM_AVR_NTSC_DRV_V010.o
LOAD CHAM_AVR_PROP_PORT_DRV_V010.o
LOAD CHAM_AVR_SOUND_DRV_V010.o
LOAD CHAM_AVR_SYSTEM_V010.o
LOAD CHAM_AVR_TEST_PGM_VER1.o
LOAD CHAM_AVR_TWI_SPI_DRV_V010.o
LOAD CHAM_AVR_UART_DRV_V010.o
LOAD d:/development/xgs2/avr/winavr/bin/./lib/gcc/avr/4.3.2/./.././../avr/lib/avr5/libm.a
LOAD d:/development/xgs2/avr/winavr/bin/./lib/gcc/avr/4.3.2/avr5/libgcc.a
LOAD d:/development/xgs2/avr/winavr/bin/./lib/gcc/avr/4.3.2/./.././../avr/lib/avr5/libc.a
LOAD d:/development/xgs2/avr/winavr/bin/./lib/gcc/avr/4.3.2/avr5/libgcc.a
...
```

List File Example (.lss)

List files are my favorite since you can change settings on the compiler and then see how the code generation changes. Additionally, you can see just how bad or good a compiler is at compiling simple statements. In many cases, you will be shocked at what the compiler's code generate outputs for something as simple as "x++"! Below is a small sample of a .LSS file, specially the code generated by one of the graphics functions in the API **VGA_Color(...)**:

```
int VGA_Color(int col)
{
    b8:      ef 92      push    r14
```

```

ba:    ff 92      push    r15
bc:    0f 93      push    r16
be:    1f 93      push    r17
c0:    8c 01      movw    r16, r24
// sets the "color" of the character, means different things under different drivers
// send driver set color command, [$0C, col 0..7]
SPI_Prop_Send_Cmd( GFX_CMD_VGA_PRINTCHAR, 0x0C, 0x00);
c2:    88 e0      ldi     r24, 0x08      ; 8
c4:    90 e0      ldi     r25, 0x00      ; 0
c6:    6c e0      ldi     r22, 0x0C      ; 12
c8:    70 e0      ldi     r23, 0x00      ; 0
ca:    40 e0      ldi     r20, 0x00      ; 0
cc:    50 e0      ldi     r21, 0x00      ; 0
ce:    0e 94 3d 12 call    0x247a ; 0x247a <SPI_Prop_Send_Cmd>
    milliseconds can be achieved.
*/
void
_delay_loop_2(uint16_t __count)
{
    __asm__ volatile (
d2:    80 e4      ldi     r24, 0x40      ; 64
d4:    e8 2e      mov     r14, r24
d6:    86 e0      ldi     r24, 0x06      ; 6
d8:    f8 2e      mov     r15, r24
da:    c7 01      movw    r24, r14
dc:    01 97      sbiw    r24, 0x01      ; 1
de:    f1 f7      brne    .-4           ; 0xdc <VGA_Color+0x24>
_delay_us(SPI_PROP_DELAY_SHORT_US);
SPI_Prop_Send_Cmd( GFX_CMD_VGA_PRINTCHAR, col & 0x07, 0x00);
e0:    07 70      andi    r16, 0x07      ; 7
e2:    10 70      andi    r17, 0x00      ; 0
e4:    88 e0      ldi     r24, 0x08      ; 8
e6:    90 e0      ldi     r25, 0x00      ; 0
e8:    b8 01      movw    r22, r16
ea:    40 e0      ldi     r20, 0x00      ; 0
ec:    50 e0      ldi     r21, 0x00      ; 0
ee:    0e 94 3d 12 call    0x247a ; 0x247a <SPI_Prop_Send_Cmd>
f2:    c7 01      movw    r24, r14
f4:    01 97      sbiw    r24, 0x01      ; 1
f6:    f1 f7      brne    .-4           ; 0xf4 <VGA_Color+0x3c>
_delay_us(SPI_PROP_DELAY_SHORT_US);
// return success
return(1);
} // end VGA_Color
f8:    81 e0      ldi     r24, 0x01      ; 1
fa:    90 e0      ldi     r25, 0x00      ; 0
fc:    1f 91      pop     r17
fe:    0f 91      pop     r16
100:   ff 90      pop     r15
102:   ef 90      pop     r14
104:   08 95      ret

```

As you can see this particular function is compiled down to a set of functions calls, thus mostly pushing parameters on the stack then making the function call with the ASM instruction **"CALL"**.

Hex File Example (.hex)

Here are a few lines from a .hex file as seen in a hex compatible editor such as Visual C++, Notepad++, etc. As you can see the data is simply a sequence or stream of bytes that are used to program the FLASH. Encoded in the format are addresses, starting locations, and data, so the entire FLASH need not be programmed if the final .HEX file is only a few hundred bytes long:

```

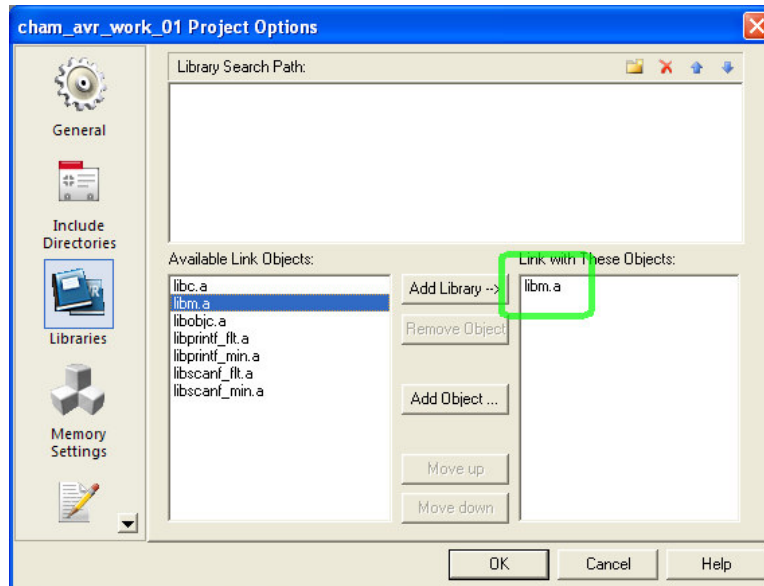
:10000000C9434000C945A000C945A000C945A002E
:10001000C945A000C945A000C945A000C945A00F8
:10002000C945A000C945A000C945A000C945A00E8
:10003000C945A000C945A000C945A000C945A00D8
:10004000C945A000C945A000C94A8130C94E513C9
:10005000C941E140C945A000C945A000C945A00E0
:10006000C945A000C945A0011241FBECFEFD8E014
:10007000DEBFCDBF14BE88E10FB6F8948093600058
:1000800010926000FBE16E0A0E0B1E0EEECF0E3ED
:1000900002C005900D92A43BB107D9F717E0A4EB7D
:1000A000B6E001C01D92A230B107E1F70E945107EE
:1000B000C9465180C940000EF92FF920F931F931D
:1000C0008C0188E090E06CE070E040E050E00E943D
:1000D0003D1280E4E82E86E0F82EC7010197F1F783
:1000E0000770107088E090E0B80140E050E00E9496
:1000F0003D12C7010197F1F781E090E01F910F9148
:10010000FF90EF9008958BE090E060E070E040E0B9

```

```
:1001100050E00E943D1280EA9FE00197F1F781E0F4
:1001200090E00895CF92DF92EF92FF920F931F938A
:100130007C016B0189E090E060E070E040E050E01D
...
```

Moving on, the next settings we are interested in are the “**Libraries**” tab of the settings dialog as shown in Figure 15.25.

Figure 15.25 – The “Libraries” tab of the Project Configuration control dialog.



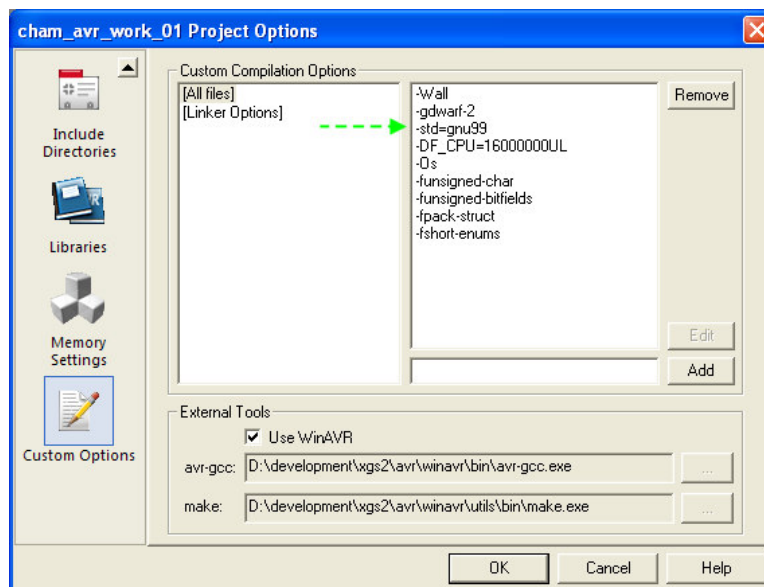
Shown in Figure 15.25 is the library tool. There are two main panes; one for the library search path and one that allows you to add/remove libraries. Due to the way we setup AVR Studio, all we need to do is add one library which is:

libm.a – Math library.

Later we might add more libraries, but for now, simply select this library on the left side of the “**Available Link Objects:**” and then click < **Add Library** → > to add it, once done, you should see it on the right side of the pane. Of course, you can also add single “objects” as well with this sub-dialog.

Next, up click the “**Custom Options**” tab to the left, this should bring up the dialog shown in Figure 15.26.

Figure 15.26 – The “Custom Options” tab of the Project Options dialog.



Referring to Figure 15.26, the **Custom Options** tab controls the compiler/linker settings for all files, or specific files. In our case, we need to make a change to make sure the programs compile correctly. On the left pane, select **[All Files]**, on the right you should see a list of options/flags that are applied to all source files by the compiler. You may recognize them from the **"General"** tab. In any event, this is where you can add/delete very specific flags. Right now, we need to delete something; notice the flag named:

-std=gnu99

This flag indicates certain compiler standards that we do *not* want to apply, thus we need to delete the flag. Simply highlight it and then click **<Remove>** in the upper right hand side of the dialog. Finally, make sure in the **"External Tools"** section at the bottom of the dialog the **"Use WinAVR"** checkbox is selected, if you wanted to you could actually change the path to the compiler and make tools here to other versions, but that's for advanced users only.

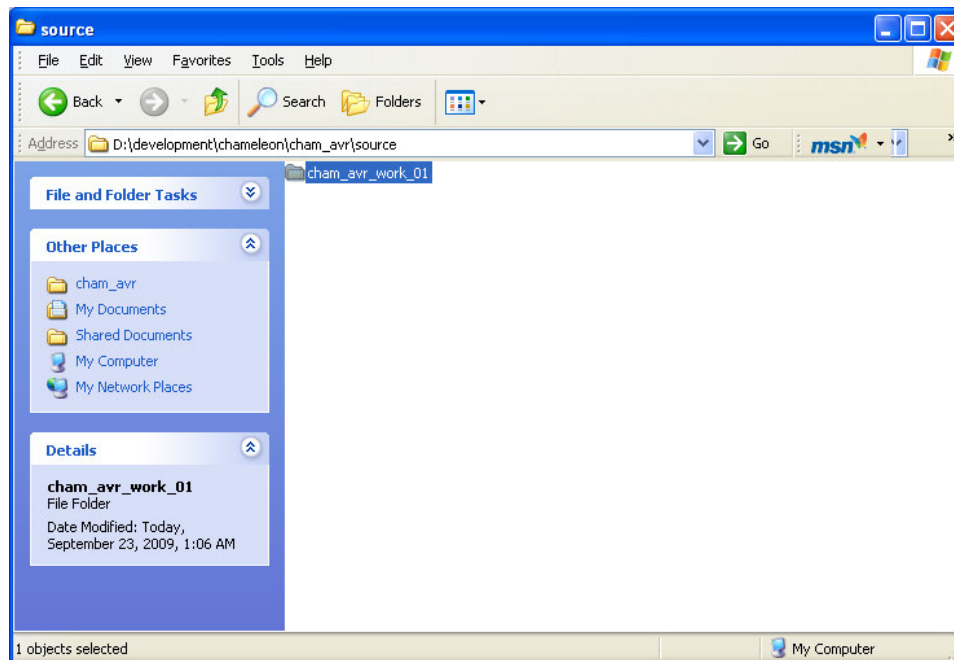
This completes the **Project Options** dialog settings, click **<Ok>** and let's move on.

15.1.4.2 Adding Files to the Project

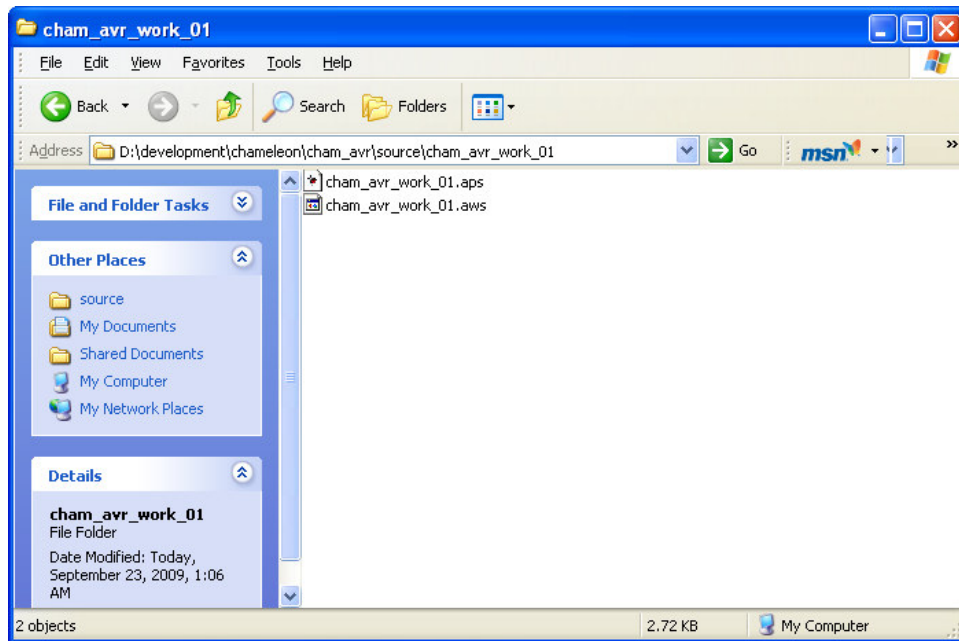
Now, we are ready to actually add files to our project, but first we need some files to add! Before, we get into that, let's briefly recap what we have been doing since at this point you might have forgotten what all these steps were for. So, we created a new GCC project, but before getting started we had to configure a number of components in the tool chain before we could do anything. At this point, AVR Studio is ready to go, and we can actually add the source files we want to compile and hopefully the compiler and linker will build our application perfectly.

The first step is to copy the source files from the DVD-ROM to the installation directory on your hard drive. To do this, you need to locate the **\Source** directory inside your AVR installation, it should look something like the listing shown in Figure 15.27.

Figure 15.27 – The "Source" directory in the AVR Studio installation or where you want your projects.



If you navigate into the **"\Source"** directory, this is where the project we created should reside in a directory called **"cham_avr_work_01"**. Yours should look something like the listing shown in Figure 15.28 if you navigate into the project directory.

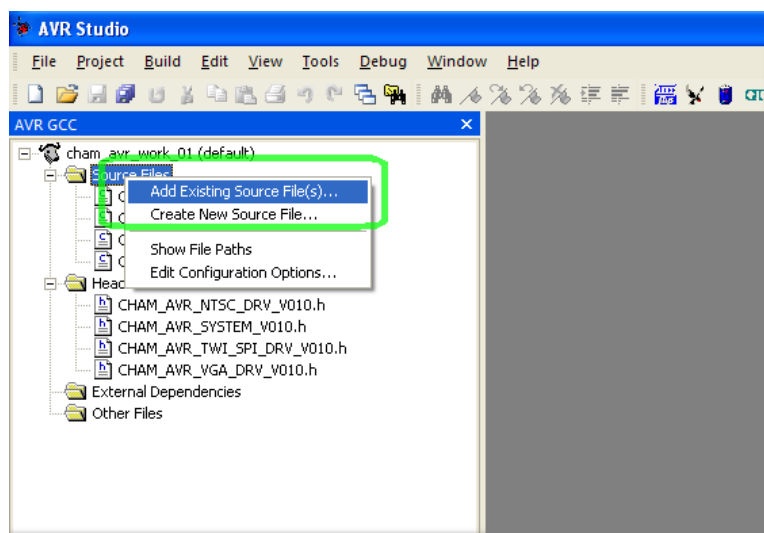
Figure 15.28 – A look inside the project directory.

Referring to the figure, the sub-directory “**cham_avr_work_01**” is highlighted, this is where you need to copy the source files from the DVD-ROM into. Additionally, note that there are some external project files at the end of the listing with the extensions **.APS**, and **.AWS**, these are the “**Project Files**” themselves and contain all pertinent settings, paths, and other information for AVR Studio to load them up. Thus, if you ever want to copy a project you have to copy these files as well as the project directory itself. Of course, just copying the files isn’t going to be good enough since there are hard coded paths in them, but they are there if you want to play with them. To copy the source files from the DVD-ROM, you simply need to drag or copy the contents of the directory **\SOURCE** located on the DVD-ROM here:

DVD-ROM:\ CHAM_AVR \ SOURCE \ *.*

Into the project directory on your hard drive “**cham_avr_work_01**”. This way we will have all the sources in a single directory which will make organization and finding things a lot easier. True the directory will have a lot of files in it, but that’s better than dozens of directories with copies of everything. Go ahead and copy the files now from the DVD.

Now, we are ready to add some files to our project and compile something. The first step is to gain access to the file adding mechanism. Like all IDEs there are a couple ways to do this, but the most straightforward is to highlight the “**Source Files**” folder under the project tree and right click the mouse to get a context menu. This is shown in Figure 15.29 below.

Figure 15.29 – Adding source files to the project.

Clicking the “**Add Existing Source File(s)**” selection will launch a standard Window’s file browser, navigate it into the project directory with all our recently copied files and then add the following files to the project:

Drivers

CHAM_AVR_SYSTEM_V010.C	- This is the main “system” file that is needed for all applications.
CHAM_AVR_TWI_SPI_DRV_V010.C	- This contains the SPI software API, needed for all applications.
CHAM_AVR_NTSC_DRV_V010.C	- This contains the NTSC API wrapper.
CHAM_AVR_VGA_DRV_V010.C	- This contains the VGA API wrapper.

Of course each of the above C files has an associated .H file. These must be added to the project as well in the headers section, or make sure the tool has the path to them, so after you add the C files to the project’s **Source Files** folder, then add the associated header files to the **Header Files** folder.

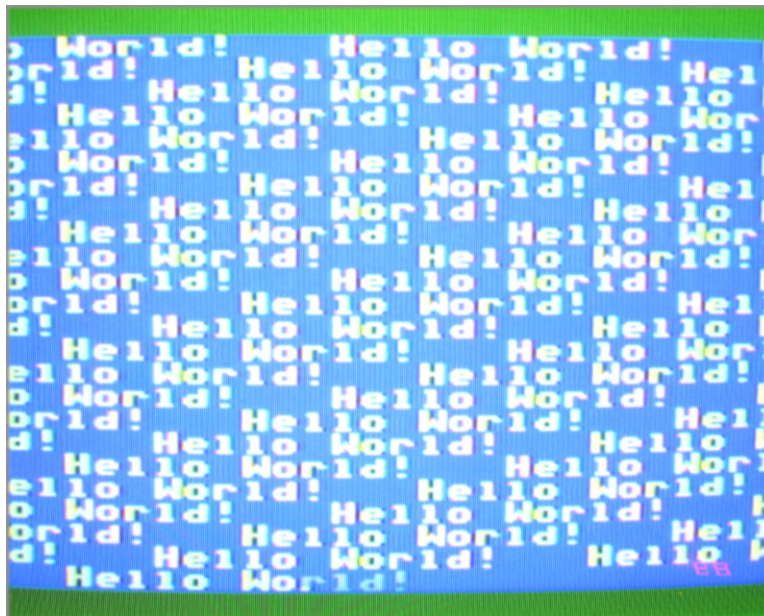
Main Program

CHAM_AVR_HELLO_WORLD_01.C	- The demo program that prints “Hello World!” to both the NTSC and VGA monitors. This of course does not have a header file associated with it, just the main C file.
----------------------------------	---

NOTE

The .C files are added the project explicitly, but the compiler still needs the .H header files, however, in the previous step you copied all of them into your working compiler directory, thus the compiler will be able to find them. However, I personally like to add them the **Header Files** folders just in case, and so I can get to them easily in my project window.

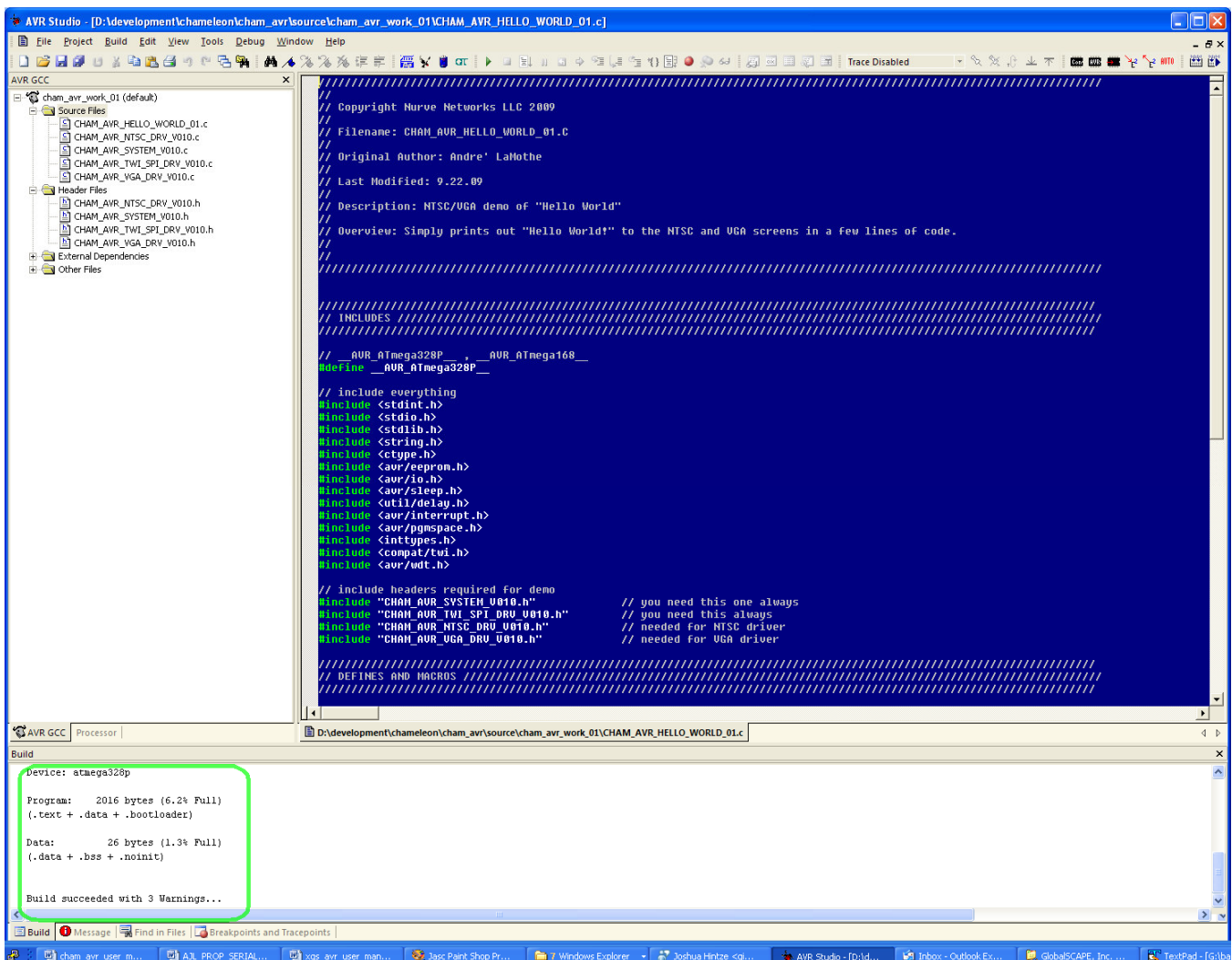
Figure 15.30 – A screen shot of the test running.



Alright, once you have all the files added to the source tree, it’s finally time to “**build**” the project! Referring to Figure 15.31, select **<Build → Rebuild All>** from the Main Menu. This will start the building process which consists of compilation, linking, and finally hex file generation for the AVR target.



Figure 15.32 – A successful build indicating data memory size, warnings, etc.



Go ahead and scroll thru the output window, so you can see all the compiler and linker output. You will notice quite a few warnings, these are fine and normal unless you want to program like a monk.

At this point, we are still not ready to program the Chameleon with AVR Studio! There is one more step – we have to setup yet another tool; the AVR ISP MKII (that is if you're not using the Arduino tool chain).

15.1.5 Setting up the AVR ISP MKII Hardware

Although we installed the driver and hardware for the AVR ISP programmer, we didn't configure the tool chain inside of AVR Studio yet to use it. This is what we need to do next. Of course, if you don't have the AVR ISPMK II programmer then you can skip this section. This might be the case if you are using AVR Studio for testing, simulation, or you simply have a different programmer that is compatible. But, if you have the AVR ISP MKII programmer then read on.

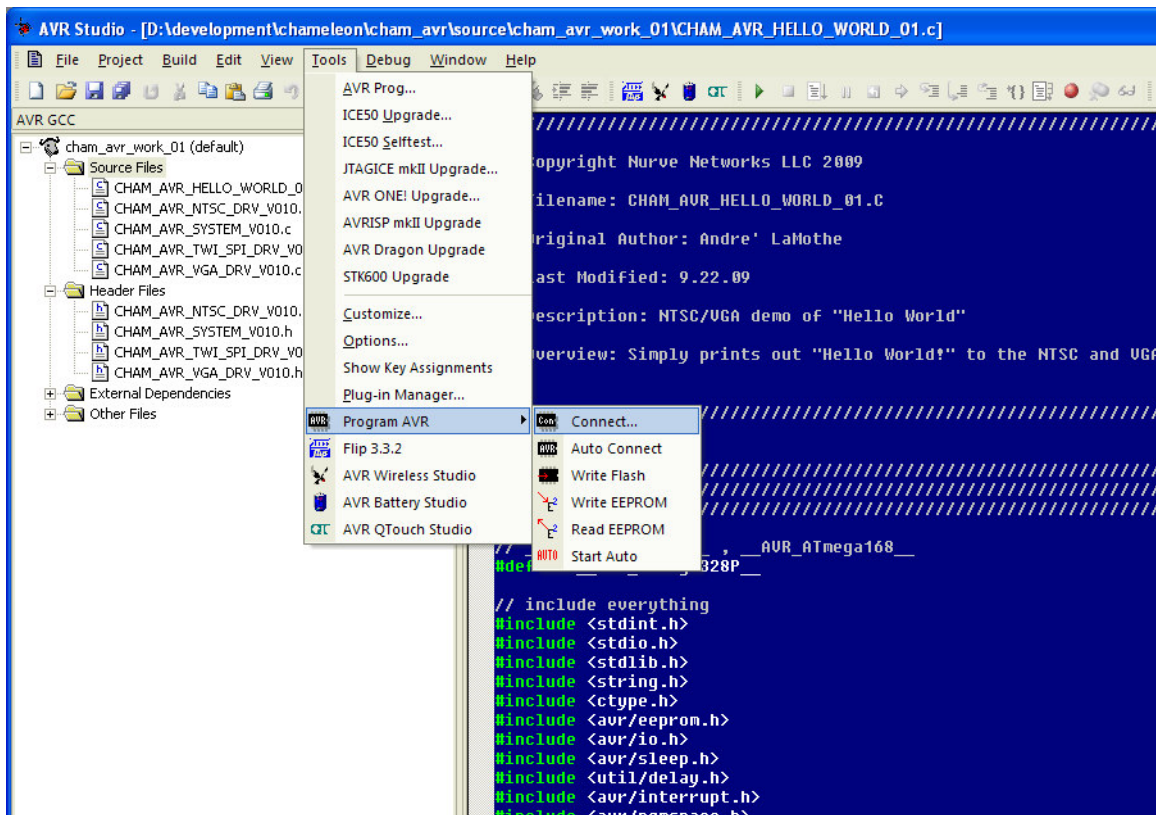
WARNING!

This process is actually **VERY** important since a wrong configuration in the programmer module can send the wrong data to the Chameleon AVR and ultimately to 328p chip. If we send the wrong bits to the chip we can potentially damage it or lock it up for eternity which would be very bad. Therefore, it's prudent to pay close attention to the programmer setup and know exactly what all the tabs and configuration settings do. We will spend a good amount of time on this.

First things first, we **have** built the project, thus if you navigate into the project directory "**cham_avr_work_01\default**" within the **Source** directory you will find a .HEX file with the name: "**cham_avr_work_01.hex**" within it. This is the final result of all our hard work, now we just need to get it into the Chameleon AVR, so let's see how to do that now.

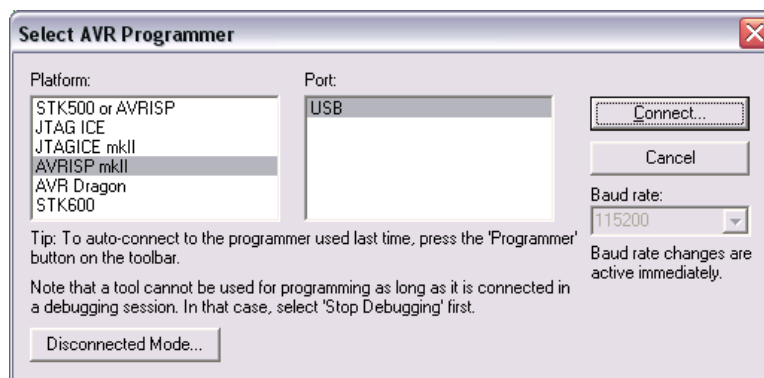
From the **Main Menu** select **<Tools → Program AVR → Connect>**, this is shown in Figure 15.33 below.

Figure 15.33 – Launching the programmer selection tool.



This should bring up the AVR programmer selection tool as shown in Figure 14.34 below.

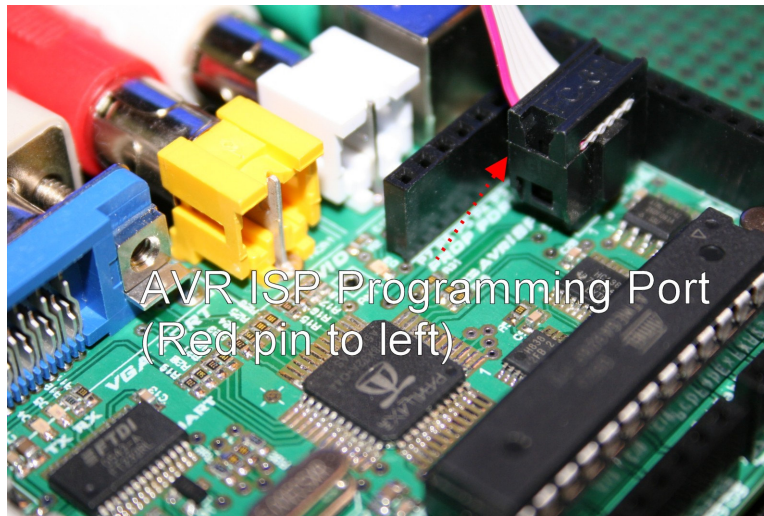
Figure 15.34 – The AVR programmer selection tool.



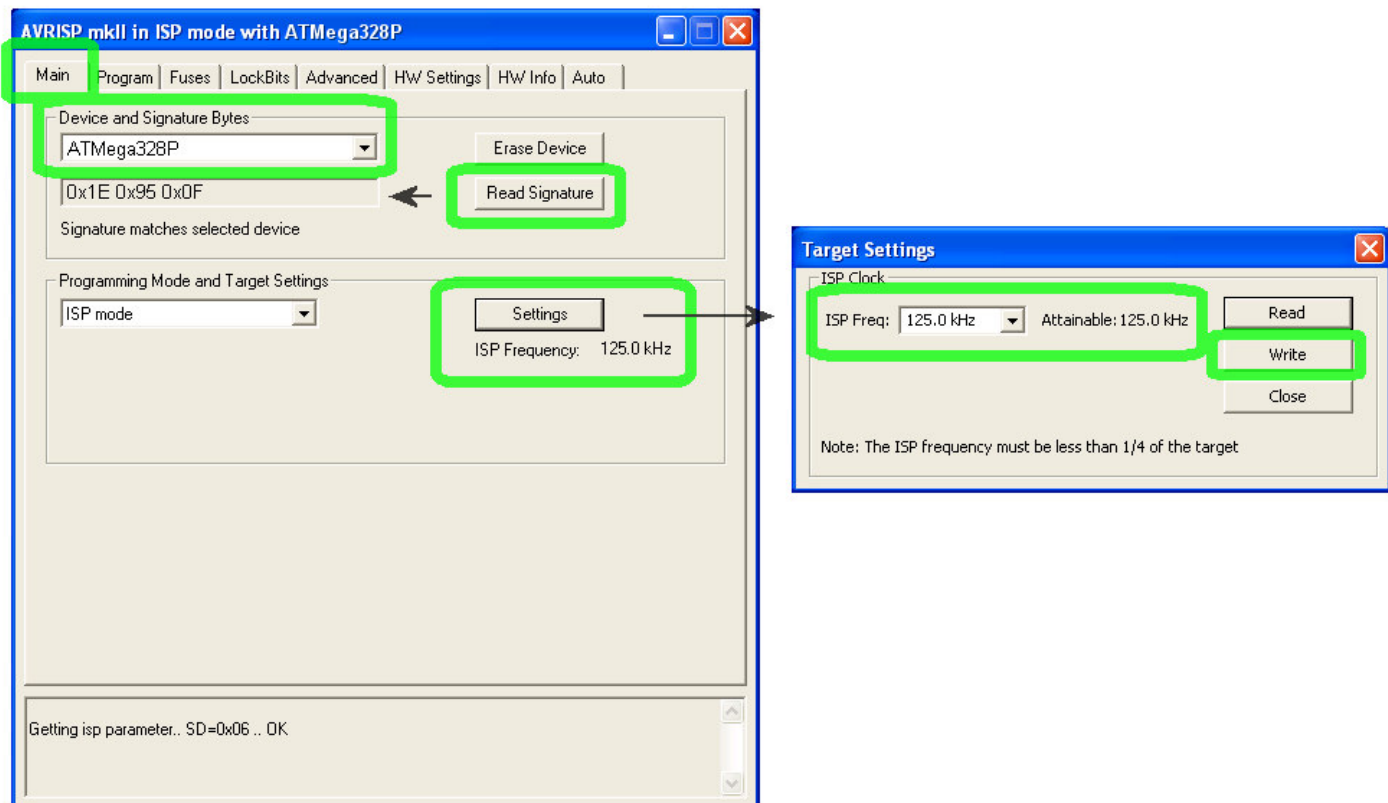
WARNING!

At this point, make sure the **AVR ISP MK II** is **plugged** into a **USB port** on the PC and the programmer itself is **plugged** into the **Chameleon AVR's ISP port** as shown in Figure 15.34. Pay attention to the pin out (the red line should be at the top). Finally, make sure the Chameleon's **power cable** is **plugged in** and the unit is powered up and **ON**.

This tool selects which "**Platform**" you want to use for your programming/debugging (left list box), and then in the right list box for the tool selected it lists the available communication devices, in this case only "**USB**" is listed, go ahead and select "**AVRISP mkII**" and "**USB**" as shown in the figure and then click connect. Make sure you have the AVR ISP USB cable connected to the PC and the 2x5 header cable from the AVR ISP programmer is plugged into the Chameleon AVR itself as shown in Figure 15.35.

Figure 15.35 – Plugging the AVR ISP MKII into the Chameleon AVR properly.

When you are ready to go, the Chameleon AVR is connected to the AVR ISP and powered up. Go ahead and plug in the 9V DC power supply or make sure the USB cable is plugged into your PC then turn the unit on by sliding the power switch to the left. Now, hold your breath and click **<Connect>**... If all goes well AVR Studio will start sending communication packets to the AVR ISP which in-turn will communicate with the ISP port on the Chameleon AVR and talk to the chip querying it. If a communications channel is successfully opened then the **AVR ISP MKII control panel** will open up as shown in Figure 15.36

Figure 15.36 – The AVR ISP MKII control panel's "Main" tab.

The control panel might launch starting with another tab, so go ahead and select the **"Main"** tab as shown in Figure 15.36, so you can follow the setup steps. This control panel sets up the AVR ISP programmer and can perform many

tasks. Additionally, this tool can actually damage the AVR chip if you don't set it up correctly. For example, you can force the wrong part, fuse bit settings, etc. and then download garbage to the chip locking it up, therefore, this tool **must** be set up correctly and care taken when using it.

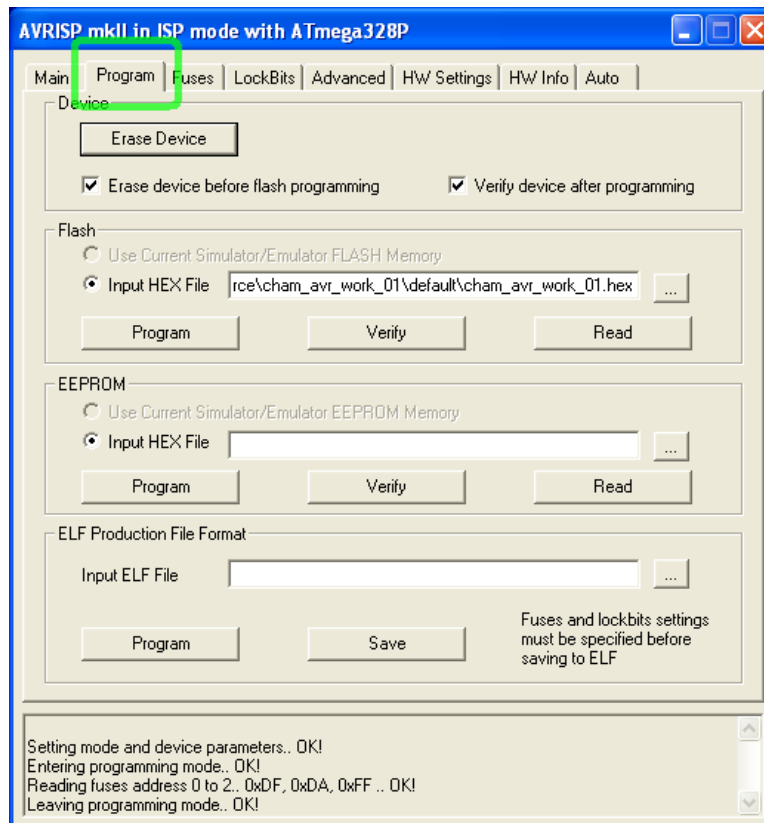
Let's begin with the "**Main**" tab as shown in the figure. This tab allows you to select the type of device you want to program, read the device signature (a unique id that indicates what the device is) as well as set the type of programming mode (ISP or JTAG) along with the frequency of programming. Let's talk about each selection in brief.

Device and Signature Bytes – These two controls select and indicate the device and respective "**signature**" read manually. In our case, we want to select the "**ATmega328P**" as shown in the figure, additionally, you can click the **<Read Signature>** button and the tool will read the device signature out of the Chameleon AVR and check it against the device you have selected in the above list box. For example, the AVR Mega329P should have the signature: "**0x1E 0x95 0x0F**".

Programming Mode and Target Settings – Depending on what programmer you have plugged in, it will have a number of communication modes; ISP mode, JTAG mode, etc. In our case, simply select ISP mode since the AVR ISP programmer uses the ISP interface on the AVR chip. Additionally, there is a speed setting that controls the download baud rate to the programmer. Set it for **125 KHz** as a nominal value, you can get away with higher speeds, but then risk some bit errors which can lead to locked up chips. Thus, **125 KHz** should be fast enough and the default setting. Simply press the **<Settings>** buttons to adjust the programmer baud rate.

When you have completed setting the "**Main**" tab up properly, select the "**Program**" tab next. You should see the controls shown in Figure 15.37 below.

Figure 15.37 – The AVR ISP MKII control panel's "Program" tab.



This tab is where you command the tool to program the chip's **FLASH** or **EEPROM** memories. In our case, we are only going to program the **FLASH** memory with the **.HEX** files generated by AVR Studio and AVR-GCC. Reviewing the "**Program**" tab at the top under "**Device**", make sure both check boxes are checked, we definitely want to erase the device each programming cycle and verify it after programming to make the download was error free. Additionally, you can click **<Erase Device>** if you wish to simply erase the AVR chip and do nothing more.

Moving on, the next area of interest is the **"Flash"** section of the tab. This is very important and what we will be using to program the AVR chip. Make sure the second radio button **"Input HEX File"** is selected, then you must navigate on your hard drive in your project directory to the .HEX file you wish programmed in each time. Now, this is an area that people make a lot of mistakes, so please read the warning below very carefully.

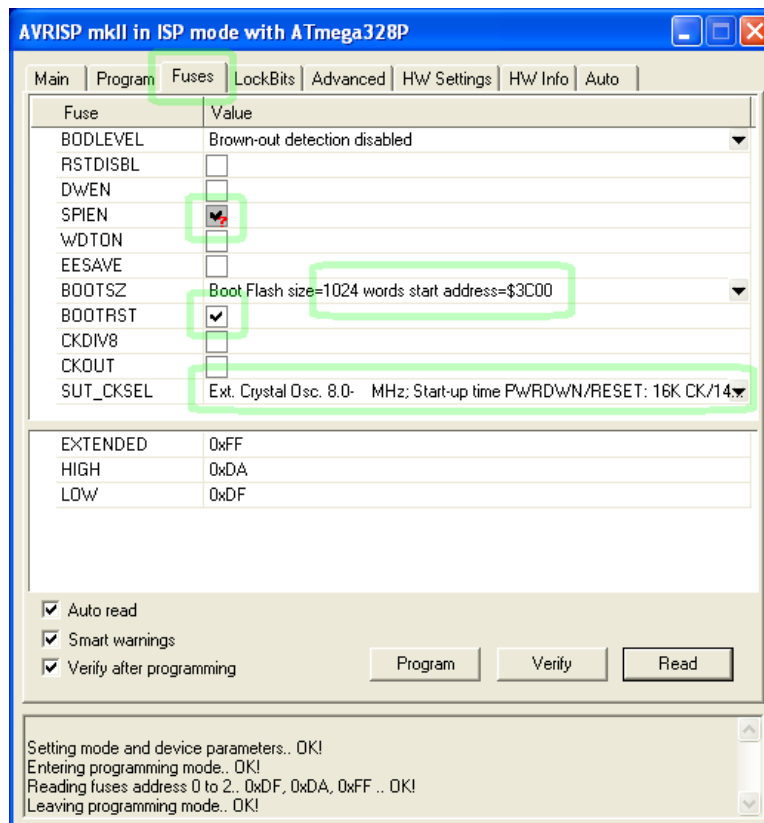
WARNING!

The AVR ISP tool is completely separate from AVR Studio. In other words, when you have a project opened in AVR Studio, it in no way communicates with the programming tool. Thus, you can open up 100 different projects, but the programming tool will **always** have the **same** settings which leads to confusion. This is a big problem since you might be working on **project_1**, set the programming tool a certain way, then program the chip, everything works fine. Then you work on **project_2**, set the programming tool another way, everything works fine. Now, you reload **project_1**, open up the programming tool, assume that it's settings were saved with **project_1**, but in fact they are whatever there were last which was **project_2** ! Thus, you have a problem. So the moral of the story is always check out the programming tool's settings, especially the **"Main"**, **"Program"**, **"Fuses"** and **"Lockbits"**.

Considering the warning, make sure that the input file is set correctly. To do this, navigate with the [...] button to the right of the control and locate your project directory for our work project named **"\cham_avr_work_01\default"** inside this directory you will find the .HEX file that AVR Studio and GCC generated, select the file and the tool will have the proper source data. Realize that every time you re-compile and build the project the .HEX file for the project is also updated, thus, the programming tool **always** points to the **latest** and most updated .HEX file.

At this point, we still are **not** ready to program the unit, we need to set the **"Fuses"** and **"Lockbits"** tabs properly. So, make sure your **"Program"** tab looks like the image shown in Figure 15.37 and then select the **"Fuses"** tab. You should see something like that shown in Figure 15.38.

Figure 15.38 – The AVR ISP MKII control panel's "Fuses" tab.



The **"Fuses"** tab controls a set of internal processor **"fuses"** that are no part of the FLASH or EEPROM memories. Rather, the fuses control the "personality" of the chip in a number of areas. Thus, we must set these fuses every time we program the processor, so that they are correct. Referring to Figure 15.38, you can see there are a lot of fuse settings and a lot of cryptic labels on the left hand side. First, simply mimic what you see in the figure and set your fuses to exactly these values. Be specially cognizant of the settings for:

BODLEVEL	- Brown out control (disabled).
JTAGEN	- JTAG Enable.
SPIEN	- SPI Enable.
BOOTSZ	- Boot FLASH size (size = 1024, address = \$3C00)
SUT_CKSEL	- Clock Select (8Mhz, Start up time 16CK/14CK + 0ms).

However, all of the fuse settings need to be exactly as depicted in the figure. Additionally, make sure the three checkboxes at the bottom of the window are all checked; **Auto read**, **Smart warning**, and **Verify after programming**.

When your fuse settings are all exactly as shown in the figure above, then the bottommost window will show you the actual binary encoding which should be:

Fuse Bank Name	Value
EXTENDED	0xFF
HIGH	0xDA
LOW	0xDF

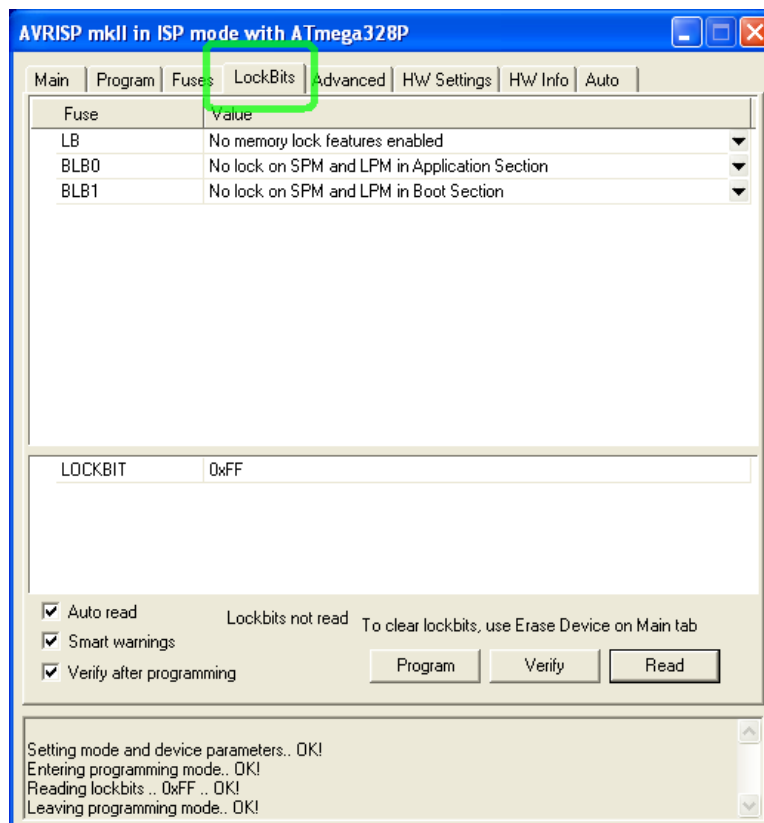
If you weren't using this tool and using another tool that needed the actual binary bit values for the "extended", "high", and "low" fuse bits then you could use these values.

Once you have all the fuse bits and drop downs set to mimic the settings in Figure 15.38, you could click the **<Program>** button to write the fuses if you wish. This will not program the chip (since we don't want to do that yet), but it will program the fuse bits only. Go ahead and click **<Program>** to write the fuse bits, if everything goes well, you should see the text output window at the bottom of the tool scroll some informational strings and you should see a lot of "OK!"s displayed. If you don't try it again.

Normally, you do not have to do this when programming the chip. The programming process will reset the fuse bits as well, but it can't hurt to make sure. If your chip ever acts strangely always go and investigate the fuse bit settings and see if they are correct (as shown in Figure 15.38).

We are almost there, the very last tab of interest is the "Lockbits" tab, select this tab as shown in Figure 15.39.

Figure 15.39 – The AVR ISP MKII control panel's "Lockbits" tab.



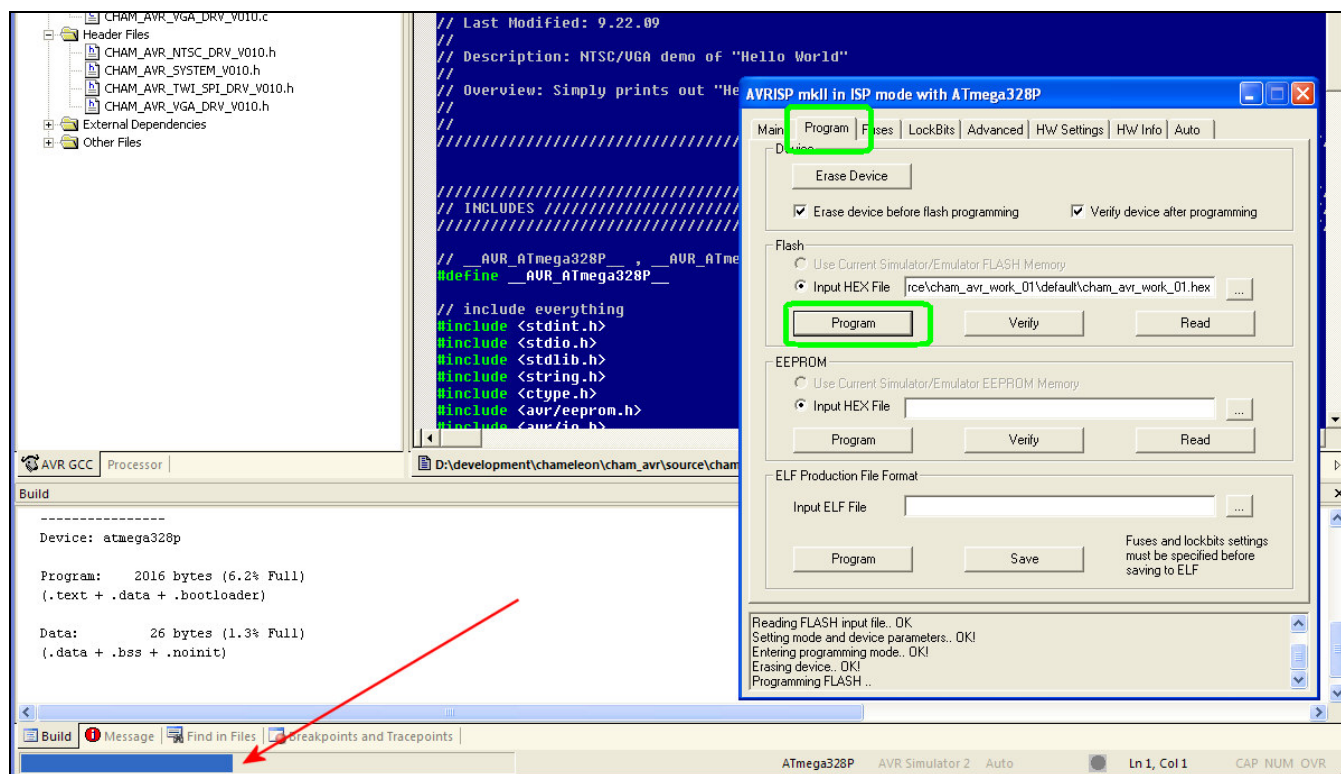
The **"Lockbits"** tab controls memory protection features of the AVR series processors. In your case, you aren't going to distribute the processor with some embedded application on it, so you want to **disable** all memory protection. Referring to the figure, make sure that the **LB**, **BLB0**, and **BLB1** are all set to **"no lock"** on each category as shown in the figure. Additionally, make sure that all the checkboxes at the bottom left of the tab are all checked. Finally, go ahead and click **<Program>** to write the lockbits. Again, this should initiate the writing process and you should list a number of status operations scroll down the bottom of the window all with **"OK!"** if everything was successful.

TIP

The word **"fuses"** is really a bit out of date. Technically, early microcontrollers had actual metal fuses inside of them that the user/program could **"blow"** with high current to enable/disable specific features. But, once blown the fuse couldn't be changed (also referred to as **OTP**, **"One Time Programmable"**). These days fuses can typically be programmed over and over and aren't fuses at all, but FLASH/EEPROM memory locations. However, the word **"fuses"** is used a convention for historical reasons more or less to this day.

At this point, the programming tool is all set up for the Chameleon AVR and our development needs. Finally, it's time to actually download the .HEX file we previously generated and is waiting in the **"cham_avr_work_01default"** folder on your hard drive. Re-select the **"Program"** tab in the tool, and under the **"FLASH"** section click **<Program>**. You should see a progress bar at the bottom of the IDE interface as highlighted in Figure 15.40 below.

Figure 15.40 – Programming the Chameleon AVR and the progress bar at the bottom of the IDE is proof.



If all works out, immediately after the download, the AVR ISP will reboot the Chameleon AVR and you should see on your NTSC/VGA displays the **"Hello World"** demo running as shown in Figure 15.30! Go ahead and press the **RESET** button on the Chameleon to restart the hardware.

At this point, you can edit the **"hello world"** program and make changes to it. If you do, I suggest you save-as another name, then remove the original from the source tree and add your modified version. This way you will always have the original on your hard drive. Of course, the original sources can always be re-copied from the DVD-ROM, but that's inconvenient.

15.1.7 Final Words on AVR Studio Tool Chain Installation

This completes our installation of the toolchain all its components and a complete example of building and downloading an application. As we develop applications, load demos, and experiment, you will only have to modify the source files in the project pane to the left adding and subtracting source files. The programming tool setup should be the same. Thus, the workflow moving forward will be to add/subtract files from the **"Source Files"** folder of the same project, then **"Rebuild All"** from the Main Menu's **"Build"** menu then using the AVR ISP programming tool control panel **"Program"** the Chameleon AVR with the new binary. That's it!

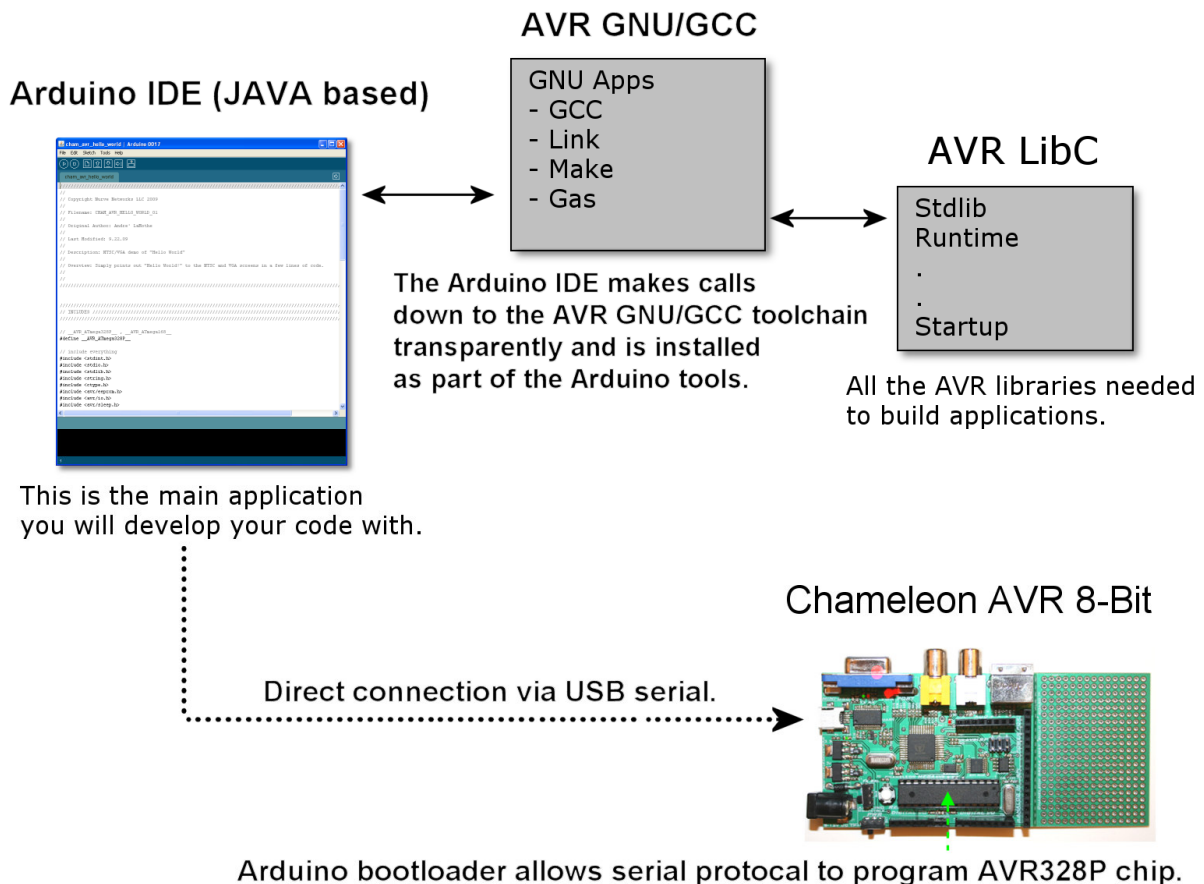
TIP

Once you get your AVR Studio and the AVRISP (or other programmer) talking to the Chameleon AVR then you can slowly increase the programming rate from 125 KHz. I have mine set for 1 MHz and works fine, but I suggest slow moves. If you get a glitch in communications and some fuse bits get set wrong you will be stuck and need a JTAG programmer potentially to unlock your chip. So after you get it working at 125 KHz, then try 250 KHz, 500 KHz, 1 MHz, and that's as high as I would go. 2 MHz will work, but its risky.

15.2 Arduino Toolchain Setup

The Chameleon AVR **ships** with the **Arduino bootloader** pre-loaded into the AVR's flash memory. This allows you to immediately use the Chameleon as an Arduino (more or less), but with added media abilities via the Propeller chip. Without the Arduino bootloader on the AVR chip, you would have to use the AVR ISP MKII programmer or something else to get code into the AVR chip. But, with the Arduino bootloader, the Arduino tool talks to the AVR chip over a serial port (via the USB serial port on the Chameleon) and you can compile and download programs. Figure 15.41 shows the relationship between the components of an Arduino like setup.

Figure 15.41 – The Arduino system setup with the Chameleon.



As you can see, the Arduino setup is very similar to the AVRStudio setup, except that the Arduino tool along with the bootloader allows us to program the AVR without the external programmer which is very convenient. The only downside is you must use the Arduino tool, IDE, and setup. This isn't too bad, but if you are a serious programmer, you will immediately feel limited by the over simplified Arduino programming tool and its limitations. But, if you are a newbie then you will probably really like the Arduino tool since it insulates you from the complexities of what's really going on.

Referring to Figure 15.41, here's what's going on. First, there is a JAVA application that acts as the IDE for the Arduino hardware (or compatible like the Chameleon). This JAVA application is nothing more than a GUI, it has very little functionality, just enough for a crude text edit and a couple menus. But, the reason its in JAVA is that it will work on Windows, Linux, and Mac OS X, that's why they decided to use JAVA. Additionally, the application has the exact same look and feel on each machine since it always uses the JAVA gui elements library which is nice.

Moving on, the Arduino editor creates projects in the form of "Sketches" these are nothing more than a primary C/C++ file that you would refer to as you "main" file. The file imports all your libraries via headers and a special directory structure that we will get to in a bit. In any event, you create a single sketch project then add #include statements to it which then invokes the inclusion of all the libraries. Now, when you compile your program what happens is the GNU GCC C/C++ compiler is called in the background from a command line. The GCC compiler is also installed with the Arduino tool, but you don't have to do **any** setup at all. On Windows machines, the GCC compiler actually runs in a Cygwin linux shell (this is a system that allows you to run Linux programs in Windows, you can learn about it below). Again, Cygwin is transparently installed and you don't have to worry about.

<http://www.cygwin.com/>

So, you edit your Sketch (program) in the Arduino editor, you compile it (with GCC under the hood) then finally you need to download it to the Chameleon. The Arduino tool calls another program AVRdude which does this and sends the binary image via the serial line with a special protocol agreed to between the bootloader and AVRdude. And that's how it all works!

TIP

The Arduino bootloader is really nothing more than the standard bootloader provided by Atmel for serial communications and loading of firmware without the AVR ISP MK II programmer or similar. Thus, you can actually use other tools like "AVR Dude" to download firmware written in AVRStudio. Moreover, there are other bootloaders for the AVR chips that you can flash into them that support other 3rd party tool. However, the problem with all this is that you need a tool to program the bootloader in the first place! Thus, if you frag a bootloader in FLASH memory and you don't have an actual programmer like the AVR ISP MK II you are out of luck. This is why we suggest spending \$30-40 and buying one just in case if you're going to do serious AVR development.

Now before we get into the Arduino tool installation, a couple things to know. The Arduino tool actually uses C/C++ to program in, its not some new language, its not BASIC, its straight up C/C++. There is a library for the Arduino called "**Processing**" which allows you to use Processing language like commands and functions. Bottom line, the Arduino is programmed in **C/C++** nothing more. The Arduino guys have simply created libraries, so the system feels like the **Processing** language, which you can read about here:

<http://processing.org/>

15.2.1 Installing the Arduino Toolchain in Windows

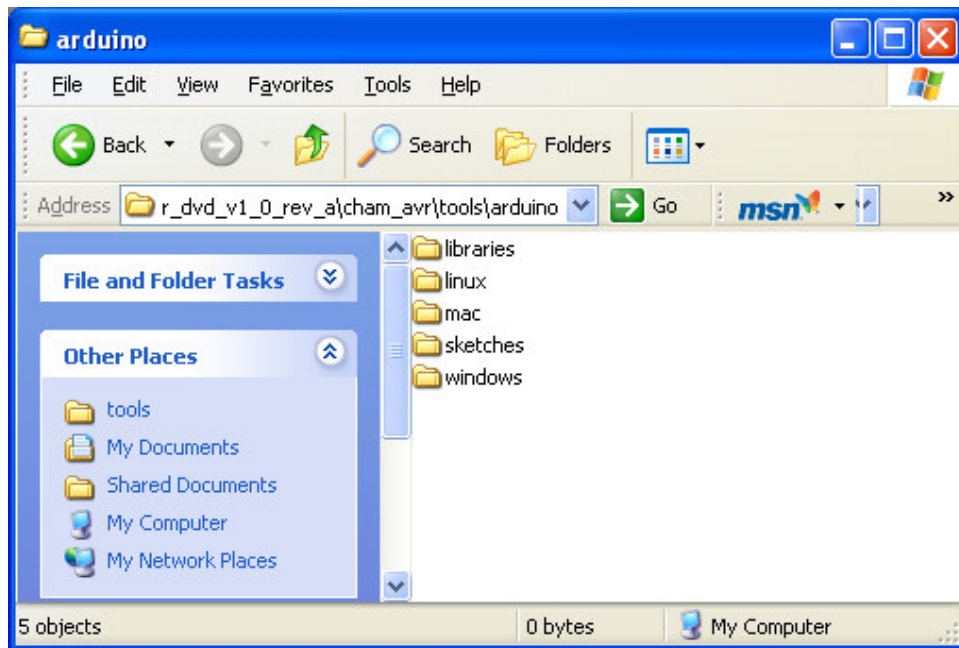
There isn't really an installer for the Arduino tool, you more or less just copy the files (unzip, untar, etc.) the Arduino files into a directory on your hard drive and that's it. With Windows, Linux, and Mac OS there are some slight differences, but more or less its all the same. We will concentrate on the Windows version here, the Linux and Mac installs are nearly identical and you can read about them on the Arduino site itself. However, the adjustments we make the installation directories are not on the Arduino site, thus read this section no matter what even if you are **not** installing on Windows.

The first step is to download the latest version of the Arduino development package for Windows. The latest files at the printing of the manual are located on the DVD here:

DVD-ROM:\ CHAM_AVR \ TOOLS \ ARDUINO \ WINDOWS \ arduino-0017.zip

There are also directories for Linux and Mac which have their respective files. But, before copying the above file to your desk, I want you to take a look at the directories on the DVD-ROM one level above. These are shown in Figure 15.42 below:

Figure 15.42 – Directory structure of the Arduino tools on the DVD-ROM.



Referring to Figure 15.42, you see the Windows, Linux, Mac directories which each contain a ZIP file (or appropriate) set of tools for the platform. But, there are two other directories that we are going to need in a moment; **\LIBRARIES** and **\SKETCHES**. These two directories are needed to augment the Arduino default installation, so we can compile Chameleon programs with the Arduino tool. The **\LIBRARIES** directory has all the Chameleon libraries ported over to work with the Arduino tool (very simple changes in most case) and the **\SKETCHES** directory has all the programs and demos from this manual in Arduino format for you.

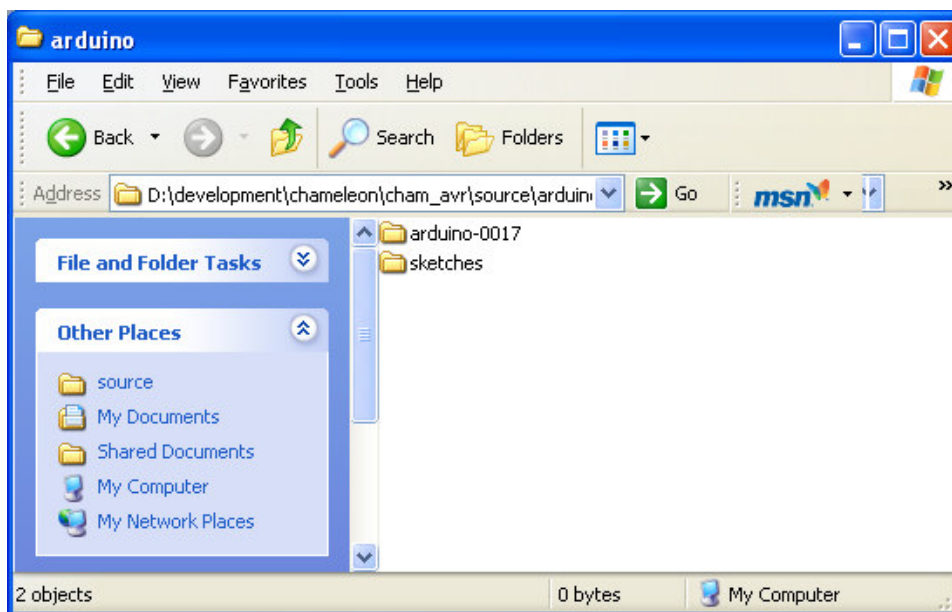
Later in the installation, I am going to direct you to copy the contents of these folders into their respective locations in the Arduino installation directory. This is important since you will perform this step on all three platforms; Windows, Linux, Mac -- whichever you are using.

15.2.2 Copying the Files to Your Hard Drive

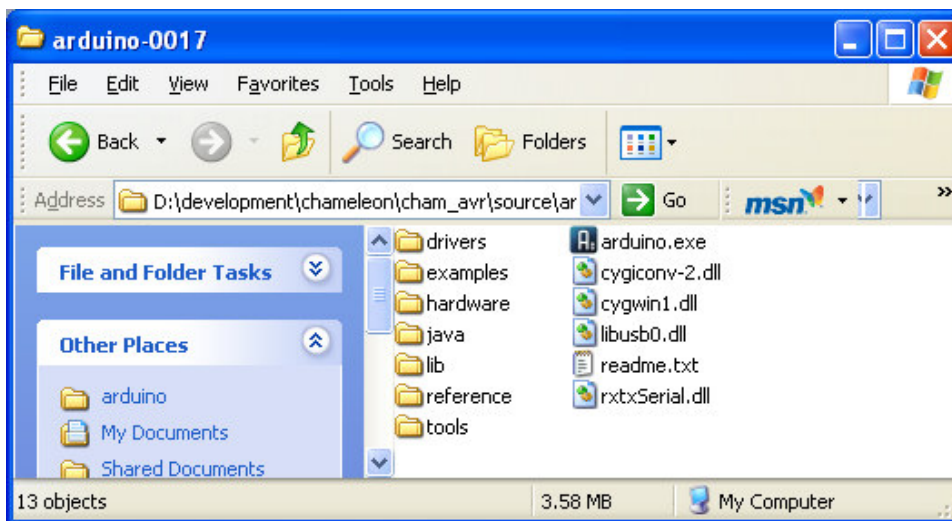
Go ahead and locate the main ZIP file on the DVD named:

DVD-ROM:\ CHAM_AVR \ TOOLS \ ARDUINO \ WINDOWS \ arduino-0017.zip

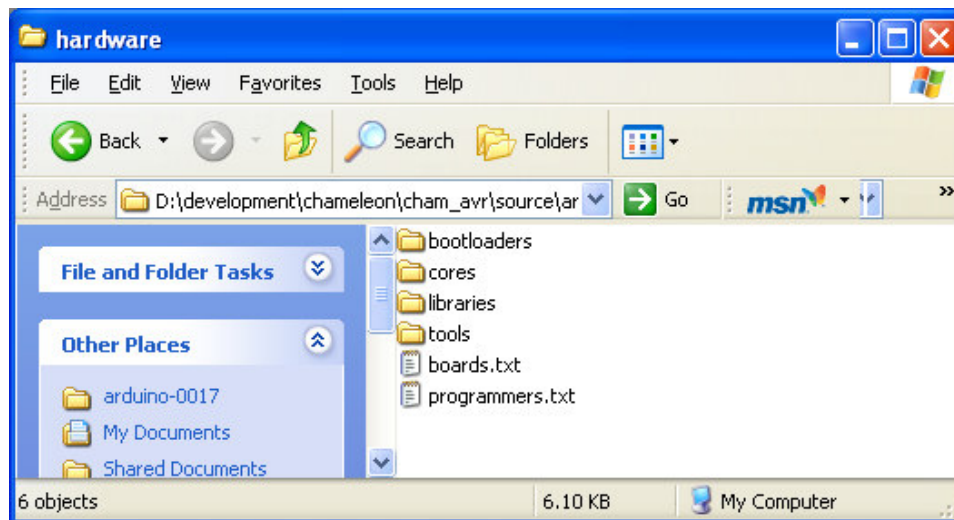
And decompress it into your development directory. I do all my work in a directory called **\SOURCE** on my computer (and this is where you should have copied all the .C\H files in previous installation steps. Go ahead and create a directory called **\ARDUINO** inside **\SOURCE**, then decompress the ZIP file right inside the **\ARDUINO** folder and enable directory names. This way the unzip will create a sub-directory named **\ARDUINO-0017** for you. Then in the future as you get new versions of the tool you can do the same and keep them organized if you want to roll back. Thus, you should have something that looks like the directory structure shown in Figure 15.43 when done.

Figure 15.43 – Setting up the main directory for the Arduino tool chain.

Referring to Figure 15.43, you will see there is a **SKETCHES** directory in there! Yup, this is where you are going to copy the **SKETCHES** from the DVD directory previously mentioned. So do that now, this is simply all the projects for the manual all pre-built and ready to go. Now, go ahead and dive into the **ARDUINO-0017** directory and take a look around, go into all the directories, as deep as you can, then when you are done exploring come back to the root which should look like the folder shown in Figure 15.44.

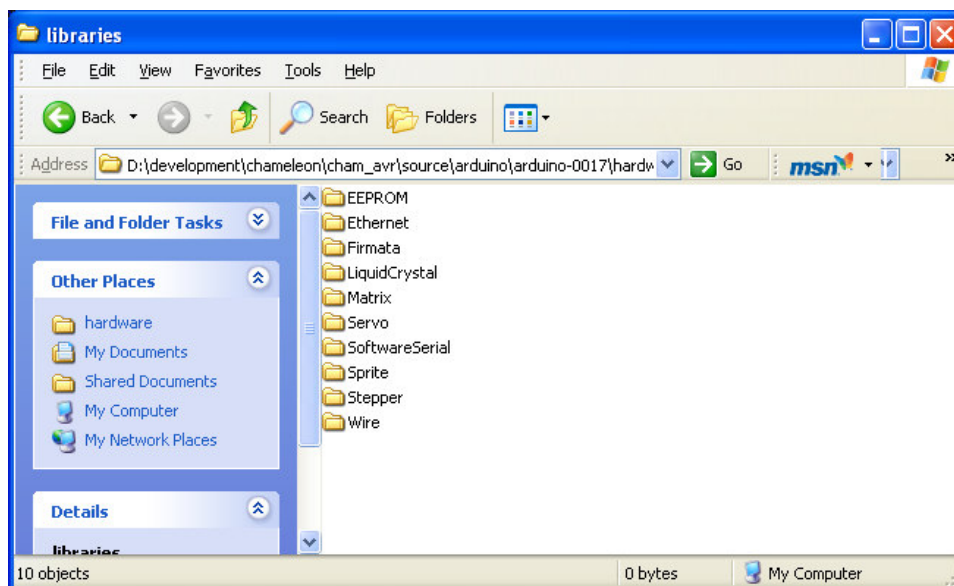
Figure 15.44 – The Arduino installation directory contents.

As you can see, there are a lot of files, hundreds if not a couple thousands with all the GNU stuff and Cygwin files buried in there. But, we luckily don't have to worry about much of it, we just want to update the **LIBRARIES** directory with the Chameleon packages, so we can build Chameleon compatible programs. To do this go ahead and dive down into the **HARDWARE** directory this is shown in Figure 15.45.

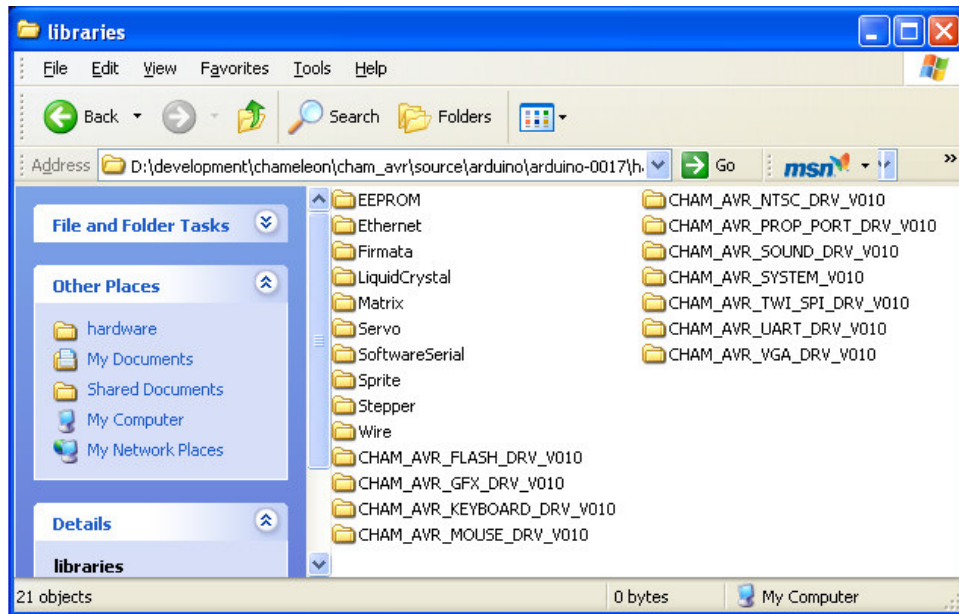
Figure 15.45 – The Arduino "Hardware" directory.

This is probably one of the most important directories, if not the most important directories of the Arduino installation. It contains a lot of goodies including two TXT files that describe the boards and programmers being used. So this is where you would add new boards and programmers. Also, there is a directory called **\BOOTLOADERS** – this is very important, its where all the .HEX files for the bootloaders are kept for all the flavors of Arduino. If you do happen to erase the Arduino bootloader accidentally or purposely and want to restore it, you will go into **\BOOTLOADERS** and pull the 328p bootloader .HEX file out and use that to program the AVR chip to restore it.

However, right now, we are interested in the **\LIBRARIES** directory. This is where all the Arduino "libraries" are kept. Technically, both the source C/C++ file and the object files are in this directory. This is how the MAKE file for Arduino finds things, it scans the **\LIBRARIES** directory and each directory name within is related 1:1 to a source library file name. Go ahead and dive into **\LIBRARIES** as shown in Figure 15.46.

Figure 15.46 – The Arduino \LIBRARIES directory.

These are all the libraries provided by the Arduino development system. Go ahead and take a look in a few to see what's in them, then come back to the root directory as shown above. Now, what we need to do is **copy** our new Chameleon libraries **into** this directory. So go back to the DVD in the Arduino folder and copy the contents of the **\LIBRARIES** sub-folder **into** your install folder. When you're done, it should look like the image shown in Figure 15.47.

Figure 15.47 – The Arduino Libraries folder after copying the Chameleon libraries into it.

As you can see, there are about a dozen new libraries that we have developed for the Chameleon to run in the Arduino environment. They are all uppercase to help you differentiate them.

Ok, that's it – the entire Arduino installation is complete! Now, let's fire up the tool, and set the environment up a little and try re-building the "Hello World" program.

15.2.3 Preparation to Launch the Arduino Tool for the First Time

Before you launch the Arduino tool, make sure you have the USB cable plugged into the Chameleon (you can also have the 9V DC power adapter plugged in if you have one). The USB is needed no matter what since that's what the Arduino tool will use to send serial data to the Chameleon's bootloader with.

Figure 15.48 below shows the USB cable plugged into the Chameleon AVR. The FTDI chip device makes it easy to use serial communications from the PC. Note that the Chameleons can be powered from the USB cable or an external 9V power supply OR both, but USB cable is needed for serial communications always.

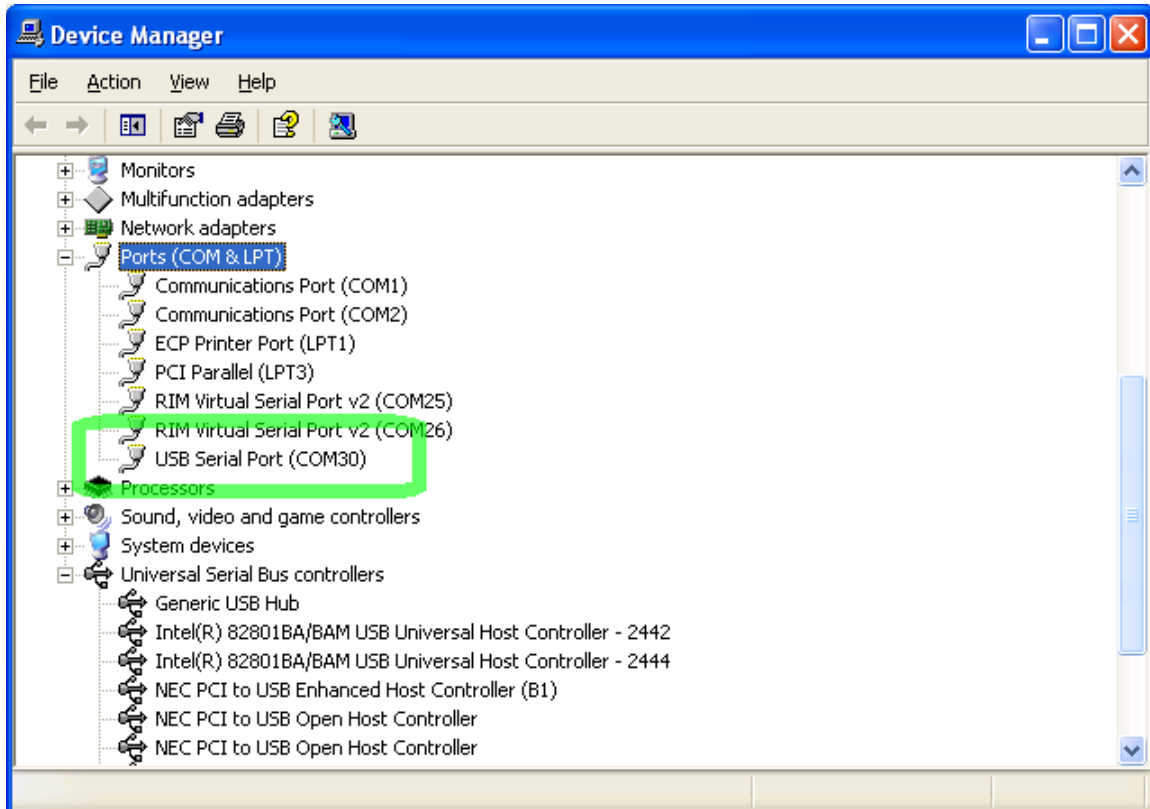
Figure 15.48 – The Mini-USB port on the Chameleon connects to the FTDI USB to Serial converter chip.

Now, if your computer recognizes the FTDI chip and has drivers it will load them automatically and install a virtual COMxx port into your system. However, if it doesn't you will need to install the FTDI drivers. This is needed for all serial communications with the Chameleon, so a necessary step. You can manually install the FTDI driver from the DVD-ROM (ZIP file), here:

DVD-ROM:\ CHAM_AVR \ TOOLS \ DRIVERS \ USBDriverInstallerV2.04.16.exe

After you have installed the driver, then each time you plug a Chameleon AVR into the PC via the USB port, the FTDI chip will assign a NEW COM port, you need to determine what **COM** port it attached itself to? Goto Windows **Start** menu on and select **<Control Panel → System Properties>**. select **<Hardware → Device Manager>** and select **<Ports>**, you should see something like that shown in Figure 15.49.

Figure 15.49 – Determining the COM port that the FTDI chip on the Chameleon AVR uses installation.



Make note of the “**USB Serial Port**”, in this case the driver installed on **COM30**, yours might be different, but you will need this when running the Arduino tool, Propeller tool and any terminal programs to talk to the Chameleon AVR for the serial RS-232 demos. So write this number down, for the next step of the tool setup.

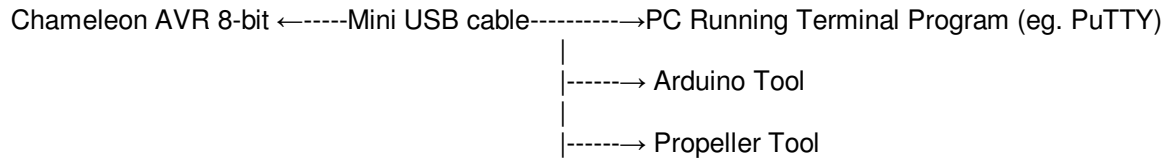
TIP!

Each time a new Chameleon with FTDI chip is plugged into the PC via the USB port, it will assign a NEW COM port, so if you own more than one Chameleon, or other products that use FTDI chips, you will more than one virtual COM port, so be advised!!!

Also, we might as well install a good serial program now, so you don't have to do it later. The Arduino tool has one built in as does the Propeller tool, but for general terminal experiments Hyperterminal won't cut it.

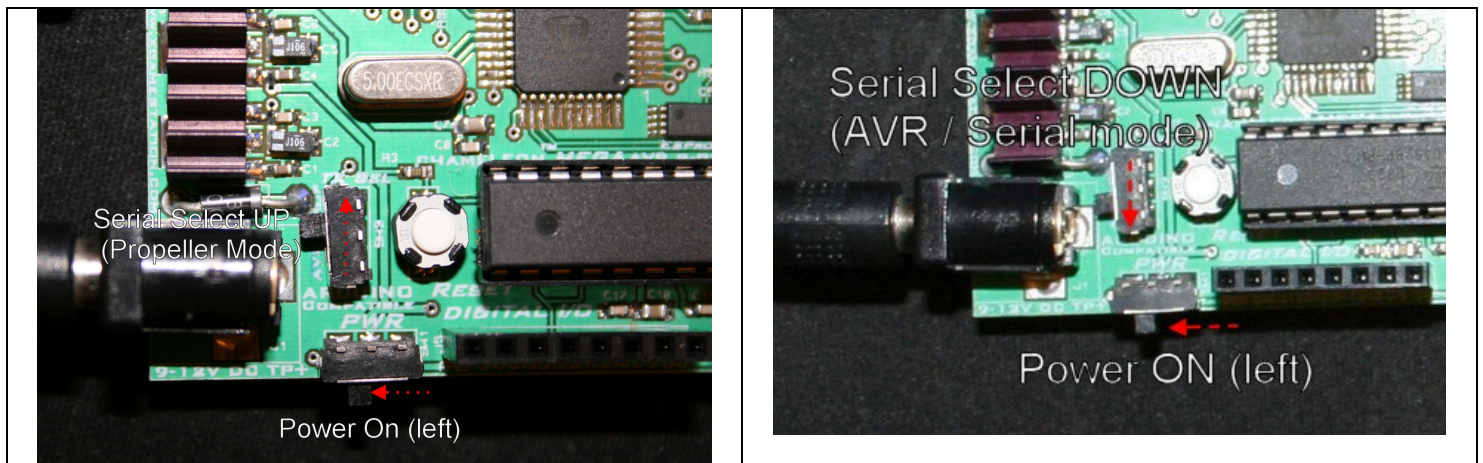
15.2.3.1 Installing a Serial Terminal Program

The **Chameleon AVR** has a USB **serial** port that is used to communicate with the Arduino tool as well as the Propeller tool. Both of these programs have built in serial communications, but if you just want to send out serial data to and from the Chameleon you are going to need a good terminal program, we suggest PuTTY.



The PC can talk to either the Propeller chip onboard the Chameleon via the USB port OR the standard serial port on the AVR chip, this is accomplished via a mechanical switch at the bottom of the board shown in Figure 15.49 below.

Figure 15.49 – The Serial port routing switch. In the UP position, it selects USB communication with the Propeller chip. In the DOWN position it selects communication with the AVR chip.

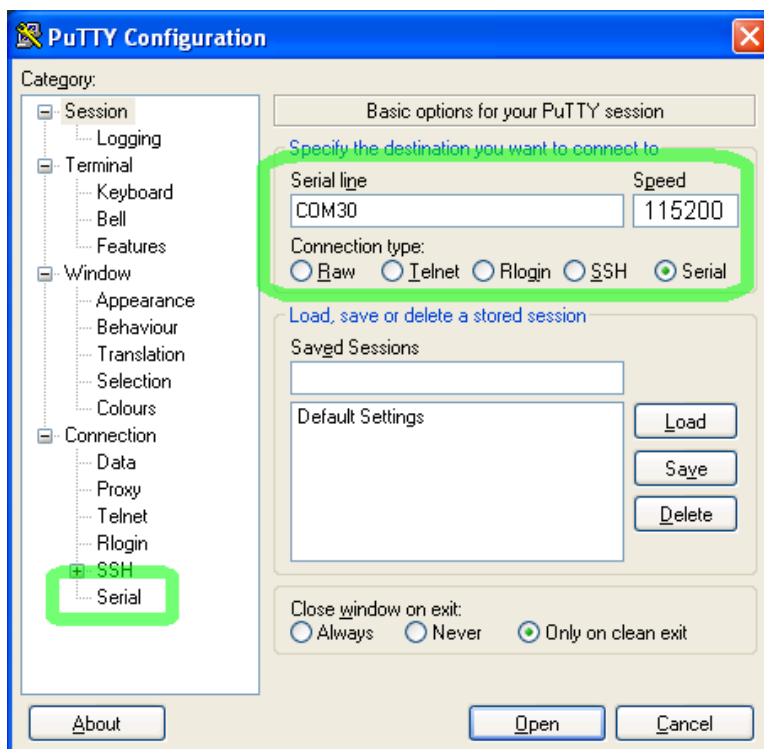


At this point, go ahead and make sure the serial select switch is in the DOWN position, so the PC can talk to the AVR thru the USB serial port.

There are many serial terminal programs available (Hyperterminal for example), but we want you to use one that is very SIMPLE to setup and that works well with the Chameleon AVR, we are going to use "**PuTTY**". From the DVD-ROM, install this program:

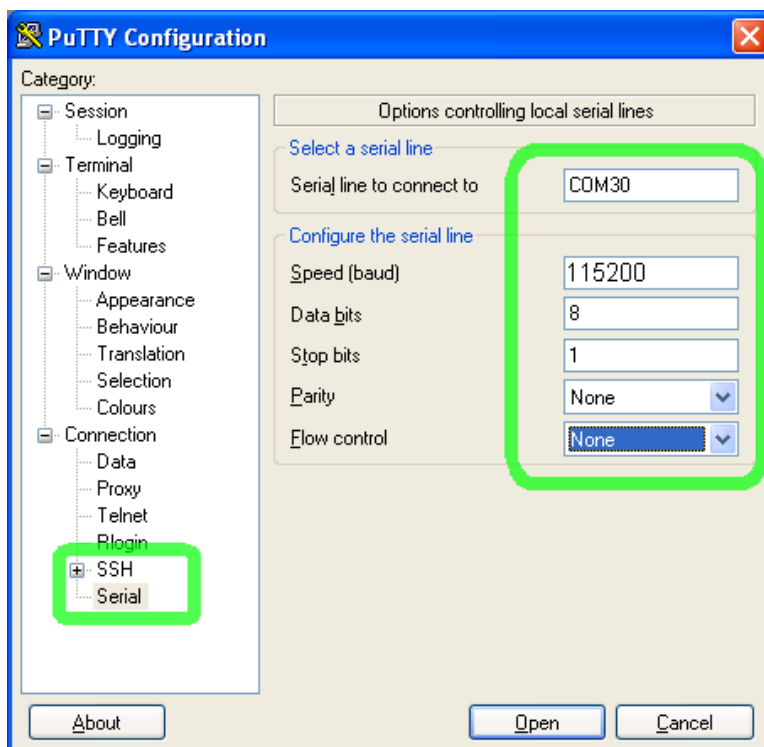
DVD-ROM : \ CHAM_AVR \ TOOLS \ COMMUNICATIONS \ putty_install.exe

It is called "**PuTTY**" and is very easy to use. Once installed, it will create a shortcut on your desktop. Click the shortcut to run the program, you will see a dialog like that shown in Figure 15.50.

Figure 15500 - PuTTY Setup for serial communications with Chameleon AVR, 115200 baud, N81.

Click on **<Serial>** for the connection type as shown in the top right, then click "**Serial**" on the bottom of the **Category tree** panel on the left and make sure the settings listed below and shown in Figure 15.51.

- Data Format: "N81", no parity, 8 data bits, 1 stop bit.
- Baud Rate: **115200**.
- Handshaking: None.

Figure 15.51 - PuTTY Setup for serial communications with Chameleon AVR, 115200 baud, N81, continued.

To launch the terminal program, you would press <Open> which opens a new terminal window. This is where you would type on the PC keyboard to communicate with the Chameleon AVR and run the demos or do serial terminal experiments.

15.2.3.1 Running the Arduino Tool

We are all ready to go, let's do a final checklist before we launch the tool:

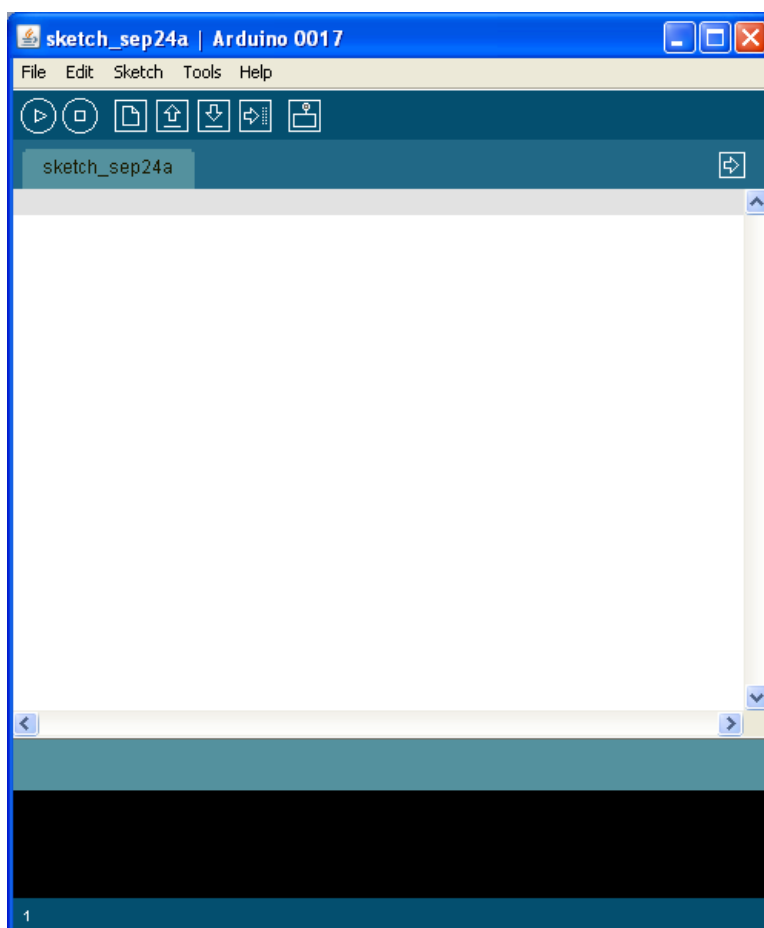
1. You have the USB cable plugged into the PC and you have installed the FTDI serial drivers if needed.
2. You have a 9V DC power supply plugged into the Chameleon (optional).
3. You have moved the serial select switch into the DOWN position, so the USB serial port is connected to the AVR chip (not the Propeller).
4. You have the VGA and NTSC cables connected to the Chameleon as well as a mouse (optional).

Finally, we are ready to launch the Arduino tool itself. If you take a look back in the installation directory where you decompressed the Arduino ZIP file (Figure 15.44) you will see an icon **ARDUINO.EXE**, go ahead and double click it. You should see the splash screen load shown in Figure 15.52.

Figure 15.52 – Arduino splash screen.



Then finally, the application will load and will look something like that shown in Figure 15.53.

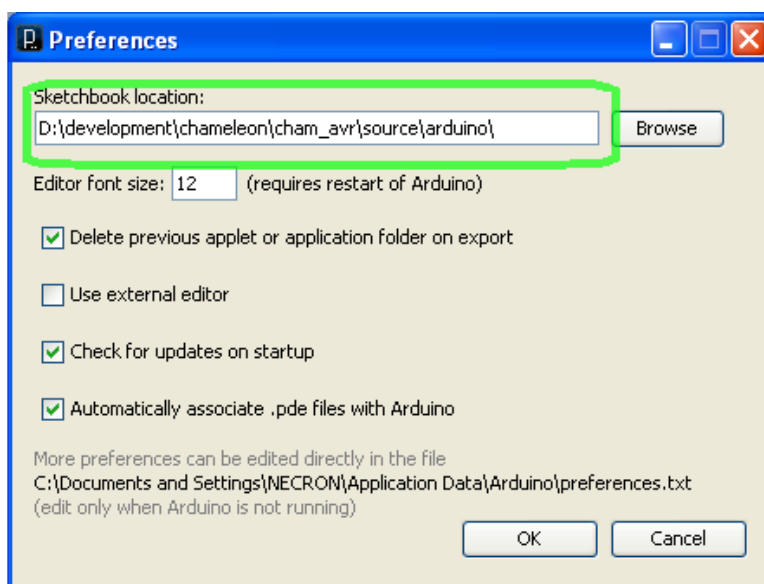
Figure 15.53 – the Arduino application at last!

Please read the Arduino documentation to learn all the features of the tool, but its fairly straightforward. It has a simple menu and a set of icons/buttons. Simply hover over each button see what they do. From left to right they are:



- Verify – Compiles only to show errors.
- Stop – Stops the serial monitor.
- New Sketch – Creates a new sketch.
- Open Sketch – Opens a sketch.
- Save Sketch – Saves the current sketch.
- Upload Sketch – Compiles and uploads the sketch to the Chameleon (Arduino).
- Serial Monitor – Launches a serial monitor terminal program.
- Tab Controls – Standard tab controls to add more views.

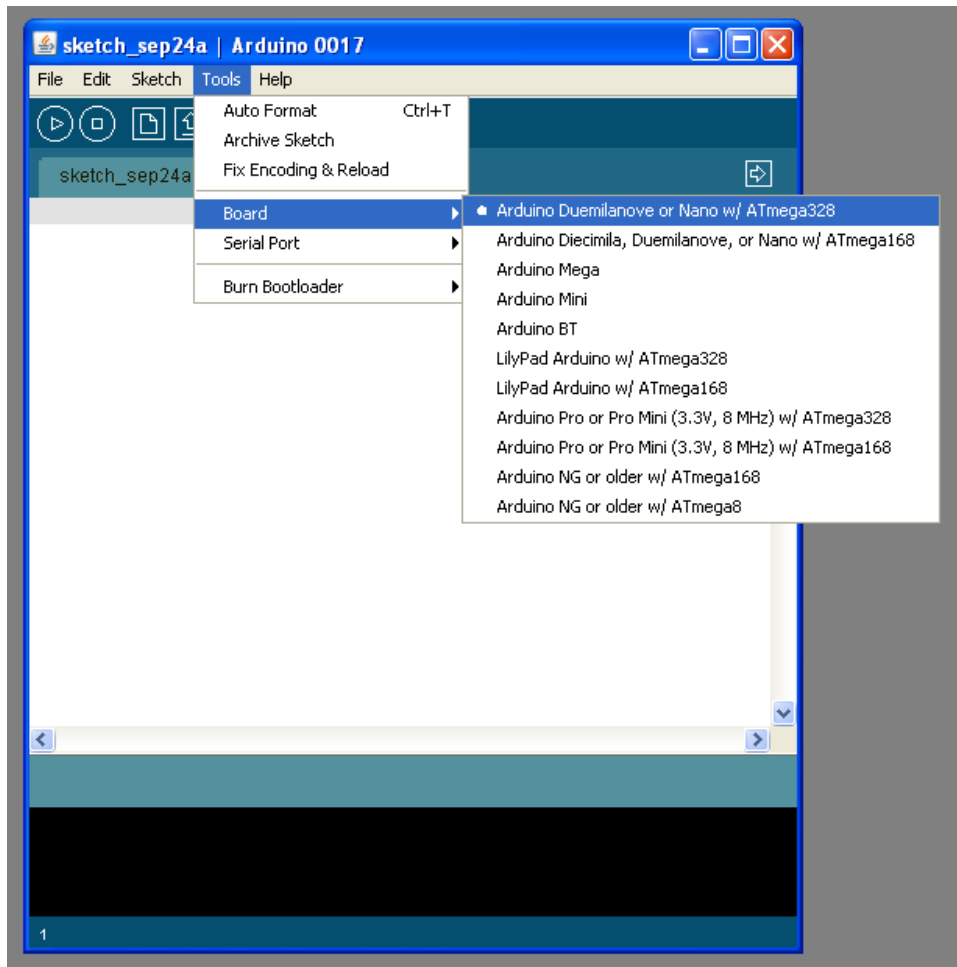
Ok, so the first thing we need to do is make sure that the Arduino tool can find our sketchbook since it likes to place sketches in a different place. If you recall, we copied our sketchbook from the DVD-ROM to the installation directory of the Arduino files, refer to Figure 15.43. So, first we need to tell the Arduino tool where to look. To do this, from the main menu select **<File → Preferences>** which should bring up the dialog shown in Figure 15.54 below.

Figure 15.54 – The Arduino tool preferences dialog.

Then browse for the location where you copied the sketchbook \SKETCHES from the DVD-ROM and point the “Sketchbook Location” to this directory that contains the folder. Click <OK>.

Setting the Proper Target

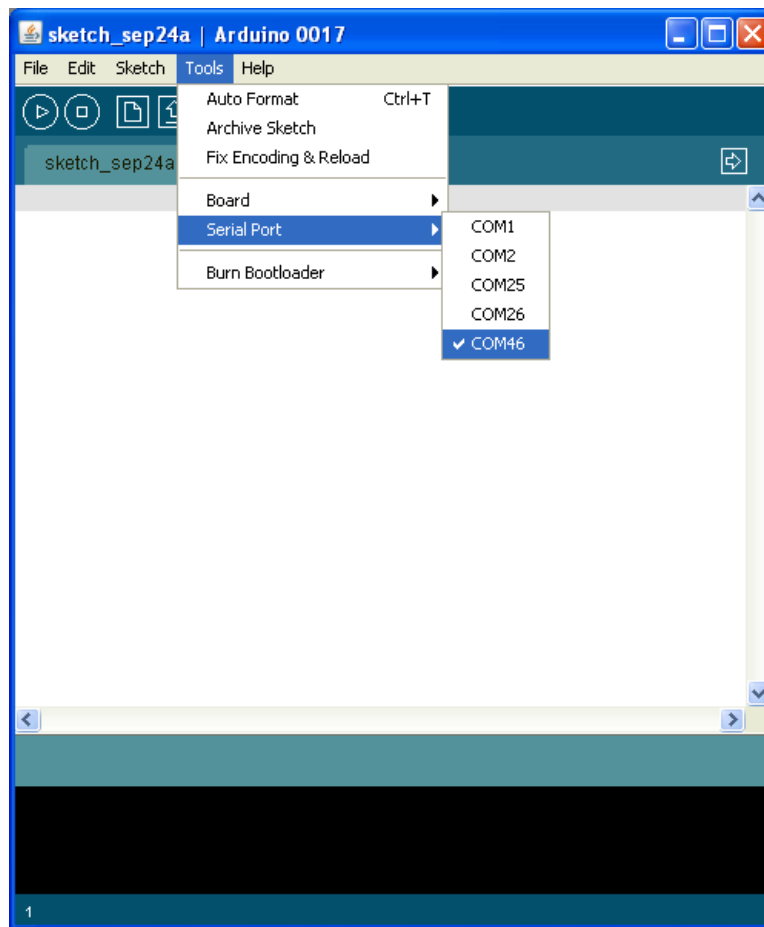
Next, we need to make sure that the Arduino tool has the correct serial port and the correct target chip. To do this, select the <Tools→Board> from the main menu and select “Arduino Duemilanove or Nano w/ ATmega 328” which is the top most item. This is shown in Figure 15.55.

Figure 15.55 – Selecting the proper AT328P target for the Arduino tool.

Next, we need to make sure that the Arduino tool is using the proper serial port. You should have noted this earlier when you plugged the USB cable into the Chameleon and the PC recognized the FTDI USB to serial chip. If you haven't done then, plug the USB cable in, and go thru the installation of the FTDI drivers. Then take note in the Control Panel where the new USB serial port was installed, you need that information for this step.

Setting the COM Port

To set the Arduino tool's COM port, simply select from the main menu **<Tools → Serial Port>** as shown in Figure 15.56 and select the proper serial port.

Figure 15.56 – Selecting the serial port that the USB serial chip is connected to.

15.2.3.2 Loading the Hello World Sketch

A sketch is like a project. For each sketch there is a single directory and in that directory, there is a main file with the extension .PDE. This is your working “main” C/C++ files. Additionally, there is a directory in each sketch named “**applet**”. This directory contains all the source files, configurations, etc. for that particular configuration, so don’t mess with it! However, you can copy a sketch from the sketches directory to another system or the entire sketches directory usually without a problem.

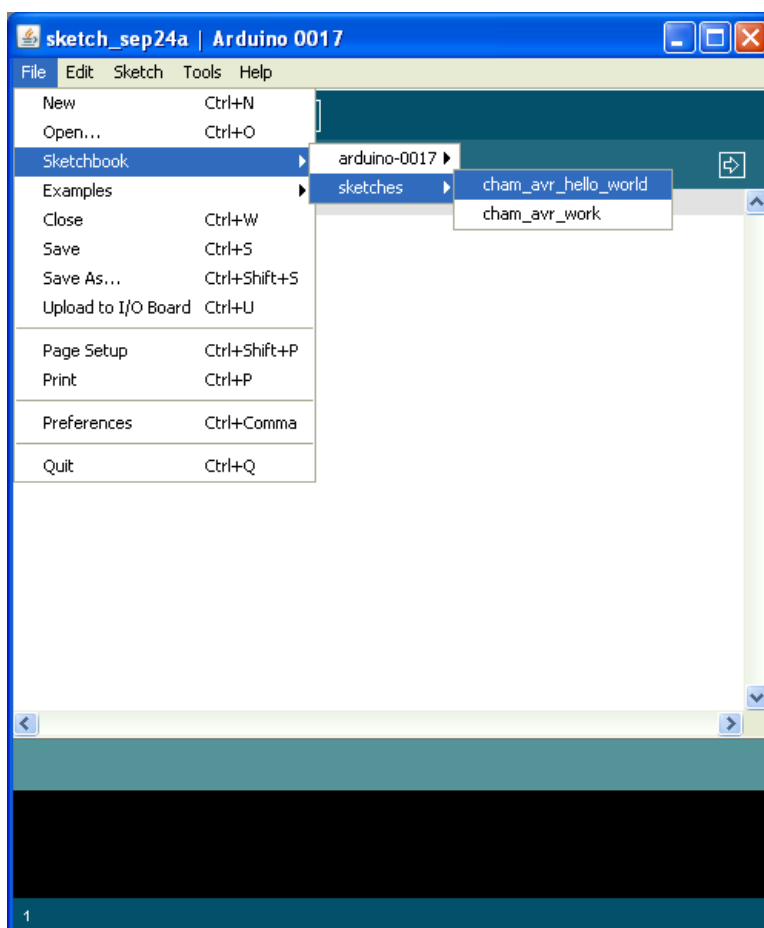
In any event, let’s load a “Hello World” sketch into the Arduino tool, compile and download to the Chameleon. Here are the steps to get things ready:

Step 1: Make sure the Chameleon is powered, the USB cable is plugged into it.

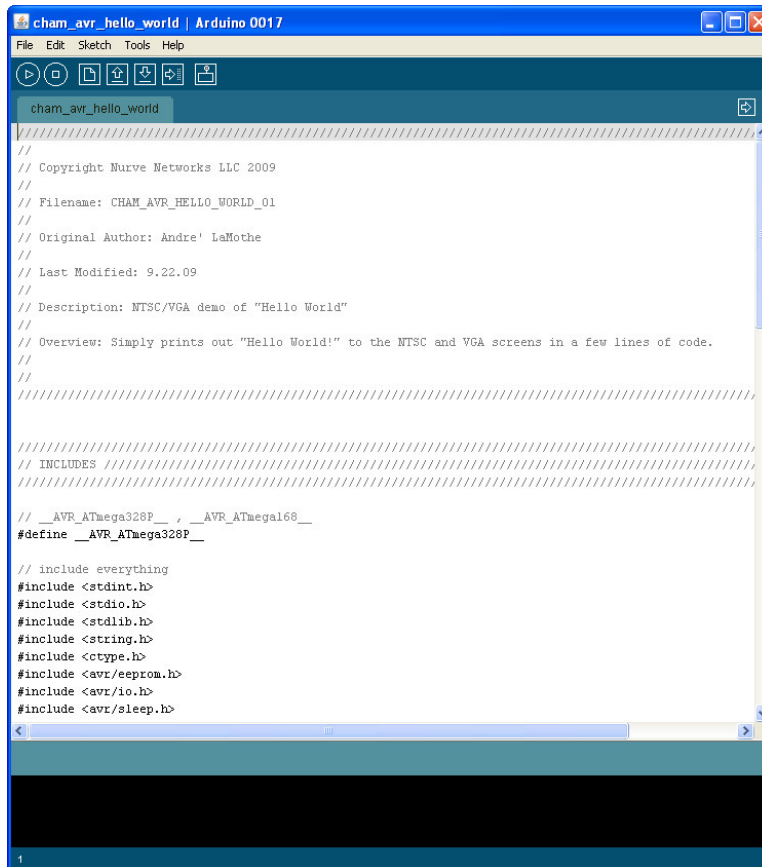
Step 2: Make sure you have the NTSC and VGA monitor connected, at least one of them.

Step 3: Make sure you have the serial selection switch in the DOWN position to allow AVR serial traffic (UP is Propeller traffic).

Step 4: On the main menu goto **<File → Sketchbook → Sketches>** and locate the sketch named “**cham_hello_world**”, select it – it should load into the tool. This is shown in Figure 15.57 below.

Figure 15.57 – Loading our first “Hello World” sketch.

Once you load the sketch the tool should load the source for the “main” program as shown in Figure 15.58 below.

Figure 15.58 – The “Hello World” program loaded into the Arduino tool.

Go ahead and scroll thru the program a bit, its very short. Once you are done, now its time to finally download it to the Chameleon. Simply press the Compile button  this will both compile and tell us if there are any errors as well as show the final size.

WARNING!

The Arduino tool has terrible user diagnostics. While compiling or downloading it just sits there without any updates in the text output area at the bottom of the screen or status. This is quite disturbing, so just hold tight while the program compiles and downloads it could take moments.

When the compilation is complete (there should be no errors), you should see the following text appear in the black text output window at the bottom of the tool:

Binary sketch size: 1034 bytes (of a 30720 byte maximum)

Now that we know the program is correct, lets upload it to the chameleon with the  button. Go ahead and press the button, this will take some time...When the program is done uploading, the Chameleon will immediately reboot and you should see the “Hello World” text scrolling on both the NTSC and VGA monitors. Also, in the black text output area of the Arduino tool you should see the message:

Binary sketch size: 1034 bytes (of a 30720 byte maximum)

That's it - you have compiled your very first Chameleon program using the Arduino tool! Hopefully, you can see the incredible power of the Chameleon. It leverages the toolchain and main processor of the Arduino, but adds a very powerful multicore processor (the Propeller) to do the heavy lifting. And with literally 10 lines of code you are displaying text on BOTH an NTSC and VGA screen! Imagine the possibilities!!!

15.2.3.3 A Couple Notes About the Arduino Version of Hello World

When we get to the **Programming API**, we will discuss this a little more. But, you might be wondering what are the differences between the pure C/C++ versions of our programs that compile on AVRStudio and the Arduino versions? Well, there aren't many. We wrote all the libraries to be 100% compatible. In fact, the differences can be listed in a few sentences, they are:

1. The Arduino run-time library has its own **main()**, thus in all our programs, we just rename **main()** to **loop()** and then add another function called **setup()** (which can be empty). **Setup()** is simply called first for all Arduino programs then the **loop()** function is called just like **main()** and the program stays in there.
2. The Chameleon under AVRStudio has a number of libraries. One of them that is redundant is the serial UART library. We had to write one for the AVRStudio version, but the Arduino has a serial library of its own. So you will notice that a lot of the programs that use serial comment out the include file for the Chameleon serial library and we just use the Arduino built in serial library (which is a C++ class).
3. When including headers with AVRStudio its typically setup to look in the local directories for the headers, so you put quotes "header.h" around the header names. Whereas, the Arduino tool makes your headers part of its libraries, so you use angled brackets **<header.h>**.

That's really about it. It takes less than 5 mins to port most Chameleon programs to Arduino mode and vice versa. Of course, we have done this for you for all the demos that follow to get you started.

So the question is which mode to use – Arduino or AVRStudio? Well, depends on what you are doing and what you have. If you do not own a **AVR ISP MK II** programmer then the decision is made for you – you probably have to use Arduino mode. But, if you do have the programmer or something else supported by AVRStudio then you probably should work in the more robust AVRStudio since it allows ASM language, simulation, etc.

But, its up to you – just have fun!

15.3 Installing the Parallax Propeller IDE

Since each Chameleon AVR (and PIC) has **dual** processors consisting of an **AVR** and Parallax **Propeller** chip, both chips need to be loaded with firmware. and accessed to program To program the Parallax Propeller chip, you need to install the **Parallax IDE tool**. This tool is a very simply IDE that allows you edit programs for the Propeller chip and download them to the target. The editor has no built in debugging which is really a result of the Propeller chip's lack of debugging internally in hardware. However, this isn't as bad as it seems. You can use "**printf**" techniques as well as run debug like processes on other cores and have them interrogate memory locations in real-time. Finally, its pretty easy to test code, change something, recompile and download to the target since programs can only be up to 32K bytes in size.

The **Propeller Chip IDE** was developed by **Parallax Inc.** to program their new multi-core "**Propeller**" chip. The Chameleon AVR (and PIC) uses a Propeller chip as the media processor, thus to program the Propeller chip's EEPROM on the Chameleon with code we need to use this tool. The Propeller is a 32-bit processor and you program it in a BASIC like language called SPIN, or you can program in Assembly language. The Propeller IDE supports both seamlessly. The assembly language is very nice, but SPIN takes a bit to get used as does the editor which uses indentation like Python for block level. This can be frustrating for C/C++ programmer initially. You can read about programming the Propeller chip here:

DVD-ROM \ CHAM_AVR \ TOOLS \ PROPELLER \ Propeller_Manual_WebPM-v1.1.pdf

DVD-ROM \ CHAM_AVR \ TOOLS \ PROPELLER \ PropellerDatasheet-v1.2.pdf

Also, there is a single book about developing games and media applications for the Propeller chip:

"Game Programming the Propeller Powered HYDRA" by Andre' LaMothe.

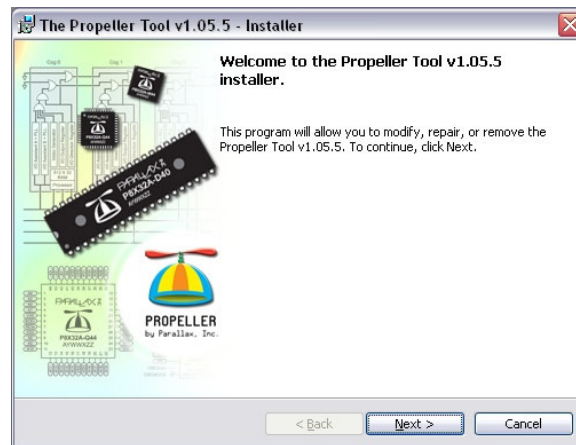
The installation procedure for the Propeller IDE is very simple and consists of the Propeller tool itself along with some "Virtual COM" port drivers developed by FTDI inc. that are needed to communicate with the Propeller Chip itself. These drivers are installed as part of the installer program if everything goes smoothly. However, you might recall we have already installed the FTDI drivers while setting up the Arduino tool. As you can see **everyone** seems to use FTDI chips!!!

Step 1: Locate the Propeller IDE installation program **Setup-Propeller-Tool-v1.2.6.exe** on the DVD here:

DVD-ROM :\ CHAM_AVR \ TOOLS \ PROPELLER \ Setup-Propeller-Tool-v1.2.6.exe

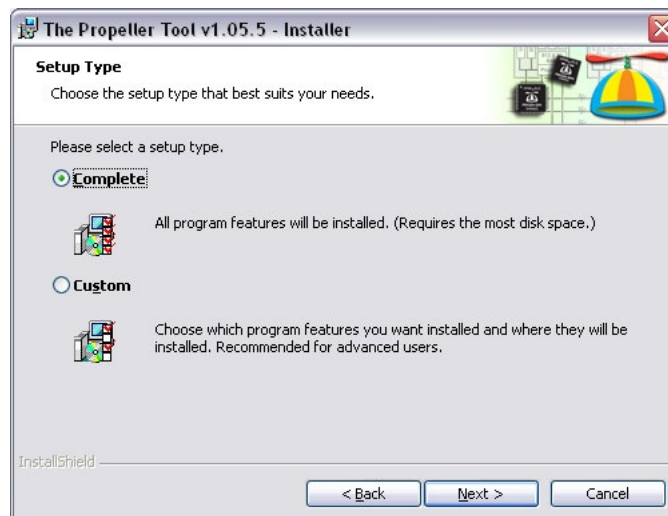
and launch it, you will see the installer splash screen as shown in Figure 15.59 below. Click **<NEXT>**.

Figure 15.59 – The Propeller IDE installation splash screen (your version # might vary).

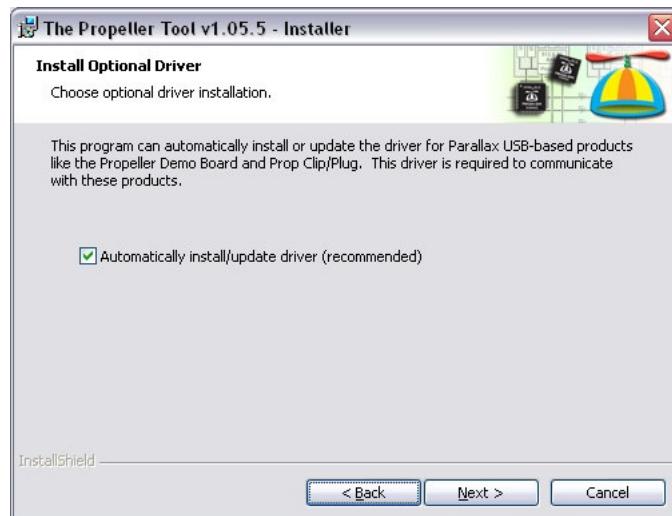


Step 2: The installation type should be displayed next as shown below in Figure 15.60, select “**Complete**” install, click **<NEXT>**.

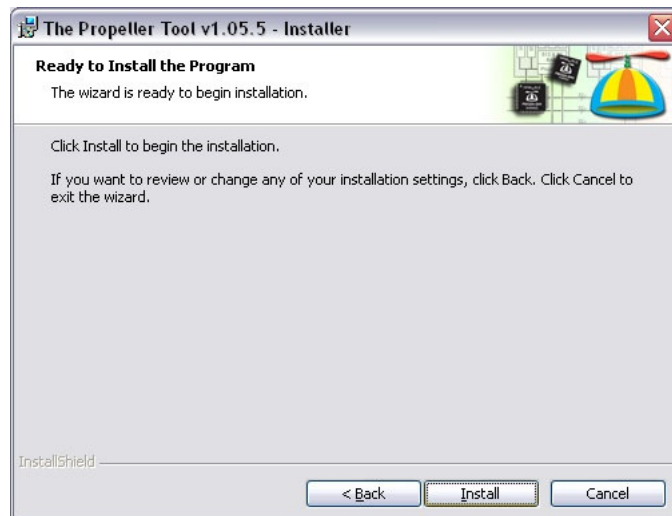
Figure 15.60 – Selection of Complete install.



Step 3: VERY IMPORTANT! The Propeller tool IDE communicates to the Chameleon AVR using a USB cable and driver, this USB driver must be installed. The next dialog shown below in Figure 15.61 should have the “**Automatically Install/Update driver**” selected, so the driver is installed. Check this box and click **<NEXT>**.

Figure 15.61 – Automatic USB driver installation dialog.

Step 4: The tool is ready to be installed, Figure 15.62 below shows the installation ready dialog, click **<INSTALL>** to begin the installation.

Figure 15.62 – The final installation dialog before the software is installed.

Step 5: As the software installs, the first thing it will do is install the FTDI inc. USB drivers that the Propeller Tool relies on, so you should see a message box like that shown in Figure 15.63 below.

Figure 15.63 – The FTDI USB drivers successfully installed.

Step 6: Click **<OK>** and the final installation should take place, you will see a dialog like that shown in Figure 24.0 below, click **<FINISH>** and this should complete the installation of the Propeller Tool IDE and the associated USB drivers it needs.

Figure 15.64 – The final Propeller tool installation dialog.

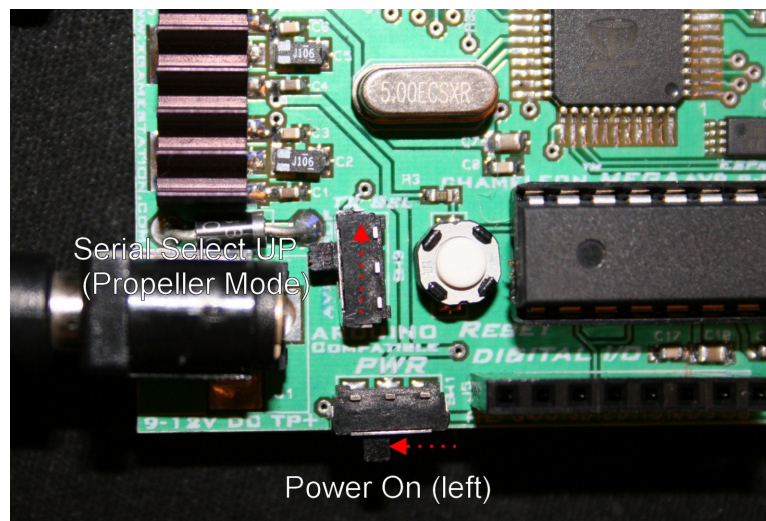
TROUBLESHOOTING

If you have trouble installing the USB drivers, there is another “stand alone” installer that will do a “force install” on them. The name of the file is: **USBDriverInstallerV2.04.16.exe**, it's located in the directory **CHAM_AVR \ TOOLS \ DRIVERS ** and usually will deal with any problems encountered during normal installation.

At this point, the Propeller tool should be installed and ready to go. Let's test it out.

15.3.1 Launching the Propeller Tool

The Propeller tool communicates to the Propeller chip over a single 2-line serial connection with the standard RS-232 TX/RX pair along with DTR (data terminal ready) as a reset line. The Chameleon's USB serial port is used for the communications channel for the Propeller chip. However, if you recall, the USB serial port can be used by the AVR or the Propeller. The device that gets connected to the port is controlled by the serial selection switch next to the RESET button. Simply place the switch in the UPPER position to select the Propeller chip to have access to the USB serial port. This is shown in Figure 15.65 below.

Figure 15.65 – Placing the Chameleon into Propeller serial mode.

You already installed the Propeller tool, so all we have to do now is run the tool which is very simple. The first step is to make sure that the PC recognizes the Chameleon and installs a virtual USB COM port. When you connected your test Chameleon AVR to the PC via the USB cable and powered the Chameleon up, the FTDI drivers on the PC should have detected a NEW USB port and assigned a COM port to it, you can find the COM port # in the system devices menu. You will need this later for the serial communications test, however, the Propeller tool will autoscan all COM ports until it finds the Propeller chip connected to one. With that in mind, here are the steps to load the firmware into the Propeller chip, you will do this for every Chameleon AVR (and PIC) in the final testing process.

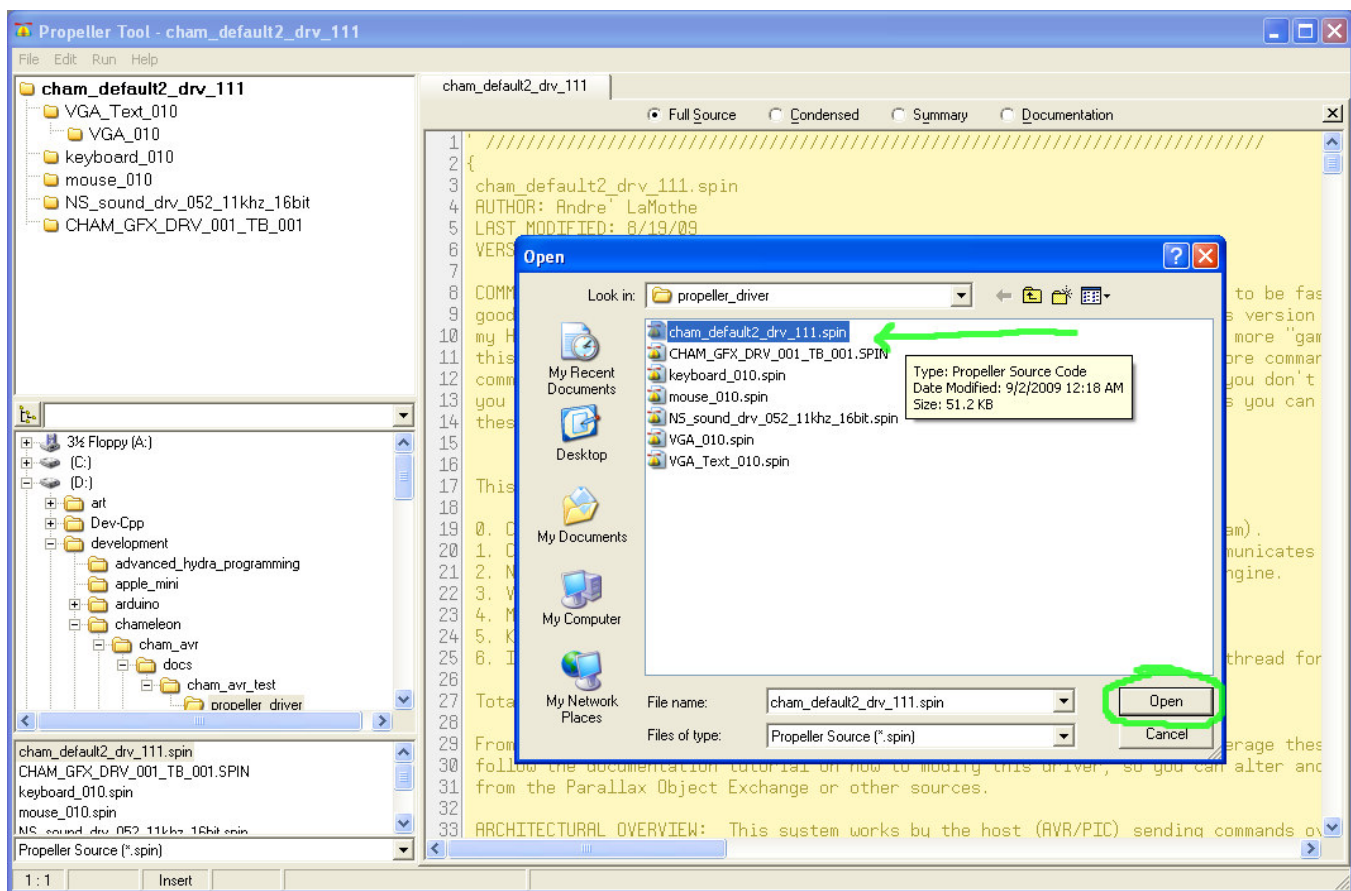
Step 1: Launch the Propeller IDE tool by clicking the shortcut on the desktop.

Step 2: Navigate to where you copied the source directory files on your PC and open the master file **cham_default2_drv_112.spin** (latest copy). It is located on the DVD-ROM here:

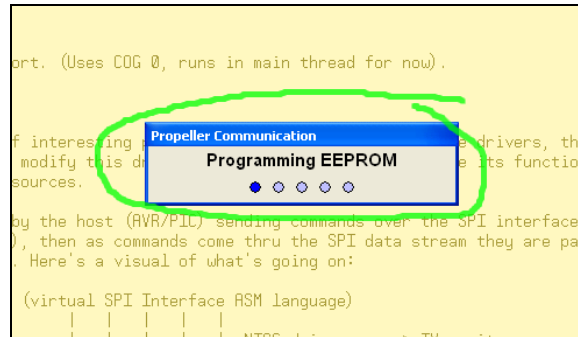
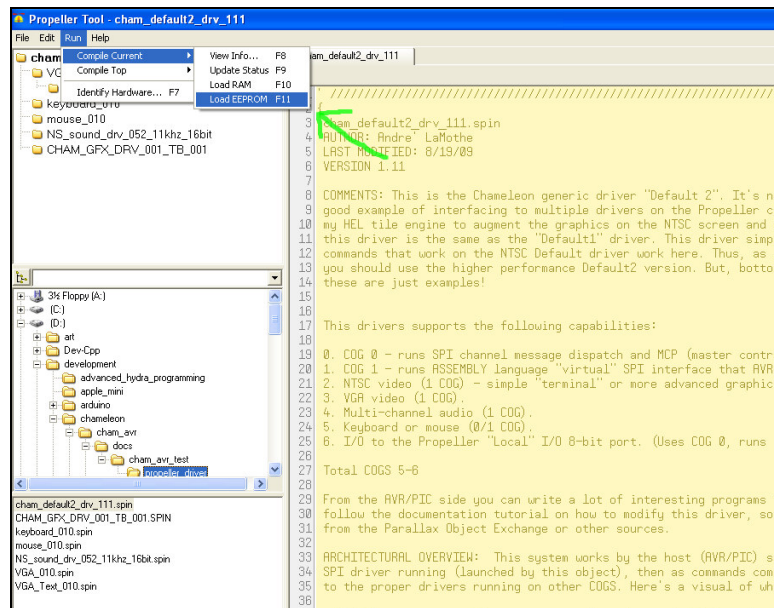
**DVD-ROM : \ CHAM_AVR \ SOURCE \ PROPELLER_DRIVER **

This is shown in Figure 15.66 (ignore my slightly different hard drive paths).

Figure 15.66 – Loading the Propeller firmware into the tool.



Once the source code is loaded into the tool, you simply have to download it into the Chameleon AVR. Make sure that you have the USB cable **plugged** into the Chameleon, power is **ON**, and the serial selection switch is in the **UP** position. When you are ready then you can press <F11> on the Propeller Tool, or from the main menu, you can select **<RUN> Compile Current -> Load EEPROM**; the results will be the same, the Propeller tool will scan all COM ports for a connection to a Propeller chip, and then compile and download the firmware to the Chameleon. This is shown in Figure 15.67.

Figure 15.67 – Downloading the firmware into the Propeller Chip.**TIP**

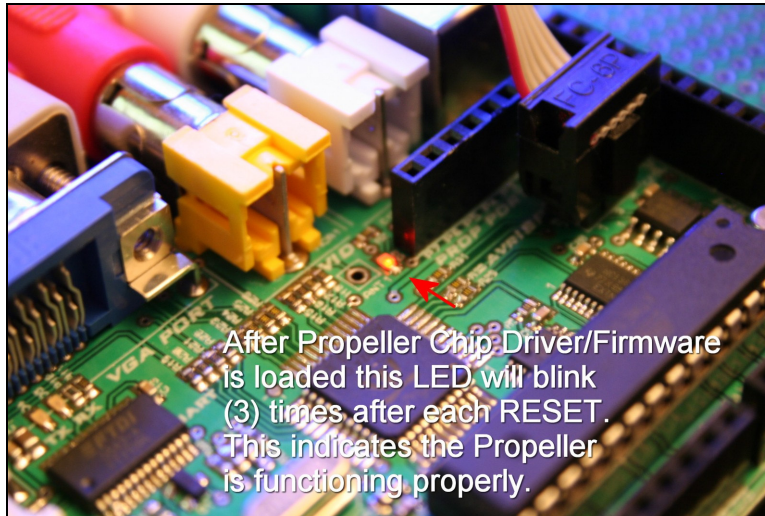
If the Propeller tool can't find the Propeller chip, then try unplugging the USB cable from the Chameleon, hit the **reset** switch, make sure that the serial select switch is in the **UP** position.

Once the firmware is downloaded into the Propeller chip on the Chameleon board it will immediately start generating NTSC and VGA video. If you hit RESET on the Chameleon, you will see the images shown in Figure 15.68 on the NTSC and VGA monitors respectively.

Figure 15.68 – The NTSC and VGA monitors showing the Chameleon AVR's output after installing the Propeller driver firmware.

Also, when you hit reset each time you will see the **LED D3** under the Video port blink (3) times (shown in Figure 15.69, this is part of the testing firmware to confirm the Propeller chip is functioning correctly as well as to verify the LED is good. Finally, you will **HEAR** the Chameleon make sounds as it boots up, a series of tones that play musical notes when you hit RESET. Make sure to have the **audio port** plugged into your NTSC TV as well. These will all be part of the final tests for each Chameleon.

Figure 15.69 – The LED will blink whenever you reset the Chameleon once you have loaded the Propeller driver firmware.

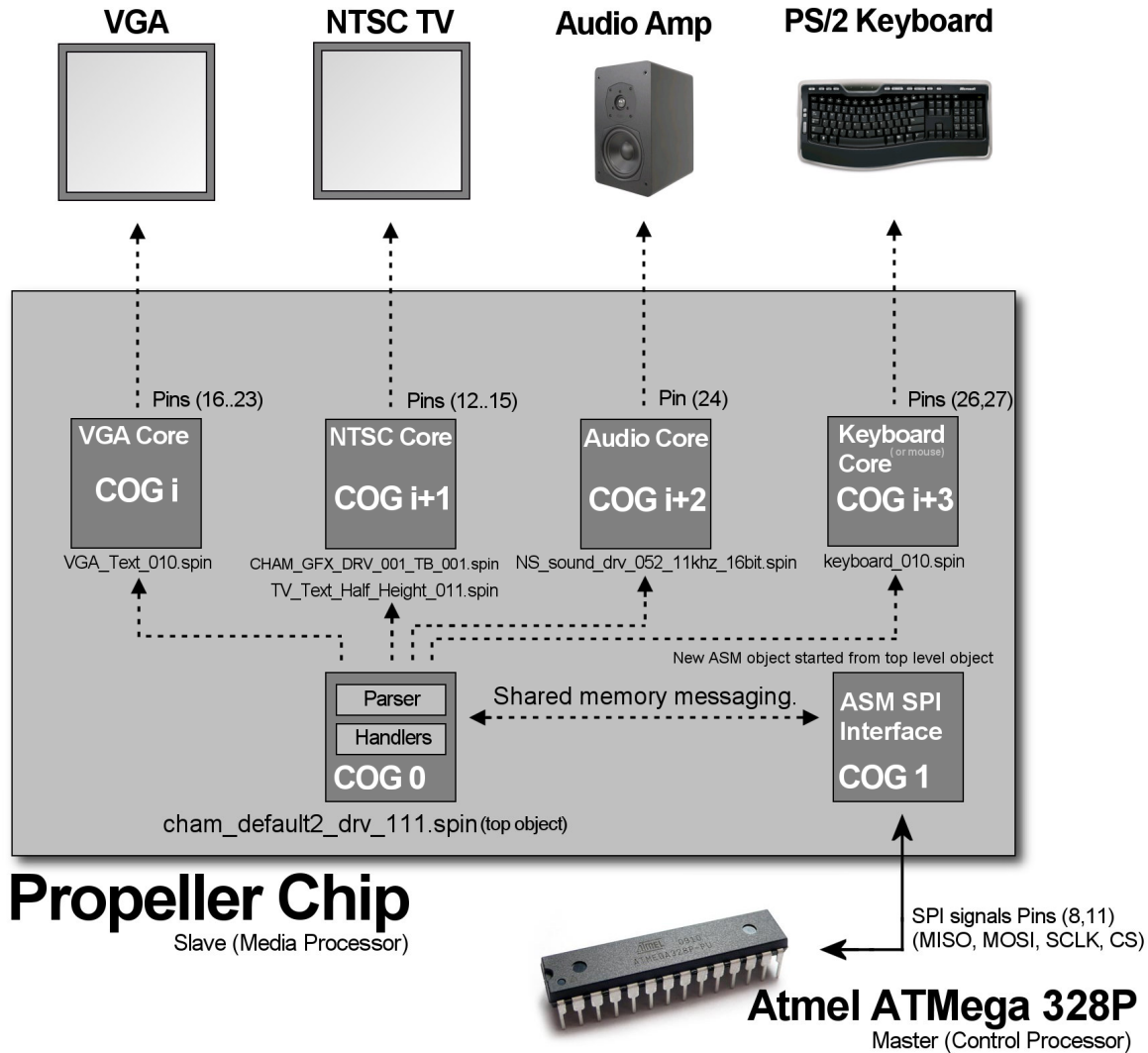


Of course, all we really did was re-download the SAME firmware back into the Propeller chip, so whatever is loaded into the AVR at this point will ultimately direct the Propeller slave what to do. But, this is the process to load the Propeller chip with a program. Additionally, you just saw how to load the default driver that this manual uses 99% of the time which is loaded on the Chameleons when they ship. But, you can surely change this driver if you wish, update it, make it faster, better, change objects it relies on etc. You will learn how to do this later in the manual – update the Propeller driver, but for now, we aren't going to touch it other than knowing where it is, and how to download it. This is the idea of the Chameleon – the Propeller chip is like a black box, when things go right, we shouldn't even know its there, the Propeller should work like another other media processing chip and just execute the command we send it.

16.0 Chameleon Inter-Processor Architecture Overview

In this section, we are going to go into a bit of detail in relation to the software engineering techniques that the Chameleon's dual processor architecture is based on. This section isn't necessary to use the Chameleon, but its necessary to understand the design more completely.

Figure 16.1 - System level modular schematic.



Referring to Figure 16.1, the Chameleon's Propeller co-processor has a number of connections to various media I/O devices. These include NTSC/PAL composite video, VGA, audio out, serial (not shown), and PS/2 keyboard/mouse devices. This uses up a minimum of (5) processing cores, leaving us (3) cores for other things and for future expansion. Of course, two cores are used for the **Master Control Program** that runs on the Propeller as well as the SPI virtual driver. Thus we are left with (1) core ultimately for expansion unless we modify the default Propeller driver. As shown in the figure, the NTSC signal is on Pin 12,13,14, VGA is on pins 16...23, audio is on pin 10, the PS/2 keyboard/mouse is connected to pins 26,27 and finally the serial is on pins 30,31. Basically, this is a standard Propeller Board Rev. C/D pin map.

TIP

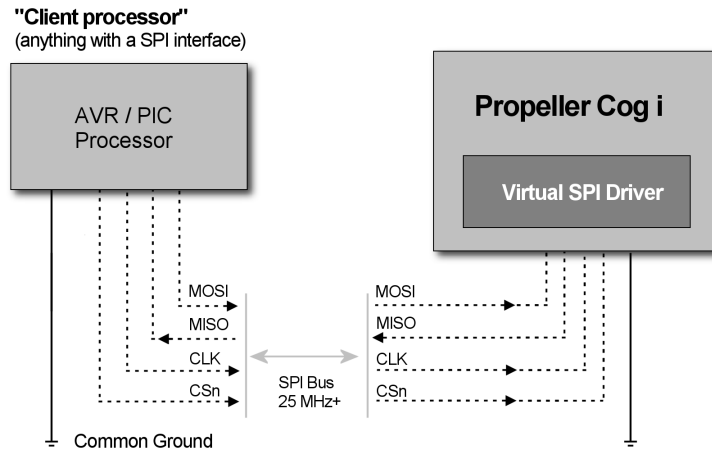
This pin map was selected to keep the Propeller sub-system very compatible with Propeller development boards, so the Chameleon could double as a Propeller development kit!

Table 16.1 shows the pin map in more detail.

Table 16.1 - The I/O pin map for the connections to media devices.

I/O device	Propeller IO Port Pins Used
NTSC video	12, 13, 14 (video LSB to MSB)
VGA video	16(V),17(H),18(B1),19(B0),20(G1),21(G0),22(R1),23(R0)
PS/2 Keyboard	26 (data), 27 (clock)
Sound	10 (PWM or pure signal)
Serial Coms	30 (TX), 31 (RX)

Figure 16.2 – The SPI link between the AVR and Propeller chip.



16.1 Master Control Program (MCP)

The MCP (**cham_default2_drv_112.spin**) is the primary driver that runs on the Propeller chip and listens to SPI traffic over the SPI link from the AVR chip as shown in Figure 16.2. As the virtual SPI interface (running on its own core) receives packets (messages/commands) from the AVR, the messages are parsed and packaged and placed into a global shared memory region. The MCP is listening to the global shared region and tracks a flag. Once this flag is noted to change, the MCP reads the packet in and then tries to parse it to determine if it's a valid command that is supported. This is achieved via a large switch statement to see if the command portion of the packet is valid. Below is the list of current commands supported by the MCP by the default driver **cham_default2_drv_112.spin**:

```
' null command, does nothing
CMD_NULL = 0

' graphics commands
GFX_CMD_NULL = 0

' NTSC commands
GFX_CMD_NTSC_PRINTCHAR = 1
GFX_CMD_NTSC_GETX = 2
GFX_CMD_NTSC_GETY = 3
GFX_CMD_NTSC_CLS = 4

' VGA commands
GFX_CMD_VGA_PRINTCHAR = 8
GFX_CMD_VGA_GETX = 9
GFX_CMD_VGA_GETY = 10
GFX_CMD_VGA_CLS = 11

' mouse and keyboard are "loadable" devices, user must load one or the other on boot before using it

' keyboard commands
KEY_CMD_RESET = 16
KEY_CMD_GOTKEY = 17
KEY_CMD_KEY = 18
KEY_CMD_KEYSTATE = 19
```

```

KEY_CMD_START      = 20
KEY_CMD_STOP       = 21
KEY_CMD_PRESENT    = 22

' mouse commands
MOUSE_CMD_RESET     = 24 ' resets the mouse and initializes it
MOUSE_CMD_ABS_X     = 25 ' returns the absolute X-position of mouse
MOUSE_CMD_ABS_Y     = 26 ' returns the absolute Y-position of mouse
MOUSE_CMD_ABS_Z     = 27 ' returns the absolute Z-position of mouse
MOUSE_CMD_DELTA_X   = 28 ' returns the delta X since the last mouse call
MOUSE_CMD_DELTA_Y   = 29 ' returns the delta Y since the last mouse call
MOUSE_CMD_DELTA_Z   = 30 ' returns the delta Z since the last mouse call
MOUSE_CMD_RESET_DELTA = 31 ' resets the mouse deltas
MOUSE_CMD_BUTTONS   = 32 ' returns the mouse buttons encoded as a bit vector
MOUSE_CMD_START     = 33 ' starts the mouse driver, loads a COG with it, etc.
MOUSE_CMD_STOP      = 34 ' stops the mouse driver, unloads the COG its running on
MOUSE_CMD_PRESENT   = 35 ' determines if mouse is present and returns type of mouse

' general data readback commands
READ_CMD            = 36

' sound commands
SND_CMD_PLAYSOUNDFM = 40 ' plays a sound on a channel with the sent frequency at 90% volume
SND_CMD_STOPSOUND   = 41 ' stops the sound of the sent channel
SND_CMD_STOPALLSOUNDS = 42 ' stops all channels
SND_CMD_SETFREQ     = 43 ' sets the frequency of a playing sound channel
SND_CMD_SETVOLUME   = 44 ' sets the volume of the playing sound channel
SND_CMD_RELEASESOUND = 45 ' for sounds with infinite duration, releases the sound and it enters the "release"
portion of ADSR envelope

' propeller local 8-bit port I/O commands
PORT_CMD_SETDIR     = 48 ' sets the 8-bit I/O pin directions for the port 1=output, 0=input
PORT_CMD_READ       = 49 ' reads the 8-bit port pins, outputs are don't cares
PORT_CMD_WRITE      = 50 ' writes the 8-bit port pins, port pins set to input ignore data

' general register access commands, Propeller registers for the SPI driver cog can be accessed ONLY
' but, the user can leverage the counters, and even the video hardware if he wishes, most users will only
' play with the counters and route outputs/inputs to/from the Propeller local port, but these generic access
' commands model how you would access a general register based system remotely, so good example
' these commands are DANGEROUS since you can break the COG with them and require a reset, so if you are going to
' write directly to the registers, be careful.

REG_CMD_WRITE       = 56 ' performs a 32-bit write to the addressed register [0..F] from the output register
buffer
REG_CMD_READ        = 57 ' performs a 32-bit read from the addressed register [0..F] and stores in the input
register buffer
REG_CMD_WRITE_BYTE  = 58 ' write byte 0..3 of output register g_reg_out_buffer.byte[ 0..3 ]
REG_CMD_READ_BYTE   = 59 ' read byte 0..3 of input register g_reg_in_buffer.byte[ 0..3 ]

' system commands
SYS_RESET           = 64 ' resets the prop

```

As you can see, there is a lot of commands supported. This is necessary, so the AVR master/client can issue commands to the Propeller/slave to do what it needs it to do. However, this is only a template, you can re-write the default2 driver if you wish, modify, optimize, change the objects it uses etc. But, you will have to modify all the APIs on the AVR side as well with the new commands and SPI API calls, so we suggest you start simply by adding commands and/or modifying commands that already exist.

Once a valid command is detected then its parsed and a case handler in the MCP tries to dispatch it to the proper driver object (themselves running on other cores potentially). An excerpt from that section of the driver code looks like this:

```

' ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
' MAIN LOOP - Listen for packets to finish then execute commands sent to Prop over SPI interface
' the longer this loop is the slower processing will be, thus you will want to remove unnecessary driver support for
' your custom drivers and of course port to ASM and blend this with the virtual SPI driver code as well for the
' best performance.
' ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
repeat
' wait for SPI driver to receive a command packet in shared memory
if ( (g_spi_cmdpacket & $1_00_00_00) > 0)
' reset the reception flags
g_spi_buffer := (g_spi_buffer & %0_1111_1111)
g_spi_cmdpacket := (g_spi_cmdpacket & $FF_FF_FF)

' extract cmd 8-bit, data 8-bit, status 8-bit
g_cmd := ((g_spi_cmdpacket >> 16) & $FF)
g_data := ((g_spi_cmdpacket >> 8) & $FF) ' low byte of 16-bit data
g_data2 := g_status := ((g_spi_cmdpacket >> 0) & $FF) ' high byte of 16-bit data

' build 16-bit data (may not be valid though based on command)
g_data16 := (g_data2 << 8) | (g_data)

' now process command to determine what user is client is requested via spi link
case ( g_cmd )

GFX_CMD_NULL:

```

```

' GFX GPU TILE ENGINE COMMANDS //////////////////////////////////////
' catch all commands right here
GPU_GFX_BASE_ID..(GPU_GFX_BASE_ID + GPU_GFX_NUM_COMMANDS - 1):
    ' call single processing function...
    g_spi_result := gfx_ntsc.GPU_GFX_Process_Command( g_cmd, g_data16 )
' // NTSC GFX/TILE SPECIFIC COMMANDS //////////////////////////////////////
' we only expose a subset of the commands the driver supports, you can add/subtract more commands as desired
' some commands like PRINTCHAR for example internally support a number of sub-commands that we don't need to expose
' at this level unless we want to add functionality
GFX_CMD_NTSC_PRINTCHAR:
    ' this command pipes right to the out() function of the driver which supports the following sub-commands already
    '
    ' $00 = clear screen
    ' $01 = home
    ' $08 = backspace
    ' $09 = tab (8 spaces per)
    ' $0A = set X position (X follows)
    ' $0B = set Y position (Y follows)
    ' $0C = set color (color follows)
    ' $0D = return
    ' anything else, prints the character to terminal
    gfx_ntsc.Out_Term( g_data )
GFX_CMD_NTSC_GETX:
    ' return x position in spi buffer, next read will pull it out on last byte of 3-byte packet
    g_spi_result := gfx_ntsc.GetX
GFX_CMD_NTSC_GETY:
    ' return y position in spi buffer, next read will pull it out on last byte of 3-byte packet
    g_spi_result := gfx_ntsc.GetY
GFX_CMD_NTSC_CLS: ' eventhough this command is supported above, we break it out as a seperate SPI command just in case we want to
    ' add functionality on top of, like "overloading" and handling manually, notice we end up calling the $00 sub-command
    ' but we have the flexibility to add stuff if we desire...
    gfx_ntsc.Out_Term( $00 )
    ' add other functionality to the clear screen here...
.
.

```

As you can see the code to extract and rebuild the SPI packets is tricky, a lot of bit fiddling and masking to get the data back from the ASM SPI driver that is passing the data into the global shared memory. All packets are 3-bytes long and look like this:

Command Packet Format

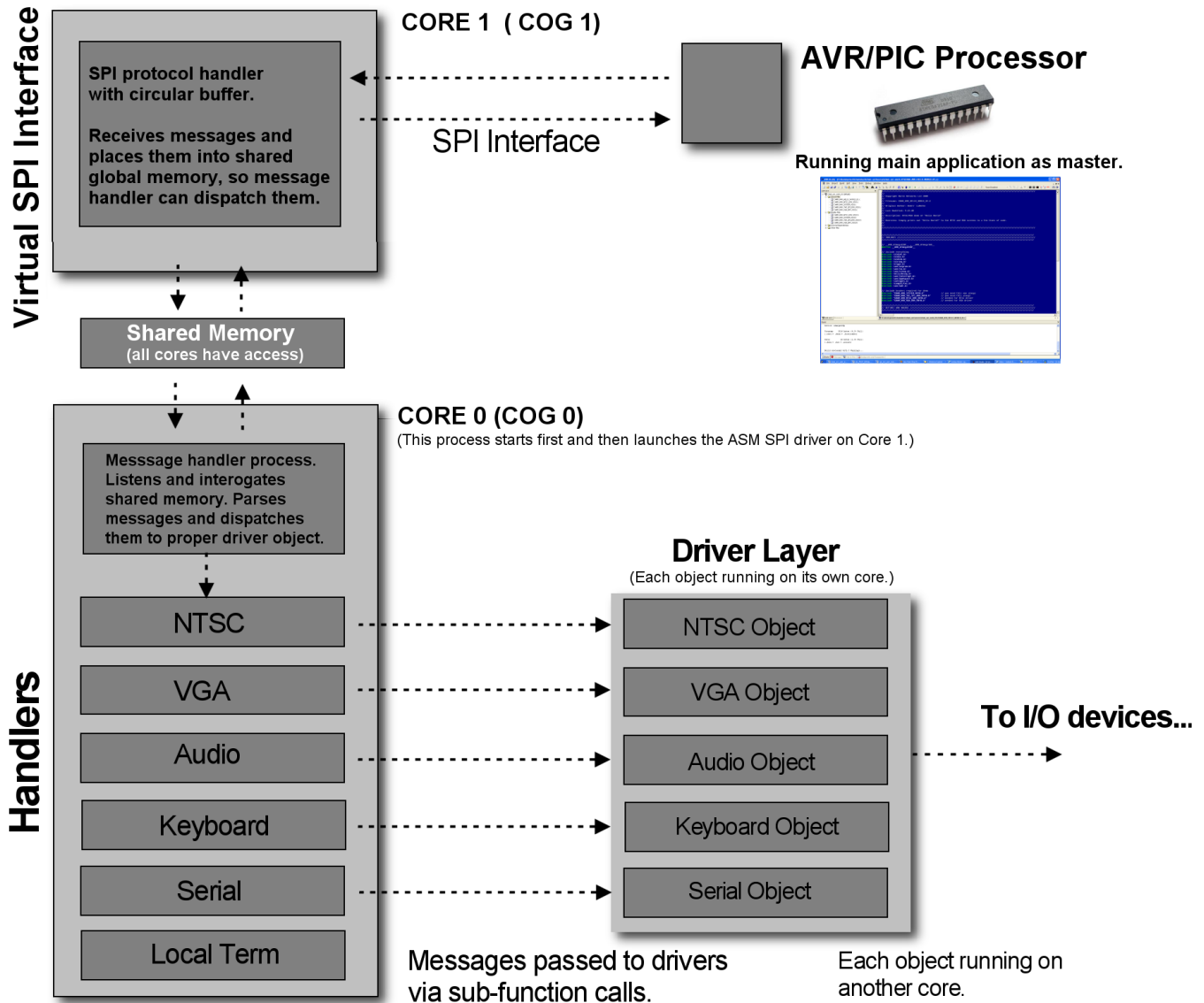
[command8, data_low8, data_high8]

Commands that only require 8-bits of data or operand only use the first data byte, commands that require 2-bytes use both the low and high. The bottom line is all commands only are 3-bytes, so you can't send a string or something large, you have to design your commands so they are simple and can be constructed with only 2 bytes of data at most. Thus, most commands are atomic in nature. For example, there is no "print string" command, there is only a "print character" command, but by stringing a collection of "print chars" together at the master/client side you can print a string. All the commands are designed with this in mind.

16.2 Selecting the Drivers for the Virtual Peripherals

The drivers for each of the media devices were selected based on functionality and popularity. There are definitely better drivers for many of the devices. For example, more robust graphics drivers, or more advanced sound drivers, etc. However, this default driver isn't about using the best, but more about system integration. Thus, drivers that are easy to interface to and easy to control with a simply subset of commands where used. In the sections below we will cover the exact drivers used, but the point is, they were chosen more or less for ease of use and user base.

Figure 16.3 – Another view of the complete system dataflow model.



16.2.1 Complete Data Flow from User to Driver

Now that you have seen all the pieces of the puzzle from a hardware and software point of view, let's review exactly how these pieces all fit together. To begin with the Propeller is running a number of drivers on multiple cores; NTSC graphics terminal, VGA graphics terminal, keyboard/mouse driver, and sound driver. Each one of these drivers takes a single core. Now, by themselves, they don't do much. So the "glue" of the system is the MCP which is pure SPIN program that not only issues commands to the drivers, but is the interface to the AVR via the virtual SPI driver's shared memory.

Thus, the MCP runs on its own core controlling the media cores themselves as well as handing SPI input from the SPI driver itself which is written in ASM running on its own core. With the system in this know state, let's cover exactly what happens in each phase when the system boots.

Initialization – After reset, the MCP software simply loads each of the drivers for NTSC, VGA, keyboard, and serial. Finally, the main **PUB start()** of the MCP is entered and it starts a core with the assembly language SPI driver.

Command Processing Loop – As SPI commands are sent over the SPI channel, the virtual SPI driver listens and if it detects traffic, it then processes the bytes and places them into a globally shared memory buffer. The MCP is sitting a

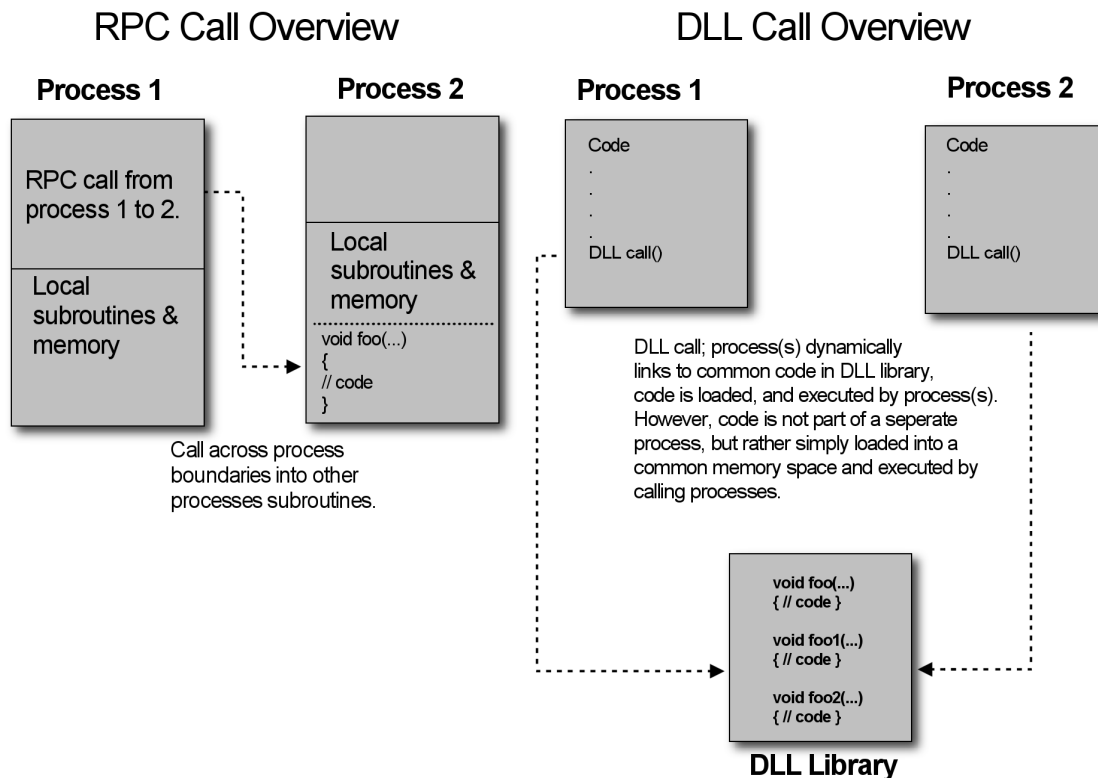
loop "listening" for the SPI driver to place data into this shared memory region. When it detects a packet of information, it then immediately tries to determine if the command portion of the packet is valid.

Command Processing and Execution – The command processor is a handler that interrogates the previously tokenized input command data from the SPI channel and looks for commands in the packet. If a command is found then it continues to process the command pattern and look for the parameters that should follow the command. The parameters (if any) are extracted then the proper "**handler**" is entered. The handler is where the action happens. Each handler is connected to its respective driver and can send and receive messages from that driver. So when a NTSC command is parsed for example, the handler simply calls the NTSC driver and passes the request for execution. When the processing is complete, the MCP loops again and "**waits**" for the next command to process.

16.3 Remote Procedure Call Primer (Theory)

Even if you don't have a degree in Computer Science you have probably used "**remote procedure calls**" or RPCs in one form or another or even invented them unknowingly! The idea of a remote procedure call came about in the 1970's actually, so it's a really old concept. The basic idea is very simple; for one process/program to be able to call/use a subroutine or function in another process/program. More or less a form of interprocess communication.

Figure 16.4 RPC call from process to process in contrast to a DLL call.



RPCs are a little different than using DLLs or libraries since they are passive entities that are loaded on demand. RPCs are more like making calls to another running program and using its resources (subroutines). Thus there is a client server relationship here, and you can think of the RPC call as "**message passing**" as shown in Figure 16.4. There are various forms of the technology and it's more of a concept/methodology than a specific algorithm. For example, in some RPC setups, the RPC call mimics the actual binary foot print of the function call. Assuming the C/C++ programming language, here's a solid example:

```
float DotProduct(float ux, float uy, float uz, float vx, float vy, float vz );
```

Looking at this function, depending on the compiler directives the parameters are passed by value on the stack right to left, so there are 6 floats, each 4-bytes let's assume. Also, there is a return value another 4-byte float, thus we need to "pack" the parameters up into a single record and pass it along and then wait for a single float (4-byte result).

So we might do something like this to "pack" the parameters up into a contiguous memory space:

```
// this is used as transport buffer for RPC calls
CHAR RPC_input_buff[256];

// now pack the parms into the array one at a time
memcpy( RPC_input_buff+=4, @ux, sizeof(float) );
memcpy( RPC_input_buff+=4, @uy, sizeof(float) );
memcpy( RPC_input_buff+=4, @uz, sizeof(float) );
memcpy( RPC_input_buff+=4, @vx, sizeof(float) );
memcpy( RPC_input_buff+=4, @vy, sizeof(float) );
memcpy( RPC_input_buff+=4, @vz, sizeof(float) );

// finally, a call would be made to the "RPC interface"
RPC_Interface( "DotProduct", RPC_input_buff, RPC_output_buff );
```

Now that the parameters are packed into a single data structure, we simply call the RPC interface and pass the starting address of the structure. The RPC interface in this case takes a string as the function name and then two pointers; one to the input parameters and one to where the output results are stored. There is obviously an **"agreement"** and set of conventions between the caller and receiver on this function and how it works, so the client can make RPC calls and the server can respond to them. In this case, the server or other process reads the first string, determines the RPC function then calls a handler with the two pointers. It's up to the handler to **"know"** how to unpack the parameters and generate the results via calling the local function. Thus, RPC calls necessitate a number of extra steps including:

RPC Steps:

(1) Encoding → (2) Transport to server → (3) Decoding → (4) Execution → (5) Encoding → (6) Transport back to client.

Obviously not the fastest thing in the world, however, if the computation workload is 2x or more than all the interface steps then it's worth it or if the local process or machine can't perform the computation, etc. Thus, RPC calls and technology allow a process or machine to use subroutines and resources running in another process or another processor or an entirely different machine.

In our case, we use the concept of RPCs to make calls to another processor from the AVR's SPI interface, thus it's a machine to machine call where each "call" packet is only 3-bytes. But, let's keep exploring the general idea of RPCs and some strategies that might help you later as you update/modify the default drivers shipped with the Chameleon.

16.3.1 ASCII or Binary Encoded RPCs

When designing an RPC system you can make it really complex or really simple. The main idea is that you want to be able to call functions in another process, processor, machine. Decisions have to be made about the **"RPC protocol"** and how you are going to do things. There are no rules for RPC calls unless you are using a Windows, Linux, Sun, etc. machine and want to use one of the official OS's RPC call APIs. When you design your own RPC protocol, it's up to you. In our case, it's tempting to make RPC calls ASCII based and human readable. This of course, eats bandwidth and is slower than binary. However, since it is human readable, the RPC calls take a format that look more like commands rather than strings of bytes representing data.

Thus, one model might be to use ASCII format which would be easy to use and remember for a human. Now, the next step up, would be to still use ASCII formatted data that is human readable, but to make it more abstract. For example, instead of having a command like this:

NTSC Print Hello

We might encode **"NTSC"** as a single number and the command **"Print"** as another number, and then the **"Hello"** would stay as is:

25 0 Hello

As you can see, this version of the ASCII protocol is much smaller, we have saved bytes already! It's still human readable, but not as warm and fuzzy. The entire RPC string is 11 bytes (including NULL terminator). But, can we do better? Sure, if we encode in binary then we don't send the longer ASCII text, we send the actual byte data. Therefore, the binary encoding would look like:

Byte 0, Byte 1, 'H', 'e', 'l', 'l', 'o'

Where byte 0 and 1 would represent the NTSC and Print sub-functions. In this case, the entire RPC call costs 7 bytes, but isn't human readable anymore since byte 0,1 are in binary and could be anything depending on what we picked them to be. The point is that, if you were using another processor or process to make the RPC calls and not a human at a serial terminal then there is no reason to use ASCII coding. However, that said, I still prefer ASCII formats when they are feasible since they are much easier to read and debug. At least sending things in ASCII format during the de-bugging phase, so you can at least look at your data strings on the other end and see what's going on.

16.3.2 Compressing RPC for More Bandwidth

The whole idea of RPC technology is to use it, thus, you might have a program running that makes 100 local calls with 100 RPC calls every time thru the main loop. Thus, you want the RPC transport process to be very quick, hence, compression of the RPC data and or caching is in order. For example, if you are sending large chunks of text or data that has repeating symbols, then its better to compress it locally, transport it, then de-compress and execute since computers are typically 1000's of times if not millions of times faster than the communications links.

Additionally, advanced RPC systems might use the same data over and over. Thus, there is no need to keep sending the data to the server and a caching system should be employed where on the first RPC call, the caller indicates that a data structure being passed is static and cacheable. Thus, the server caches it after its first use, then on subsequent calls, the client need not send the structure until it needs refreshing. For example, say you have a 3D data base that you want a RPC to perform calculations on. The database never changes, so no need to keep sending it over and over, once its in the servers memory space, you can save the bandwidth.

16.3.3 Our Simplified RPC Strategy

Considering all these interesting methods, the method used for the Chameleon is a 3-byte binary encoded SPI packets with the following format:

[command8, data_low8, data_high8]

Where command8 is an 8-bit command code, data_low8 and data_high8 are the operands for the command. Thus, each SPI packet is rather small and can't do much. However, with proper design you can use these small SPI command packets to create larger commands. For example, say you want a command to print a character to the VGA screen? Well, that's easy enough – you can use a specific command code for "print to VGA" then a single one of the 8-bit data words can hold the 8-bit character.

But, what if you wrote a sound driver that plays .wav files and you want to send down 1000 bytes? Well, that's easy, you just have to break the process into to "states" or steps and then create a command for each state. For example, you could create a "memory write" command that consists of 3 sub-commands:

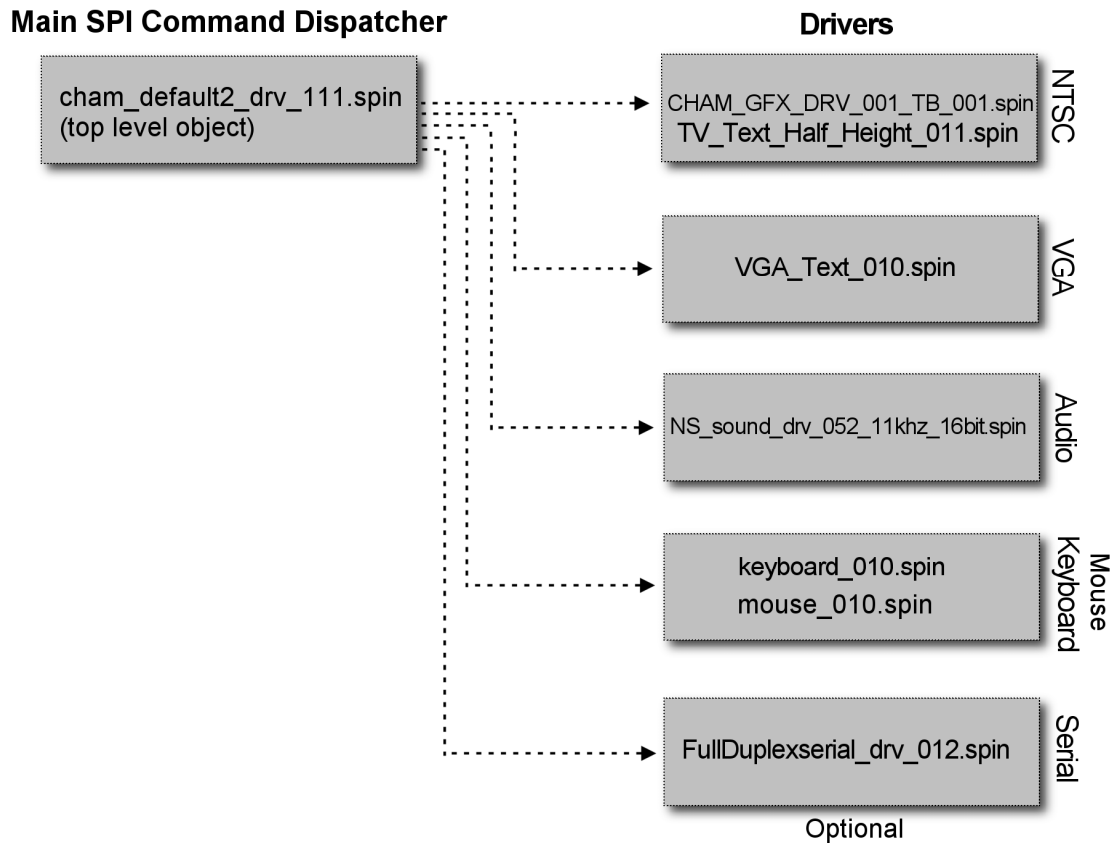
Command 1: Set memory pointer addr.

Command 2: Write memory mem[addr] with x.

So Command 1 needs a 16-bit address, you can fit that in the two operand bytes, and command 2 needs only a single 8 of data! Thus, you can write 1000 bytes from addresses 2000 to 2999 with this simple algorithm:

```
for (mem_addr = 2000; mem_addr < 2999; mem_addr++)
{
    send_spi_command(COMMAND1, mem_addr & 0xFF, mem_addr >> 8)
    send_spi_command(COMMAND2, data, 0);
} // end for
```

As you can see, the 3-byte SPI command packet isn't a limitation. In fact, most SPI (and I²C) devices use 2 byte command rather than 3, thus even more steps have to be performed for larger more complex operations.

Figure 16.5 - The virtual drivers used for the project.

16.4 Virtual Peripheral Driver Overview

There isn't much to say about the drivers used on the project other than I went to the Parallax Object Exchange located here:

<http://obex.parallax.com/objects/>

and hunted around for appropriate objects to use for this project based on my experience with developing objects and using them. The objects aren't the fastest, the coolest, or the best necessarily they just work and get the job done and in most cases are the reference objects developed by Parallax initially with small changes by myself and other authors. The idea was to have the NTSC, VGA, audio and keyboard/mouse all running at the same time and be able to access these devices. In the future, you might want to use other objects or improve these for more specific needs. In any event, referring to Figure 16.5. The objects used are shown in Table 16.2.

Table 16.2 - Objects used for the Chameleon default MCP drivers.

Function	Version	Top object file name ⁽¹⁾
MCP (master control program)	1.11	CHAM_DEFAULT2_DRV112.spin ⁽²⁾
NTSC	1.1	CHAM_GFX_DRV_001_TB_001.spin ⁽²⁾ TV_Text_Half_Height_011.spin ⁽²⁾
VGA	1.0	VGA_Text_010.spin
Audio	5.2	NS_sound_drv_052_11khz_16bit.spin
Serial	1.2	FullDuplexSerial_drv_012.spin
PS/2 Keyboard	1.0	keyboard_010.spin ⁽³⁾
PS/2 Mouse	1.0	mouse_010.spin ⁽³⁾
Note 1: Many of the drivers include other sub-objects as well. Note 2: There are two versions the MCP driver, one is called CHAM_DEFAULT2_DRV_V112.spin this is used for all our examples. However, there is a slightly modified version called CHAM_DEFAULT1_DRV_V112.spin that uses the plain vanilla NTSC terminal that might work better on some LCDs. You can drop down to it if you have trouble with the default2 driver. Note 3: The Chameleon only has one PS/2 port, so only one driver; keyboard or mouse can be active at once. However, the default MCP driver can "hot" swap the drivers, so you can unplug the keyboard/mouse in real-time and with software "start" the new device.		

All of the drivers and their sub-objects are included in the source directory for this chapter located on DVD-ROM in:

DVD-ROM : \ CHAM_AVR \ SOURCE \ PROPELLER_DRIVER \ *.*

If you look on the Parallax Object Exchange you should be able to find most of these drivers as well; however, we will use the ones from my chapter and my sources since I made slight modifications to each of them to make things easier.

16.4.1 Normalization of Drivers for Common RPC Calls in Future

The last thing I want to discuss about the drivers in the interfaces to all of them. Since this is a pieced together system of other people's drivers, each driver obviously has its own methodology and API. For example, the NTSC calls look entirely different from the keyboard calls and so forth. Alas, if you were to develop a system from the ground up and design drivers for NTSC, VGA, keyboard, etc. you would be wise to design all the APIs in a similar fashion with conventions for function calls, inputs and outputs, so that technologies like RPC calls and others could more easily be implemented and optimized better.

17.0 Chameleon AVR API Overview

In this section of the manual we are going to discuss the Chameleon AVR API and its related components. First and foremost, I want to make it clear that the API we have developed is by no means complete, the best, the fastest, etc. It's just a set of source files and functions that get you started developing applications. Moreover, since the whole idea of the Chameleon is to leverage the functionality of the drivers running on the Propeller chip. The AVR side API is nothing more than "wrapper" functions that container a number of SPI commands, so you don't have to type a lot to get things done. In other words, you don't have to use these libraries in most cases, you can just send command directly if you wish. Moreover, these API functions are designed for the specific drivers running on the default Propeller drivers. If you change or modify drivers on the Propeller side chances are these new drivers will have different functionality and you will want to re-write a new high level "wrapper" API for the driver in question.

However, I hope that you use them as starting points only to develop your **own** API, functions, and drivers that are much more optimized and complete. On the other hand, I have spent a good deal of time trying to develop a base set of API libraries to get you started, so that you can do everything from graphics and sound to UART communications with the PC. Table 17.1 lists all the API library modules we have developed for you to jumpstart your development and exploration of the Chameleon AVR and the AVR processor itself.

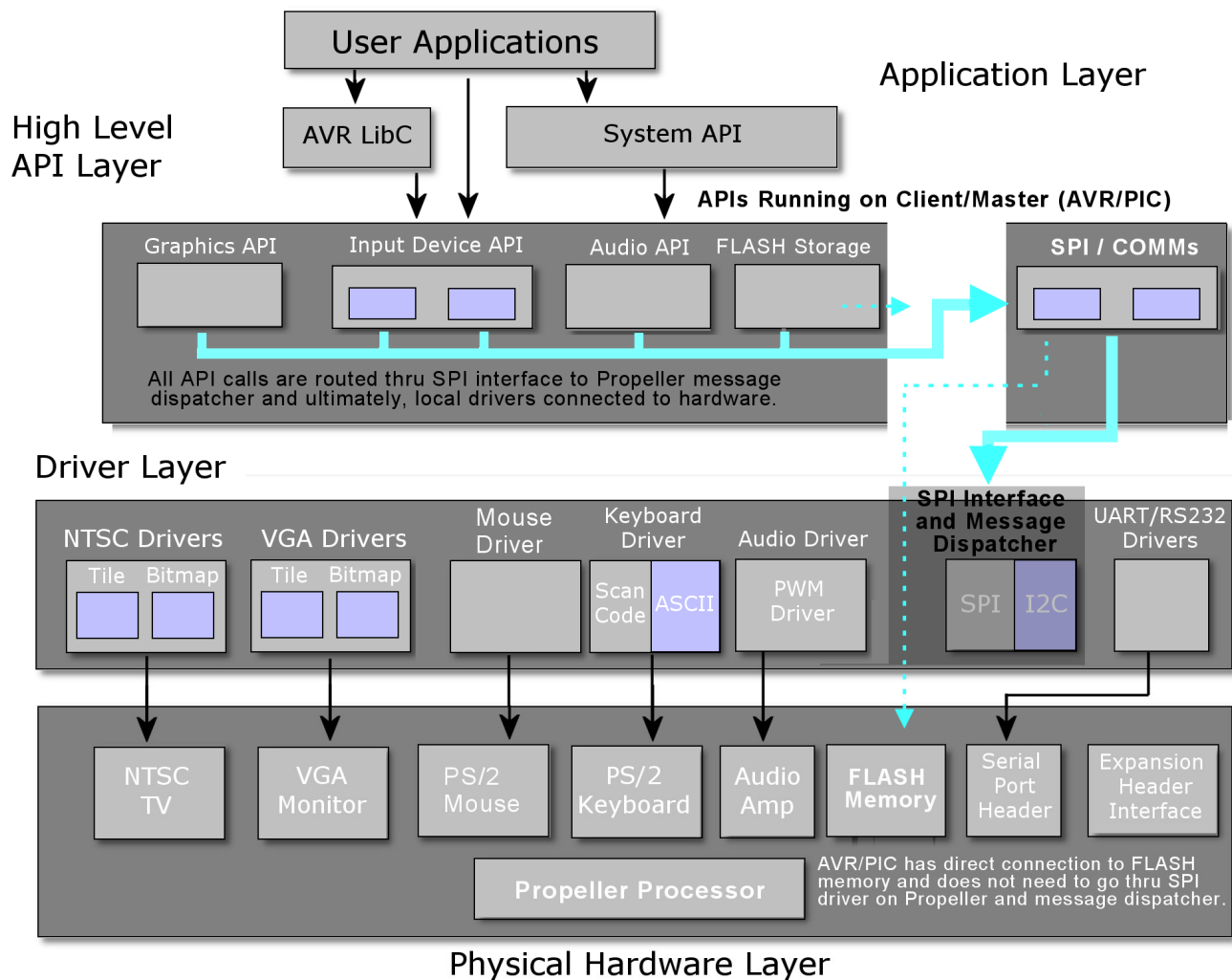
Table 17.1 – The Chameleon AVR source level library modules at a glance.

Library Module Name	Description
System Level Library Module	
CHAM_AVR_SYSTEM_V010.c h	System level utility functions and configuration.
High Level Graphics Module	
CHAM_AVR_GFX_DRV_V010.c h	Functions to take advantage of the graphics tile engine that runs under the "default2" drivers.
NTSC Text and Tile Graphics Module	
CHAM_AVR_NTSC_DRV_V010.c h	Tile engine and text console/terminal functions.
VGA Text and Tile Graphics Module	
CHAM_AVR_VGA_DRV_V010.c h	Tile engine and text console/terminal functions.
Audio Driver Module	
CHAM_AVR_SOUND_DRV_V010.c h	Sound driver functions.
Communications Modules	
CHAM_AVR_UART_DRV_V010.c h	Low level Serial RS-232 communications driver.
CHAM_AVR_TWI_SPI_DRV_V010.c h	Low level I2C and SPI peripheral driver.
Input/Output Devices Modules	
CHAM_AVR_GAMEPAD_DRV_V010.c h	Gamepad driver.
CHAM_AVR_KEYBOARD_DRV_V010.c h	PS/2 keyboard driver.
CHAM_PROP_PORT_DRV_V010.c h	Propeller "local" 8-bit I/O driver.
Storage Device Modules	
CHAM_AVR_FLASH_DRV_V010.c h	1MB SPI FLASH memory chip driver.
Mechatronics, Actuator and Sensor Modules	
N/A	
Note 1: This mode uses 6x8 tiles to squeeze more characters per line for text modes.	

When building application using the Chameleon AVR library you will typically include the System module always and then include various modules such as video, keyboard, UART, etc. as needed by your application. All the C, and Header files are located on the DVD here for reference:

DVD-ROM : \CHAM_AVR \SOURCE *.*

In the section below we will briefly cover each one of these libraries and the functions within them. Before, we get started let's take a look at the "**big picture**" illustrating all the components of the Chameleon AVR development suite of software and tools. Figure 17.1 shows a system level architecture diagram of all the pieces involved.

Figure 17.1 – System level architecture of the Chameleon API and other library modules.

Referring to the figure there are a lot of components to discuss. First off, the primary library that all C/C++ programs use is the open source “**AVR Libc**” library that is part of the WinAVR install and is what the GNU GCC compiler uses as the base for all of the standard C library functions you are used to. Therefore, when you create a C/C++ program for the Chameleon AVR and include say `<stdio.h>` you are including functions from the AVR Libc installation. Let’s talk about this for a moment. First, embedded systems are very unique in that they are very constrained, have little memory and resources. So any function that you use from the standard C library Libc emulates what you might expect on a PC platform, but in many cases the behavior might not be as expected. Thus, you need to take care when using standard library functions and realize that they may have side effects or limitations that VC++ or GNU C on your PC do not. Please refer to the AVR Libc manual located here on the DVD-ROM for a complete overview of the library:

DVD-ROM:\ CHAM_AVR \ DOCS \ avr-libc-user-manual-1.4.6.pdf

Additionally, a great resource for the library is the developer’s website itself:

<http://www.nongnu.org/avr-libc/>

TIP

The Arduino toolchain uses the same GNU GCC compiler and AVRLibc library, thus everything related to C/C++ programming with AVRStudio applies to the Arduino tool as well where it relates to GCC and WinAVR/Libc.

There are lots of tricks and tips to using the AVR Libc functionality on there. However, for the most part as long as you don’t push it, you can use standard C functions and everything will work out. Of course, you have to have common sense when using functions. For example, on a PC the function ***printf(...)*** makes sense since there is some kind of terminal

device. On the Chameleon AVR, this won't work since the developer of AVR Libc has no idea about the Chameleon AVR, thus anything that is hardware specific won't work since Libc has no idea about the hardware. On the other hand, you might ask "is it possible to get **printf(...)** to work?". The answer is yes. However, you would have to go into the source files where **printf(...)** is located and modify it to do "something" intelligent on the Chameleon AVR such as print to the UART, or to the NTSC/VGA screen. But, you get the idea. Don't use standard C functions that make no sense on an embedded system.

Next, embedded systems have multiple memory types; **RAM, FLASH, EEPROM**. And the AVR Libc library knows this. Thus, many functions only work in RAM or FLASH specifically. Therefore, you always have to keep in mind that you are working with a **Harvard** memory architecture and a separate RAM/FLASH memory model. The AVR Libc documentation has a lot to say about this subject, so make sure to read it.

Alright, so that's the story with AVR libc, it's part of the GNU GCC WinAVR installation and when you build a C/C++ application functions are pulled in from this library. To give you an idea of what Libc supports, Table 17.2 lists the header files you can include in your Chameleon AVR applications and what functionality they support.

Table 17.2 – AVR Libc library functionality listed by header file.

Libc Header File	Description
General C/C++ Library Functions	
<alloca.h>	Allocate space in the stack.
<assert.h>	Diagnostics.
<ctype.h>	Character Operations.
<errno.h>	System Errors.
<inttypes.h>	Integer Type conversions.
<math.h>	Mathematics.
<setjmp.h>	Non-local goto.
<stdint.h>	Standard Integer Types.
<stdio.h>	Standard I/O facilities.
<stdlib.h>	General utilities.
<string.h>	Strings.
AVR Specific Library Functions	
<avr/boot.h>	Boot loader Support Utilities.
<avr/eeprom.h>	EEPROM handling.
<avr/fuse.h>	Fuse Support.
<avr/interrupt.h>	Interrupts.
<avr/io.h>	AVR device-specific IO definitions.
<avr/lock.h>	Lockbit Support.
<avr/pgmspace.h>	Program Space Utilities.
<avr/power.h>	Power Reduction Management.
<avr/sfr_defs.h>	Special function registers.
<avr/sleep.h>	Power Management and Sleep Modes.
<avr/version.h>	avr-libc version macros.
<avr/wdt.h>	Watchdog timer handling.
AVR Specific Utility Functions	
<util/atomic.h>	Atomically and Non-Atomically Executed Code Blocks.
<util/crc16.h>	CRC Computations.
<util/delay.h>	Convenience functions for busy-wait delay loops.
<util/delay_basic.h>	Basic busy-wait delay loops.
<util/parity.h>	Parity bit generation.
<util/setbaud.h>	Helper macros for baud rate calculations.
<util/twi.h>	TWI bit mask definitions.
<compat/deprecated.h>	Deprecated items.
<compat/ina90.h>	Compatibility with IAR EWB 3.x.

As you can see the first set of headers looks pretty familiar, but the second and third set are totally AVR specific, but not hardware specific to the Chameleon AVR, only to the AVR 328p processor.

Now, the **hardware specific** functionality for the Chameleon AVR, we had to develop. Referring to Figure 17.1, there are a number of library classes you can glean from the figure, they are:

System	The system library module is a "glue" module that ties the other modules together and is used as a common place to put functions and constants that don't fit into any other category. Currently, the system library module is very small, but it will grow in the future.
Graphics	This library simply makes calls to the graphics drivers running on the Propeller chip. So whatever they do, this library tries to expose to the user. The current drivers are "tile" graphics only and support text, printing, scrolling, and crude control of color. The default2 driver series adds some functionality with a "gaming" tile engine that supports fine vertical scrolling, control of overscan colors, large playfield tilemaps, re-definable character bitmaps (4 colors per tile), and many other cool features.
Sound	This library interfaces to the Propeller sound driver object and exposes limited capabilities of the driver itself (which can do a lot more). The API gives you a few functions to play sounds, control volume, etc. But, you will probably want to enhance this API.
Keyboard/Mouse	There is a PS/2 port on the Chameleon which is connected to the Propeller. The default driver on the Propeller has drivers for both keyboard and mouse support. Thus, we have developed a simple API to communicate to both the keyboard and mouse driver running on the Propeller (one at a time of course).
UART	The AVR 328p processor comes with two hardware UART's, but we have developed a library of functions that abstracts the functionality, so you can perform buffered I/O very easily with interrupt driven drivers.
SPI	The AVR 328p has built in S erial P eripheral I nterface hardware as well, but there is quite a bit to set it up and communicate with it, thus we have created a nice abstraction layer to initialize it, read and write bytes.
I²C	The AVR 328p again has built in hardware for Inter-Integrated Circuit communications, but the hardware is tricky to setup and the I2C protocol is a bit difficult to work with, so once again, we have developed a software layer on top of it for your convenience.
FLASH	The Chameleon AVR has a 1MB SPI FLASH memory on-board which can be used for storage of assets, data, code, whatever you wish. This driver API runs 100% on the client AVR chip and gives you the ability to read and write sectors/pages in the FLASH memory, erase, etc. Considering that, you might want to write some software to abstract the FLASH memory into a FAT16 like device? All you need are functions to read/write sectors/pages (which we provide) then you can use a 3 rd party library to add support so the FLASH memory "feels" like a FAT16 drive. There are numerous open source drivers for SD cards and FAT16, so you are free to use them in your development. For example the Procyon AVR Library that has FAT16 support, but for read only. Simple google a bit and you can find other libraries or write one yourself.
Mechatronics	N/A

NOTE

Some modules like the SPI and I²C are actually in the same source file since they are so similar.

These compose the main library modules that make up the Chameleon AVR specific API functions (mostly wrapper functions). Now, before we continue, I want to make a point about language here. I have been using the word "**library**" in a cavalier way. Library as it relates to C/C++ programming is a pre-compiled container with many binary objects. For example, AVR Libc is a "**library**", hundreds of functions have been compiled into a single library file and that single file is linked against during builds with GNC GCC. In our case, we could have built an official "**library**" and put all the source

modules into it, but this would be tedious since it would hide what you are including in each project. Thus, I have decided to keep the "library" in a source format as a collection of .C and .H files, so we can see what's going on, include what we need, and when we update one of the source files we don't have to rebuild the library, we simply include the source file in the source tree with our main project file.

With that in mind, there are a number of .C and .H files that make each up class of library functionality for the Chameleon API. At this point, we are going to drill down and list each of the files with a short description. Then we are going to discuss the key data structures and header files for each library module, list the functions one by one with examples.

Finally, all source files can be found on the DVD at location:

DVD-ROM:\CHAM_AVR\SOURCE*.*

Of course, you should have already copied this to your hard drive into your project's working directory. To save space, we are not going to list the contents of each source file since that would take hundreds of pages; however, key elements will be listed for reference. Nonetheless, you should have your editor pointed to the source directory, so you can review the complete contents of each file as we discuss them.

17.1 System Library Module

The "**System**" library module consists of a single C source file and its header. The "System" modules must be included in every Chameleon AVR program since its contains some top level #defines, types, and other pertinent elements that other API modules depend on. Additionally, the C source file contains a few utility functions that will continue to grow in the future.

The "System" source files are named below:

CHAM_AVR_SYSTEM_V010.c	Main C file source for "System" module.
CHAM_AVR_SYSTEM_V010.h	Header file for "System" module.

You should include the .h file with all your applications and add the .C file to your source tree for all applications as well. Next, we will take a look at some excerpts from the header file, review key elements, and any data structures that are of interest.

15.1.1 Header File Contents Overview

The header files for many of the API modules are rather large and we won't have time to list their contents out in their entirety; however, the "System" header is quite manageable, so we are going to take a look inside. To begin with, the header contains some useful macros:

```
// macros to manipulate bits
#define SET_BIT(value, bit) value = ( (value) | (1 << (bit)) )
#define SET(bit) (1 << (bit))
#define RESET_BIT(value, bit) value = ((value) & ~(1 << (bit)))
#define RESET(bit) ~(1 << (bit))
#define WRITE_BIT(value, bit_num, bit_val) value = (((value) & ~(1 << (bit_num))) | ( (bit_val) << (bit_num) ))
#define SETPORTBITS(b7, b6, b5, b4, b3, b2, b1, b0) ( (b7 << 7) | (b6 << 6) | (b5 << 5) | (b4 << 4) | (b3 << 3) | (b2 << 2) | (b1 << 1) | (b0 << 0) )

// conversion macros
#define HEX_TO_DEC(n) (((n >= '0') & (n <= '9')) ? (n - '0') : (((n >= 'A') & (n <= 'F')) ? (n - 'A' + 10) : 0) )

// a more useful random function
#define RAND_RANGE(x,y) ( (x) + (rand()%((y)-(x)+1)) )

// AVR RESET MACROS ////////////////////////////////////////

#define AVR_Soft_Reset() \
do \
{ \
    wdt_enable(WDTO_15MS); \
    for(;;) \
    { \
        ; \
    } \
} while(0)
```

These macros are primarily used to manipulate port bit I/O, set bits, reset bits, and write to the ports in a clean way. Finally, there is a convenient conversion from hex to decimal used by some of the conversion functions in the API.

Next, are the #defines and one very important definition is the processor speed:

```
// define CPU frequency in MHz here if not defined in Makefile
#define F_CPU 16000000UL
```

This constant is special to many of the C libraries and it used as a compile time constant. You might recall us setting this constant in the compiler build configuration setup. However, we are going to override that constant here (making it the same), as extra insurance to make sure that manually set this constant and then use it throughout the code base to set up timing calculations.

Although the Libc libraries have many standard types, I prefer to always create a few of my own that are compiler independent, so I know they work if I port the code, with this in mind, the “System” module has a few types it declares:

```
// basic unsigned types
typedef unsigned short USHORT;
typedef unsigned short WORD;
typedef unsigned char BYTE;
typedef unsigned char UCHAR;
typedef unsigned long QUAD;
typedef unsigned int UINT;

// generic input device data record type, helps get back data from various multitdata calls
typedef struct gid_event_type
{
    int x,y,z;    // position data
    int buttons;  // buttons, bit encoded
                  // bit4 = right-side button
                  // bit3 = left-side button
                  // bit2 = center/scrollwheel button
                  // bit1 = right button
                  // bit0 = left button

    } gid_event, *gid_event_ptr;
```

Lastly, so that the user can change the system frequency if he wishes during run-time, there is a variable shadow of the **F_CPU** constant that is loaded with said constant during startup. Thus, the variable is used in the Chameleon API library for calculations. This gives it typing information and as noted the ability to be changed during run-time, here’s the declaration:

```
extern unsigned long f_cpu;
```

That wraps it up for the header file declarations. The remainder of the header file is simply the prototype listing for the functions themselves which we will see in the next section.

17.1.2 API Listing Reference

The API listing for the “System” module **CHAM_AVR_SYSTEM_V010.c** is quite short. As said, this C file is a placeholder for future functionality that relates to system level housekeeping and utility functions, so at this early stage there isn’t much here. Considering that, Table 15.3 below lists the “System” API functions categorized by functionality.

Table 15.3 – “System” module API functions listing.

Function Name	Description
Time and Counting	
void Delay_Clocks(long count);	Delays for the requesting number of processor clocks (use _delay_us() and _delay_ms() for time based delays, these are out of AVRLibc).
void _delay_s(int sec);	Delays for the requested number of seconds.
Conversion	
long atoi2(char *string);	Converts binary, decimal, and hex formatted strings to integer.

System Reset	
int Reset_Master(void);	Resets the AVR chip.
int Reset_Prop(void);	Resets the Propeller chip remotely.
Note: The time functions are currently approximations and need updating to be more accurate. Please use AVRLibc functions instead.	

17.1.3 API Functional Declarations

The following lists each function, a description, comments, and example usage of the function.

Function Prototype:

```
void Delay_Clocks(long count);
```

Description: *Delay_Clocks(...)* waits the specified number of processor clocks then returns. The AVR Libc library has similar timing functions as well. **Note:** it is recommended AVR Libc functions are used instead.

Example(s): Wait for 1000 clocks.

```
Delay_Clocks( 1000 );
```

Function Prototype:

```
void _delay_s(int sec);
```

Description: *_delay_s(...)* waits the specified number of processor clocks then returns. The AVR Libc library has similar timing functions as well. **Note:** it is recommended Libc functions are used instead.

Example(s): Wait 60 seconds.

```
_delay_s( 60 );
```

Function Prototype:

```
long atoi2(char *string);
```

Description: *atoi2(...)* has a built in parser that can parse strings in decimal, hexadecimal, and binary formats. The sentinels \$, and % are used for hex and binary representations with omission of sentinel meaning decimal conversion. This function is convenient for text input and conversion algorithms, so users can use any of the popular numeric bases as their inputs. The function returns the value as a **long** thus all inputs have to fit into a signed **long** result. Returns 0 if the function can't perform a conversion.

Example(s): Conversion examples from all bases.

```
char *decimal_string = "12345";
char *hex_string     = "$FFFF"; // 65536
char *binary_string  = "101";   // 5

long result;

// decimal conversion
result = atoi2( decimal_string );

// hex conversion
```

```
result = atoi2( hex_string );
// binary conversion
result = atoi2( binary_string );
```

Function Prototype:

```
int Reset_Master(void);
```

Description: **Reset_Master(...)** causes a soft reset on the AVR chip, basically re-booting it. Always returns 1.

Example(s): Reset the AVR.

```
Reset_Master();
```

Function Prototype:

```
int Reset_Prop(void);
```

Description: **Reset_Prop(...)** causes a soft reset on the Propeller chip, basically re-booting it. Always returns 1.

Example(s): Reset the Propeller, then the AVR.

```
// reset the Propeller
Reset_Prop();

// wait a moment...
_delay_ms(1000);

// now reset the AVR
Reset_Master();
```

18.0 UART and RS-232 Library Module Primer

The “**UART and RS-232**” communications module supports basic transmission and reception of bytes to and from the Chameleon AVR serial port. The AVR 328P (“**P**” – Pico power version that we are using) has a single very advanced **USART** (Universal Synchronous and Asynchronous Serial Receiver and Transmitters). It is connected directly to the TX/RX pins of the USB FTDI chip which allows serial communication via the USB port. Additionally, the TX/RX lines are exported on the I/O headers.

Before discuss the overall architecture of the UART library module, here are the main features of the AVR 328P USART:

- Full Duplex Operation (Independent Serial Receive and Transmit Registers).
- Asynchronous or Synchronous Operation.
- Master or Slave Clocked Synchronous Operation.
- High Resolution Baud Rate Generator.
- Supports Serial Frames with 5, 6, 7, 8, or 9 Data Bits and 1 or 2 Stop Bits.
- Odd or Even Parity Generation and Parity Check Supported by Hardware.
- Data Overrun Detection.
- Framing Error Detection.
- Noise Filtering Includes False Start Bit Detection and Digital Low Pass Filter.
- Three Separate Interrupts on TX Complete, TX Data Register Empty and RX Complete.
- Multi-processor Communication Mode.
- Double Speed Asynchronous Communication Mode.

As you can see they are pretty powerful and the library API we provide here only scratches the surface of what you can do with the USART hardware. They can even be used to stream audio or video data if you are really creative since they are more or less interrupt driven high speed shifting devices. In any event, let's talk about some preliminary materials

before we begin. You should review the data sheet on the AVR 328p, especially the USART. The data sheets is located here on the DVD:

DVD-ROM:\ CHAM_AVR \ DOCS \ DATASHEETS \ ATmega48_88_168_328_doc8161.pdf

And here are some application notes specifically on using the USART:

DVD-ROM:\ CHAM_AVR \ DOCS \ WHITEPAPERS \ avr_uart_doc2547.pdf

DVD-ROM:\ CHAM_AVR \ DOCS \ WHITEPAPERS \ avr_usart_spi_doc2577.pdf

Finally, the "UART and RS-232" API libraries are contained in the following files:

CHAM_AVR_UART_DRV_V010.c
CHAM_AVR_UART_DRV_V010.h

- Main C file source for "UART and RS-232" module.
 - Header file for "UART and RS-232" module.

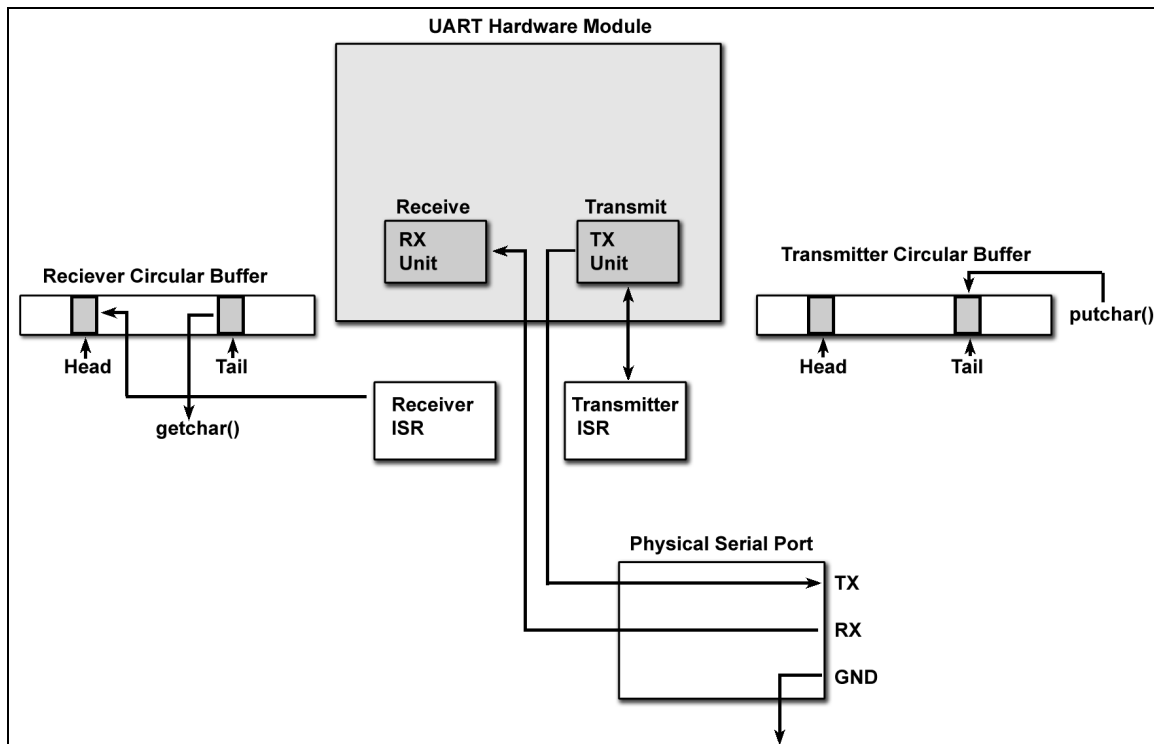
ARDUINO TIP

The Arduino library has some built in C++ classes for serial I/O, thus when using the Chameleon with the Arduino tools, you do **not** need to use the UART library, you can use the built in library the Arduino libraries provide.

18.1 The Architecture of the UART API Library and Support Functionality

The UART library is relatively complex in as much as to save you a lot of grief trying to program the AVR USARTs, we wanted to give you some ready to go software and API functionality, so you could perform basic serial I/O and have some fun. To that end, we developed a buffered, interrupt driven UART communications system with some moderately powerful support functions to save you time. Let's take some time to briefly discuss the overall architecture of the system, so you know how it's all put together. This will help understand how the API functions fit together and the data flow of the system.

Figure 28.1 – The system architecture of the UART driver.



Hopefully, you have read thru the application notes and data sheet on the AVR USARTs. Nonetheless, serial communications at the PC level is usually quite easy. The PC has an API, hardware, interrupts, and lots of sub-systems,

so that you are insulated from what's really going on. This is not true with microcontrollers. The AVR 328p series has hardware USARTs (we are using for UARTs) to take the load off the bit banging, polling, and transmission and reception of actual bytes, but we still need to write a lot of software on top of the hardware to develop an API that is useful.

There are lots of ways to approach this; for example, if you are happy with waiting for characters you could write a receive function that simply waits for the reception buffer to get a character, this is slow though since you have to poll for it. Secondly, you could do the same with your transmission software and write a loop that constantly writes bytes to the transmit buffer while waiting for the bytes to transmit before writing another. These methods are fine, but I wanted to give you a little bit more to work with as a starting point, so you can see how you might develop a more robust serial communications system. Referring to Figure 18.1, this is the architecture of the UART module we have developed. The UART driver API is an interrupt driven system with circular buffers on both the incoming and outgoing characters. The system is easy to understand from a design perspective; interrupts are setup to fire when both a character has been received and when a character has been transmitted. These interrupts both call "**interrupt service routines**" (ISRs) that have been developed to support "**circular buffers**". These circular buffers allow the calling functions to place characters in the outgoing buffer and queued for transmission as well as for characters to fill up the receiving circular buffer until the caller wants to read characters.

The idea of this being the functions that transmit and receive characters do not have to wait, they make to functions that interrogate the circular buffers and either insert or remove characters. This way the system is responsive and only if the buffers are overflowed is data lost. For example, say you want to transmit the string "**Hello World!**". Normally, you would have to transmit the "**H**", then wait, "**e**", then wait, etc. But, with interrupt driven architecture and the buffers, the string is simply copied into the circular output buffer and the interrupt sends the string out for you.

The receiver is the same, you can interrogate the circular buffer anytime you wish. For example, say that you know the transmitter on the other end is going to send a single string "**The World is Fine.**" back to you. This string fits into the circular buffer, so you don't have to interrogate the buffer immediately and worry about loss of data. You can finish your activities, maybe a disk I/O or some rendering then at your leisure call the receive function which will check the buffer for data and send it to you.

Referring to Figure 18.1 once again, you can see some of the details of the system. There are two functions similar to the C `putc(...)` and `getc(...)` called `UART_putc(...)` and `UART_getc(...)` these functions directly access the transmit and receive buffers for you. Thus, from a software perspective all the work of polling the transmission and reception hardware is taken care of for you and all you have to do is use these two functions (and pay attention to their return values)!

ARDUINO TIP

The Arduino uses similar serial methods from the "serial" class named `serial.begin(...)`, `serial.print(...)`, `serial.read()`, etc. learn about them in the Arduino documentation.

So the idea of our UART module is that it loads up the ISRs for communication, set the USART hardware up (speed, bits per character, parity, etc.) and then you communicate via a collection of functions. Now, although we can use the put and get class functions, we have developed a number of other functions that help print strings, numbers, as well as directly access the USART transmit buffer, so you can mix and match your serial library use as you wish or write your own.

18.2 Header File Contents Overview

The "UART and RS-232" module header `CHAM_UART_DRV_V010.h` has a few globals as well as the #defines, so at least there is something interesting to talk about! Let's begin with the #defines:

```
/* default baud rate */
#define UART_DEFAULT_BAUDRATE    115200 // slow down to 57600, 38400 if you have trouble

// transmit and receive buffer constants
#define UART_TX_BUFF_SIZE        32
#define UART_RX_BUFF_SIZE        32
```

The #defines are trivial; simply a constant for a convenient baud rate as well as the size of the circular buffers. You can make these any size you wish, but memory is at a premium, so keep it in mind. If you are going to be using a lot of serial communications and know you are going to send large strings or receive large strings and not be able to interrogate the

buffer functions then you might want to increase the buffers to 64, 128, 256 etc. But, for now 32/32 seems to work find in all examples.

Next are all the global data structures and variables used by the API functions:

```
// global that holds UART current baud rate
extern int g_baudrate;

// receiver and transmitter head and tail indices for circular buffers
extern int g_uart_rx_head;
extern int g_uart_rx_tail;
extern int g_uart_tx_head;
extern int g_uart_tx_tail;

// buffers for uart interrupt driver
extern UCHAR g_uart_buffer_rx[UART_RX_BUFF_SIZE]; // receiver buffer for incoming characters
extern UCHAR g_uart_buffer_tx[UART_RX_BUFF_SIZE]; // transmit buffer for outgoing characters
```

First we have a global baud rate, pointers to the circular buffers, and finally the buffers themselves. These are all declared in the C file of course, these are simply the external references.

18.3 API Listing Reference

The API listing for the "UART and RS-232" module **CHAM_UART_DRV_V010.c** is listed in Table 18.2 categorized by functionality.

Table 18.2 – "UART and RS-232" module API functions listing.

Function Name	Description
Initialization	
int UART_Init(unsigned int baudrate);	Initializes UART, ISRs, etc.
int UART_Shutdown(void);	Shuts down the UART.
Transmission	
int UART_putc(unsigned char ch);	Transmits a character using interrupt buffer.
void UART_Transmit(unsigned char ch);	Transmits a single character immediately.
void UART_Newline(void);	Sends an ASCII newline - CR/LF.
void UART_Print_String(char *string);	Prints a NULL terminated string.
void UART_Print_Int(int num);	Prints a signed integer.
void Terminal_Clear_Screen(void);	Clears the VT100 terminal with ANSI code for clearsreen.
Reception	
int UART_getc(void);	Receives a character thru interrupt buffer.
int UART_Receive(int wait);	Waits for incoming character in receive buffer directly

18.4 API Functional Declarations

The following lists each function, a description, comments, and example usage of the function.

Function Prototype:

```
int UART_Init(unsigned int baudrate);
```

Description: **UART_Init(...)** initializes **USART0** (the primary USART connected to the **SER1** serial port). The

parameter is the baud rate in bits per second and should be one of the following values; 1200, 2400, 4800, 9600, 19200, 38400, 57600...115200. If you send any other values then a mathematical computation is performed to estimate the baud rate register settings and its not very accurate. Additionally, the initialization function enables serial settings of "N81", that is, **no** parity, **8**-data bits, **1**-stop bit. You must set your PC or serial device to the same settings as well. Finally, the initialization function enables the interrupt handlers which are hidden in the C source file. This function must be called before any other API functions. The function returns 1 always.

Example(s): Initialize the serial communication system to a baudrate of 115200 with setting N81.

```
// function always sets serial to N81, so just need the baudrate to max
UART_Init( 115200 );
```

Function Prototype:

```
int UART_Shutdown(void);
```

Description: *UART_Shutdown()* shuts down USART0's interrupts. Call it when you are done with the UART. Note that you can still use the manual transmit and receive functionality of the UART, but the interrupt driven system is disabled after this call. Returns 1 always.

Example(s): Shutdown the UART interrupts.

```
UART_Shutdown();
```

Function Prototype:

```
int UART_putc(unsigned char ch);
```

Description: *UART_putc(...)* places a character into the outgoing transmit circular buffer from there the ISR will continue to process the characters and send them out the UART as the transmitter buffer empties. This function is non-blocking and returns immediately. Returns the character sent if successful, else 0 if fails.

Example(s): Send an ASCIIZ encoded string out to the UART.

```
int length;
char string[] = "This is a test";
.
.
// compute length
length = strlen( string );
// loop and send each character
for (index = 0; index < strlen; index++)
    UART_putc( string[ index ] );
```

Function Prototype:

```
void UART_Transmit(unsigned char ch);
```

Description: *UART_Transmit(...)* sends a single character out the UART. The function takes as a parameter the character to send, and the function blocks until the transmitter is free. Use this function if you don't care about using the transmitter interrupt and can wait for the characters to transmit before the function returns. Function returns nothing.

Example(s): Initialize UART to N81, 38400, manually transmit "Chameleon", shut the UART down.

```
// initialize to N81 always
UART_Init( 115200 );

// send the characters
UART_Transmit( 'C' );
UART_Transmit( 'H' );
UART_Transmit( 'A' );
UART_Transmit( 'M' );
UART_Transmit( 'E' );
UART_Transmit( 'L' );
UART_Transmit( 'E' );
UART_Transmit( 'O' );
UART_Transmit( 'N' );

// shutdown the UART system
UART_Shutdown();
```

As noted, the shutdown function really only disables the ISR's, so you can still use the manual transmit and receive functions if you wish.

Function Prototype:

```
void UART_Newline(void);
```

Description: *UART_Newline()* simply send a CR/LF (0x0A, 0x0D) to the UART. Assuming, a terminal program is connected to the Chameleon AVR, this function is a nice helper to print newlines when needed. Returns nothing.

Example(s): Assuming the Chameleon AVR is connected to a PC or other terminal with VT100 capabilities, print 10 newlines.

```
for (index = 0; index < 10; index++ )
    UART_Newline();
```

Function Prototype:

```
void UART_Print_String(char *string);
```

Description: *UART_Print_String(...)* prints a NULL terminated string. The function first prints a newline then the string, takes a pointer to the ASCII string. Returns nothing.

Example(s): Print out "Hello World!".

```
UART_Print_String("Hello world!");
```

Function Prototype:

```
void UART_Print_Int(int num);
```

Description: *UART_Print_Int(...)* takes a signed decimal integer and prints it to the UART. Returns nothing.

Example(s): Print the powers of 2 from 1 to 256.

```
int pow2 = 1;
while (pow2 <= 256)
{
```

```

UART_Print_Int( pow2 );
UART_Newline();
pow2 *= 2;
} // end while

```

Function Prototype:

```
void Terminal_Clear_Screen(void);
```

Description: *Terminal_Clear_Screen()* simply uses VT100 codes to clear the screen. If you are communicating with a terminal program that has VT100 emulation, calling this function will clear the screen and place the cursor at the top left of the screen.

Example(s): Clear the screen and print my favorite computer.

```

Terminal_Clear_Screen();
UART_Print_String( "Atari 800");

```

Function Prototype:

```
int UART_getc(void);
```

Description: *UART_getc()* retrieves the next character from the circular reception buffer. Returns the next character in the buffer or -1 if buffer is empty.

Example(s): Echo each character that is received.

```

// forever..
while(1)
{
    // get next character?
    ch = UART_getc();

    // was there a character?
    if (ch != -1)
        UART_putc( ch );
} // end while

```

Function Prototype:

```
int UART_Receive(int wait);
```

Description: *UART_Receive(...)* doesn't use the interrupt system. The function interrogates the UART hardware directly and checks if the receive buffer has a character in it, if so, the function returns it. However, the function can work in two different modes; blocking and non-blocking. That is, you can call the function with a **wait** = 1 and the function will wait for a character, or **wait** = 0, and the function will not wait for a character and return immediately if a character isn't ready. In either case, the function returns the character in the receiver buffer or -1 if a character isn't ready. Note that if you set **wait** = 1 the function could theoretically wait forever.

Example(s): Wait for the character 'X' when received blow up the world ☺

```

// wait for the 'X'
while( UART_Receive(1) != 'X' );

```

```
// call the mainframe computer in Zion and execute its last command..
Blow_Up_The_world();
```

19.0 SPI and I²C Library Module Primer

The “**SPI and I²C**” module consist of a single C source file and a number of functions for initialization, transmission and reception. The SPI module is one of the most important in the system since all communications with the Propeller chip are performed via the SPI hardware. The API consists of the following files listed below:

CHAM_AVR_TWI_SPI_DRV_V010.c - Main C file source for “SPI and I²C” module.
CHAM_AVR_TWI_SPI_DRV_V010.h - Header file for “SPI and I²C” module.

ARDUINO TIP

Arduino tool users you will use the “Import Library” command from the main menu to include **CHAM_AVR_TWI_SPI_DRV_V010**, however all this really does is add the line of code:

```
#include < CHAM_AVR_TWI_SPI_DRV_V010.h >
```

To your program, which you can just do yourself. The Arduino tool already knows where to look for other C/C++ files (since we copied them into the appropriate directories already), thus for the Arduino tool simply add the header include manually or you can let the tool do it from the main menu <Sketch -> Import Library>. The Arduino MAKE file is rather complex and knows how to add all the C/C++ files we dumped into the library directory, so there is little to do with the Arduino tool when accessing these libraries other than including the header in your program.

Before we discuss the module and functionality, let’s briefly review both SPI and I²C protocols and their features. For more detailed descriptions and tutorials on both protocols, I suggest reading the SPI and I²C material in the AVR 328 data sheet located here:

DVD-ROM:\ CHAM_AVR \ DOCS \ DATASHEETS \ ATmega48_88_168_328_doc8161.pdf

Also, there are numerous SPI and I²C documents located on the DVD here:

DVD-ROM:\ CHAM_AVR \ DOCS \ I2C \ *.*
DVD-ROM:\ CHAM_AVR \ DOCS \ SPI \ *.*

A few of which I highly recommend you read specifically are:

DVD-ROM:\ CHAM_AVR \ DOCS \ SPI \ avr_spi_doc2585.pdf
DVD-ROM:\ CHAM_AVR \ DOCS \ SPI \ avr_spi_master_doc1108.pdf
DVD-ROM:\ CHAM_AVR \ DOCS \ SPI \ SPI.pdf
DVD-ROM:\ CHAM_AVR \ DOCS \ I2C \ avr315_TWI_I2C.pdf
DVD-ROM:\ CHAM_AVR \ DOCS \ I2C \ twi_programming.htm
DVD-ROM:\ CHAM_AVR \ DOCS \ I2C \ I2C_bus.pdf

These cover AVR specific SPI controls as well as general software implementations and some interesting examples of SPI controls. But, if you just want a quick overview, read on...

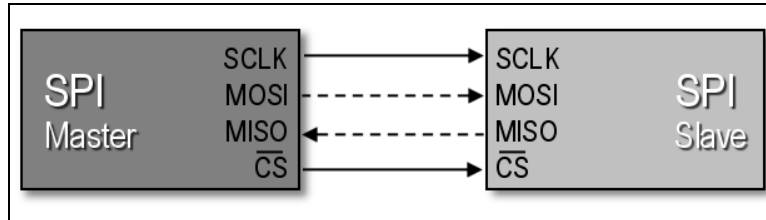
NOTE

The Atmel processors have multi-functional 2-wire hardware devices inside them. These devices can support I²C as well as other protocols under programmability. Thus, instead of calling the interfaces “I²C” Atmel calls them “TWI” for “Two Wire Interfaces”. I suspect it has something to do with licensing as well. And to avoid licensing the I²C interface Atmel created generalized hardware that supports the I²C format, but isn’t designed specifically for it. In any event, TWI = I²C, I²C = TWI for our purposes, but the registers

in the AVR 644 are all "TWI" prefixed and suffixed, so don't get confused. Finally, SPI has nothing to do with this, so "SPI" is called "SPI".

SPI stands for **Serial Peripheral Interface** originally developed by **Motorola**. Its one of two very popular modern serial standards including **I²C** which stands for **Inter Integrated Circuit** by **Phillips**. SPI unlike I²C (which has no separate clock) is a clocked synchronous serial protocol that supports full duplex communication. However I²C only takes **2 wires** and a ground. Where SPI needs **3 wires**, ground, and potentially chip select lines to enable the slave devices. But, SPI is much faster, so in many cases speed wins over and the extra clock line is warranted. The advantage of I2C is that you can potentially hook hundreds of I²C devices on the same 2-bus lines since I²C devices have addresses that they respond to. SPI bus protocol on the other hand requires that every SPI slave has a chip select line.

Figure 19.1 – The SPI electrical interface.



19.1 SPI Bus Basics

Figure 19.1 shows a simple diagram between a **master** (left) and a **slave** (right) SPI device and the signals between them which are:

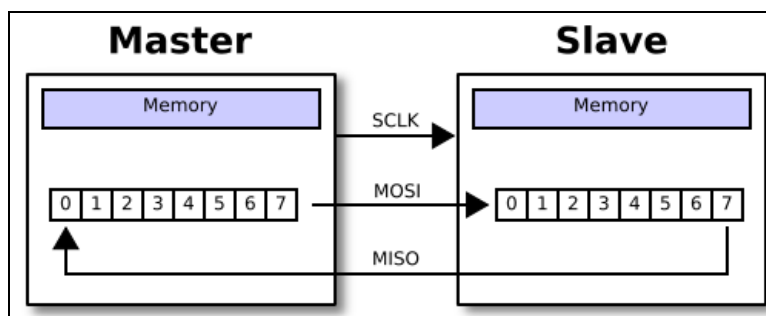
- **SCLK** - Serial Clock (output from master).
- **MOSI/SIMO** - Master Output, Slave Input (output from master).
- **MISO/SOMI** - Master Input, Slave Output (output from slave).
- **SS** - Slave Select (active low; output from master).

Note: You might find some devices with slightly different naming conventions, but the idea is there is data out and data in, a clock, and some kind of chip select.

If you refer back to the Chameleon's SPI interface discussion in Section 9.0 of the **Hardware Primer** you can see that these signal pins are mapped from the AVR to the Propeller chip.

SPI is very fast since not only is it clocked, but it's a simultaneous **full duplex** protocol which means that at you clock data out of the master into the slave, data is clocked from the slave into the master. This is facilitated by a transmit and receive bit buffer that constantly re-circulates as shown in Figure 19.1.

Figure 19.1 – Circular SPI buffers.



The use of the circular buffers means that you can send and receive a byte in only 8 clocks rather than clocking out 8-bits to send, then clocking in 8-bits to receive. Of course, in some cases the data clocked out or in is "dummy" data, meaning when you write data and you are **not** expecting a result the data you clock in is garbage and you can throw it away. Likewise when you do a SPI read, typically you would put a \$00 or \$FF in the transmit buffer as dummy data since something has to be sent and it might as well be predictable.

Sending bytes with SPI is similar to the serial RS-232 protocol, you place a bit of information on the transmit line, then strobe the clock line (of course RS-232 has no clock). As you do this, you also need to read the receive line since data is being transmitted in both directions. This is simple enough, but SPI protocol has some very specific details attached to it about **when** signals should be read and written that is, on the rising or falling edge of the clock as well as the polarity of the clock signal. This way there is no confusion about edge, level, or phase of the signals. These various modes of operation are logical called the SPI mode and are listed in Table 19.1 below:

Table 19.2 – SPI clocking modes.

Mode #	CPOL (Clock Polarity)	CPHA (Clock Phase)
0	0	0
1	0	1
2	1	0
3	1	1

Mode Descriptions

Mode 0 – The clock is **active** when **HIGH**. Data is **read** on the **rising edge** of the clock. Data is **written** on the **falling edge** of the clock (default mode for most SPI applications).

Mode 1 – The clock is **active** when **HIGH**. Data is **read** on the **falling edge** of the clock. Data is **written** on the **rising edge** of the clock.

Mode 2 – The clock is **active** when **LOW**. Data is **read** on the **rising edge** of the clock. Data is **written** on the **falling edge** of the clock.

Mode 3 – The clock is **active** when **LOW**. Data is **read** on the **falling edge** of the clock. Data is **written** on the **rising edge** of the clock.

Note: Most SPI slaves **default to mode 0**, so typically this mode is what is used to initiate communications with a SPI device.

Figure 19.2(a) – SPI timing diagrams for clock phase polarity (CPHA=0).

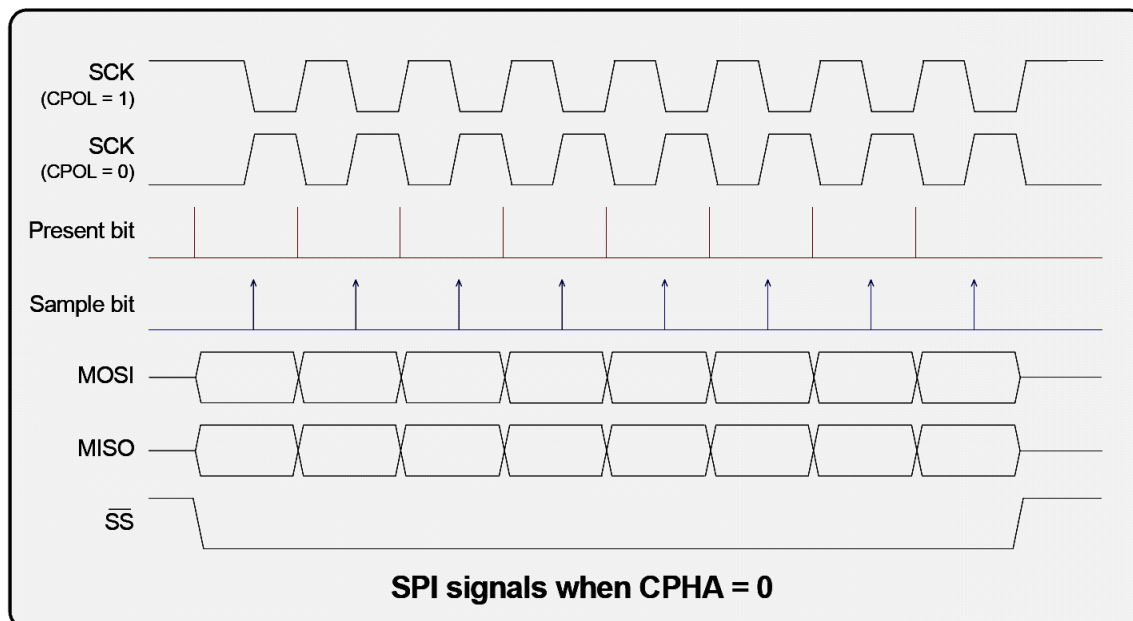
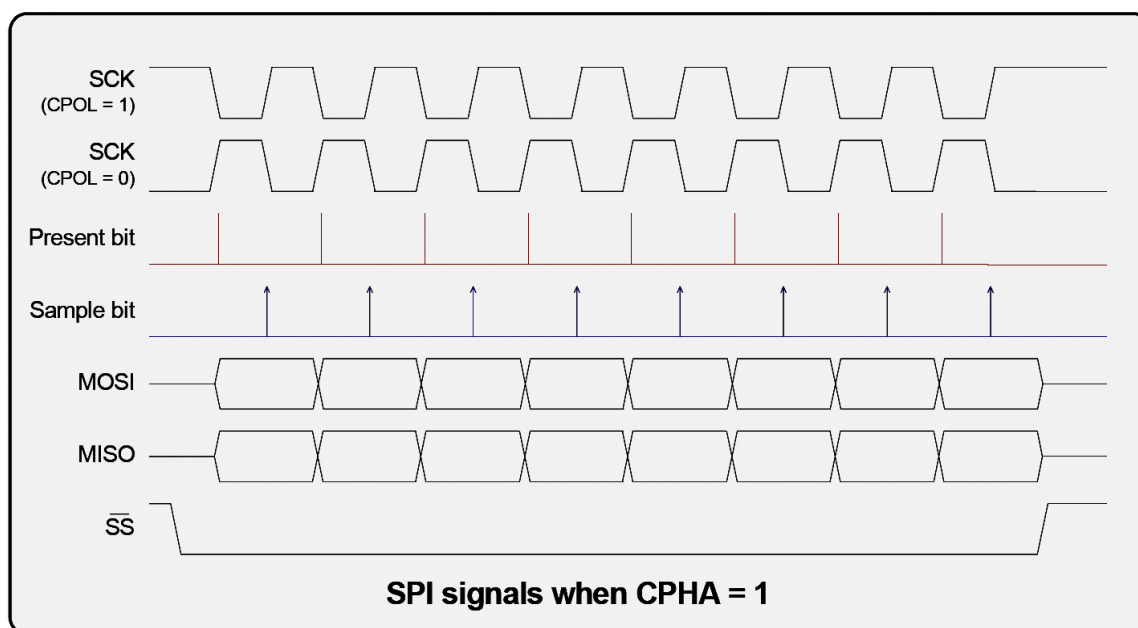


Figure 19.2(b) – SPI timing diagrams for clock phase polarity (CPHA=1).

19.1.1 Basic SPI Communications Steps

Figures 19.2(a) and (b) show the complete timing diagrams for all variants of **clock polarity** (CPOL) and **clock phase** (CPHA). You must adhere to these timing constraints during communications. In most cases, you will use **mode 0** since it's the default that most SPI devices boot with.

The good news is this is all taken care of by the AVR 328P, in fact, most AVR microcontrollers have complete SPI (and I²C) hardware built in that take care of all the details of both transmission and reception. The only thing we have to do as programmers is set the hardware up, then send and receive bytes (and maybe handle an interrupt or two). Thus, working with SPI in the AVR 328P is very easy. Nonetheless, the library module we developed wraps the SPI hardware with a thin layer of functions, so that you don't have to deal with it all yourself. However, it's good to understand the protocol if you need to manually build a SPI interface with I/O pins. For example, maybe you need (4) SPI interfaces all at the same time on the Chameleon's expansion port, the only way to do this will be to do it manually with pin toggling.

However, the AVR 328P's USARTs (basically UARTs with extra features) both support "SPI modes". In other words, even though the AVR 328P has a primary SPI controller, the USART on the AVR 328P can be put into "SPI mode" as well, giving you **(2)** fully functioning SPI devices, interrupt driven, etc. This is very good news. Moreover, the Chameleon's expansion headers port exports out the master SPI device itself as well as the ancillary SPI-mux selects.

Using the AVR SPI interface is straightforward, the steps are:

- Initialize the hardware (set speed, clock phase and polarity, etc.).
- Send and receive bytes via a pair of registers (and possibly interrupts).

That's all there is to it. Of course, there's a lot of detail to getting things to work and believe me, I had my share of frustration figuring out the little quirks, but the library functions provided shortly will get you up and running very quickly with SPI interfacing. Also, you can always write your own, or use external libraries. The AVR Libc system does not have any SPI library of note, but there are many out there like the **Procyon** library located here:

<http://hubbard.engr.scu.edu/embedded/avr/avrlib/>
<http://ccrma.stanford.edu/courses/250a/docs/avrlib/html/index.html>
<http://ccrma.stanford.edu/courses/250a/docs/avrlib/html/install.html>

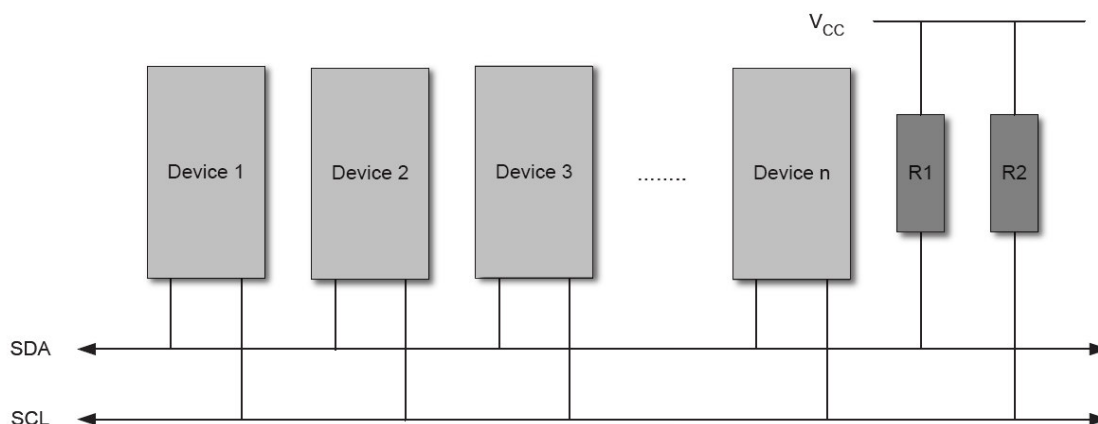
19.2 I²C Bus Basics

The I²C bus is a little more complex than the SPI bus interface and protocol. The reason is that the I²C bus uses only 2 signal lines (SDA – data, SCL – clock), thus more protocol and conventions must be adhered to, to deal with bus contention etc., secondly I²C supports up to **128** devices **simultaneously** connected to the bus! This feature makes I²C superior for **"daisy chaining"** devices together with as well as cheaper. Of course, you never get something for nothing

and the I²C bus is not without its shortcomings. First, it is nowhere near as fast as SPI. SPI can operate at 25 MHz and even up to 50 MHz. I²C on the other hand averages around 100 KHz+, with **400 KHz** being fast with many new devices supporting 1Mhz. Thus, SPI is at least 25 times faster. But, that's not the whole story. The added overhead that I²C protocol attaches to communication (addressing, commands, etc.) slow the protocol even more. Thus, I²C devices tend to find there way into "**slow**" peripherals where speed isn't and/or issue, but addressing many of them. For example, serial memories, sensors, etc. where the device itself is slow, the 100-400 KHz average speed of I²C is more than enough. However, SPI devices you will see in very high speed applications even video and audio.

Figure 19.4 shows an architectural diagram of how the I²C bus is laid out in relation to the master device, and the slaves on the line.

Figure 19.3 – I2C bus layout.

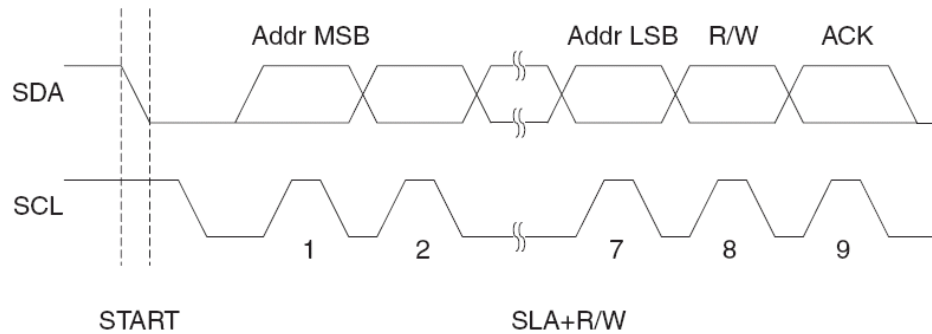


Electrically, the I²C bus consists of any number of "**masters**" and "**slaves**" on the bus. Masters initiate communication while slaves listen and respond. Masters can transmit and receive from a slave, but a slave can not initiate communications. Additionally, to enforce that masters are in charge, the clock line SCL can **only** be controlled by a master furthermore placing the slave(s) into a passive role. Moreover, so that multiple devices can be connected to the same 2-signal bus from an electrical point of view, both SDA and SCL are open drain, thus "pull-up" resistors must be connected from SDA and SCL to Vcc via a **5-10K** resistor on both lines. Note that with the SPI bus, all SPI devices share the MISO (master input), MOSI (master output), and SCLK (serial clock) lines; however, the CS or chip select lines for each device controls the selection of the target device (not an address), and when a device is selected its bus is active while any other de-selected devices go tristate. Thus, the I2C bus is **always** active and arbitration is achieved thru an open drain design where SDA and SCL can only be pulled down by the master, and SDA alone by the slave. Then addressing is achieved by a **7-bit** address sent down the I²C bus to **all** devices, only the listening device with the matching address responds and communication begin. Let's discuss this process more in detail.

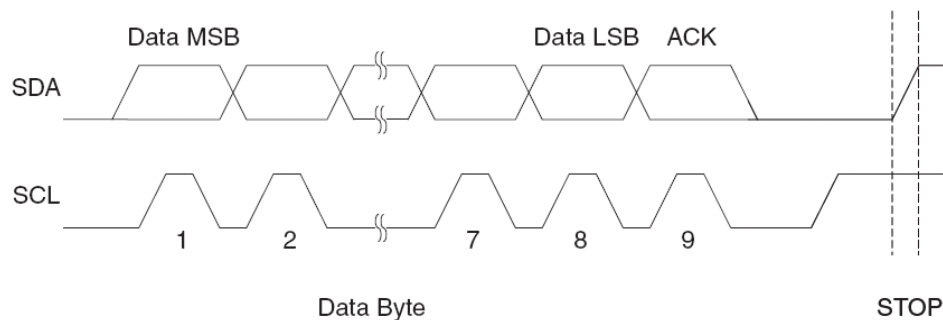
19.2.1 Understanding I²C Bus States

The I²C protocol is rather complex compared to SPI protocol. In fact, there is no "**SPI protocol**" per se. SPI is a simple serial packet much like RS-232. The only "**protocol**" that SPI devices have are inherent in the devices themselves and what the bytes sent and received mean. I²C on the other hand has this very complex **arbitration system** to handle multiple Masters on the bus, this is facilitated via the electrical design as well as a "**state machine**" that I²C is based on. Considering this, I²C is rather complex. We are only briefly going to cover some of the high level concepts of the protocol and its primary states; **START**, and **STOP**.

To begin with, data transfer is always initiated by a **Master** device (remember Slaves can only respond to Masters). A high to low transition on the SDA line while SCL is high is defined to be a "**START condition**" or a "**repeated START condition**". Everything begins with the "START" condition, that's the first thing to remember with I²C. Figure 19.4 below show the timing waveforms for this event.

Figure 19.4 – Timing waveforms for I²C “START Condition”.

A START condition is always followed by the (unique) 7-bit slave address along with a single bit signifying the “**Data Direction**” either read or write. The Slave device addressed acknowledges the Master by holding SDA low for one clock cycle. If the Master does not receive an acknowledgement the transfer is terminated. Depending of the **Data Direction bit**, the Master or Slave now transmits 8-bits of data on the SDA line. The receiving device then acknowledges the data. Figure 19.5 shows what happens during data transmission following the initial START and addressing phases.

Figure 19.5 – Timing waveforms for I²C data transmission.

Multiple bytes can be transferred in one direction before a repeated START or a STOP condition is issued by the Master. The transfer is terminated when the Master issues a “**STOP condition**”. A STOP condition is defined by a low to high transition on the SDA line while the SCL is high. Confusing isn't it?

If a Slave device cannot handle incoming data until it has performed some other function, it can hold SCL low to force the Master into a “**wait-state**” for as long as it needs to. All **data** packets transmitted on the I²C bus are 9 bits long, consisting of 8 data bits, and an acknowledge bit. Now, during a data transfer, the **Master** generates the clock and the START and STOP conditions, while the receiver is responsible for acknowledging the reception. An Acknowledge (ACK) is signaled by the receiver pulling the SDA line low during the 9th clock cycle pulse on SCL. If the receiver leaves the SDA line high, a “**NACK**” (no ACK) is signaled, and the transfer is not completed.

In conclusion, I²C is a lot more complex than SPI, and writing a software implementation is a lot harder as well due to the state machine, bus contention and so forth. However, the AVR 644P has built in I²C hardware as mentioned which makes it all much easier to implement. Setting up the registers is still a bit challenging and you have to read the documentation quite a bit to get the hang of it. I suggest you review the following references for a more complete treatise on I²C:

DVD-ROM:\ CHAM_AVR \ DOCS \ I2C \ avr315_TWI_I2C.pdf
 DVD-ROM:\ CHAM_AVR \ DOCS \ I2C \ twi_programming.htm
 DVD-ROM:\ CHAM_AVR \ DOCS \ I2C \ I2C_bus.pdf

Now, let's move onto the SPI and I²C library module that implements a very thin layer of software on top of the built in hardware. Moreover, you may wish to develop much higher level functionality yourself. If you do, make note that the “SD card” library uses the SPI functions in this module, so you will have to emulate their functionality and linker names.

19.3 Header File Contents Overview

The "SPI and I2C" module header **CHAM_TWI_SPI_V010.h** is very important since it houses the constants for the SPI communications interface to the Propeller, so we are going to cover it in detail. Let's begin with the #defines that declare some of the hardware signals and constants used to represent I²C and SPI interfaces):

```
// TWI/I2C defines
// I2C clock in Hz
#define TWI_CLOCK 100000L // initially set to 100khz, this is the default when you call TWI_Init(...)
#define SCL_CLOCK TWI_CLOCK

#define TWI_PORT PORTC
#define TWI_DDR PORTC
#define TWI_SCLK PORTC0
#define TWI_SDA PORTC1

// SPI defines
#define SPI_DDR DDRB
#define SPI_PORT PORTB
#define SPI_CS PORTB2 // not used for spi addressing
#define SPI_MOSI PORTB3
#define SPI_MISO PORTB4
#define SPI_SCLK PORTB5

#define SPI_CS_DDR DDRC
#define SPI_CS_PORT PORTC

#define SPI_CS0 PORTC2
#define SPI_CS1 PORTC3

// SPI rate bit settings
#define SPI_FOSC_2 0b100
#define SPI_FOSC_4 0b000
#define SPI_FOSC_8 0b101
#define SPI_FOSC_16 0b001
#define SPI_FOSC_32 0b110
#define SPI_FOSC_64 0b010
#define SPI_FOSC_128 0b011
```

The #defines simply declare the I/O ports, signal bits and some timing constants. For example, **TWI_CLOCK** is a convenient constant to pass the initialization function (covered later), and the last set of constants are paraphrased from the hardware control register settings for the SPI speed settings, but broken out, so you don't have to decode them yourself from the datasheet.

Also, note that the I²C and SPI #defines declare very specific hardware and I/O. The I²C interface for example uses the TWI hardware on **Port C** at bits 1:0 for the interface. These are exported via the expansion port. Similarly, the SPI interface is the "**primary**" SPI interface in the AVR 328P, not the USARTs in SPI mode. Moving on, the SPI channel connects to the SPI-Mux as well as the Propeller chip and is exported to the I/O headers. So, if you want to control an external SPI device you can use the other SPI-Mux addresses (the Propeller and FLASH use address 0 and 1).

Next, there are some timing constants that slow things down and make sure when SPI commands are sent to the Propeller the AVR waits long enough before issuing another command. These don't matter too much as the drivers evolved, but were needed in earlier driver development:

```
////////////////////////////////////
// TIMING CONSTANTS FOR SPI COMMS - PLAY WITH THESE IF THINGS GET FUNKY
////////////////////////////////////

// what rate to run the SPI at? this effects the following constants, faster rates, longer delays
#define SPI_DEFAULT_RATE (SPI_FOSC_32) // SPI_FOSC_32 for optimization settings -01, -02, -03, -0s
// SPI_FOSC_16 for no optimizations -00

// these defines "throttle" the SPI interaction with the virtual SPI software interface on the Prop which is slow
#define SPI_PROP_SLOW_FACTOR 4 // this number scales the delays, so if you turn on heavy optimization, then make this number
// larger to offset the code speed
// the optimizer seems to cause issues with the final code, interrupts, etc. if you do try turning on
// the optimizer, you will have to play with delays and get things to work properly
// Optimizer settings: -00 | use slow factor of 1 or 2
// -01, -02, -03, -0s | use slow factor of 4 or 5

#define SPI_PROP_STRING_DELAY_US (50*SPI_PROP_SLOW_FACTOR) // the prop needs time to react to each spi terminal print command
#define SPI_PROP_CMD_DELAY_US ((2*SPI_PROP_SLOW_FACTOR)/2) // the prop needs time to react to each spi transition,
// this delay insures it can keep up
// if you optimize the Prop driver or convert to ASM then you can
// reduce this of course

#define SPI_PROP_DELAY_SHORT_US (100*SPI_PROP_SLOW_FACTOR) // this delay is used where the critical path thru the SPI/Prop is relatively short
#define SPI_PROP_DELAY_LONG_US (250*SPI_PROP_SLOW_FACTOR) // this delay is used where the critical path thru the SPI/Prop is relatively long
```

The next set of `#defines` are rather long, they represent all the current commands that the Propeller driver understands. These commands each have a specific set of parameters and functionality that will be clarified when we review the individual APIs for each class of messages; graphics, sound, keyboard, etc.

```
// display devices
#define DEVICE_NTSC      0
#define DEVICE_VGA      1

// PROP SPI driver media and IO command set

// SPI commands
#define CMD_NULL          0

// NTSC commands
#define GFX_CMD_NTSC_PRINTCHAR 1
#define GFX_CMD_NTSC_GETX     2
#define GFX_CMD_NTSC_GETY     3
#define GFX_CMD_NTSC_CLS     4

// vga commands
#define GFX_CMD_VGA_PRINTCHAR 8
#define GFX_CMD_VGA_GETX     9
#define GFX_CMD_VGA_GETY    10
#define GFX_CMD_VGA_CLS    11

// keyboard commands
#define KEY_CMD_RESET       16
#define KEY_CMD_GOTKEY      17
#define KEY_CMD_KEY         18
#define KEY_CMD_KEYSTATE    19
#define KEY_CMD_START       20
#define KEY_CMD_STOP        21
#define KEY_CMD_PRESENT     22

// mouse commands
#define MOUSE_CMD_RESET     24 // resets the mouse and initializes it
#define MOUSE_CMD_ABS_X     25 // returns the absolute X-position of mouse
#define MOUSE_CMD_ABS_Y     26 // returns the absolute Y-position of mouse
#define MOUSE_CMD_ABS_Z     27 // returns the absolute Z-position of mouse
#define MOUSE_CMD_DELTA_X   28 // returns the delta X since the last mouse call
#define MOUSE_CMD_DELTA_Y   29 // returns the delta Y since the last mouse call
#define MOUSE_CMD_DELTA_Z   30 // returns the delta Z since the last mouse call
#define MOUSE_CMD_RESET_DELTA 31 // resets the mouse deltas
#define MOUSE_CMD_BUTTONS   32 // returns the mouse buttons encoded as a bit vector
#define MOUSE_CMD_START     33 // starts the mouse driver, loads a COG with it, etc.
#define MOUSE_CMD_STOP      34 // stops the mouse driver, unloads the COG its running on
#define MOUSE_CMD_PRESENT   35 // determines if mouse is present and returns type of mouse

// general data readback commands
#define READ_CMD            36

// sound commands
#define SND_CMD_PLAYSOUNDFM 40 // plays a sound on a channel with the sent frequency at 90% volume
#define SND_CMD_STOPSOUND   41 // stops the sound of the sent channel
#define SND_CMD_STOPALLSOUNDS 42 // stops all channels
#define SND_CMD_SETFREQ     43 // sets the frequency of a playing sound channel
#define SND_CMD_SETVOLUME   44 // sets the volume of the playing sound channel
#define SND_CMD_RELEASESOUND 45 // for sounds with infinite duration, releases the sound and it enters the "release" portion of ADSR envelope

// propeller local 8-bit port I/O commands
#define PORT_CMD_SETDIR     48 // sets the 8-bit I/O pin directions for the port 1=output, 0=input
#define PORT_CMD_READ       49 // reads the 8-bit port pins, outputs are don't cares
#define PORT_CMD_WRITE      50 // writes the 8-bit port pins, port pins set to input ignore data

// general register access commands, Propeller registers for the SPI driver cog can be accessed ONLY
// but, the user can leverage the counters, and even the video hardware if he wishes, most users will only
// play with the counters and route outputs/inputs to/from the Propeller local port, but these generic access
// commands model how you would access a general register based system remotely, so good example
// these commands are DANGEROUS since you can break the COG with them and require a reset, so if you are going to
// write directly to the registers, be careful.

#define REG_CMD_WRITE       56 // performs a 32-bit write to the addressed register [0..F] from the output register buffer
#define REG_CMD_READ        57 // performs a 32-bit read from the addressed register [0..F] and stores in the input register buffer
#define REG_CMD_WRITE_BYTE  58 // write byte 0..3 of output register g_reg_out_buffer.byte[ 0..3 ]
#define REG_CMD_READ_BYTE   59 // read byte 0..3 of input register g_reg_in_buffer.byte[ 0..3 ]

// system commands
#define SYS_RESET           64 // resets the prop

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Commands in this range for future expansion...
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

The commands are fairly straightforward and if there is anything that you want more detail on you can always open up the Propeller driver file's spin code and see exactly how the command is processed. Remember, all of these commands work in the same way; they are sent over the SPI interface to a command dispatching loop on the Propeller, a large case statement catches the commands and then sends it to the appropriate driver (graphics, sound, keyboard, etc.). So there are a lot of commands, but each one is rather simple when you follow its complete path from the AVR to the Propeller to the media driver itself.

The next subset of commands are specific to the default2 driver and support the added graphics abilities of the tile map engine for NTSC modes. I wrote the tile engine to give you more performance and the ability to do some game like applications. These commands are more complex and it will take a bit to explain them, but for now, just peruse them below:

```

// advanced GFX commands for GFX tile engine
#define GPU_GFX_BASE_ID 192 // starting id for GFX commands to keep them away from normal command set
#define GPU_GFX_NUM_COMMANDS 37 // number of GFX commands

#define GPU_GFX_SUBFUNC_STATUS_R (0+GPU_GFX_BASE_ID) // Reads the status of the GPU, writes the GPU Sub-Function register and issues a high level
// command like copy, fill, etc.
#define GPU_GFX_SUBFUNC_STATUS_W (1+GPU_GFX_BASE_ID) // Writes status of the GPU, writes the GPU Sub-Function register and issues a high level
// command like copy, fill, etc.

// sub-function constants that are executed when the GPU_GFX_SUBFUNC_STATUS_W command is issued
#define GPU_GFX_SUBFUNC_COPYMEM16 0 // Copies numbytes from src -> dest in wordsize chunks
#define GPU_GFX_SUBFUNC_FILLMEM16 1 // Fills memory with data16, 2 bytes at a time

#define GPU_GFX_SUBFUNC_COPYMEM8 2 // Copies numbytes from src -> dest in byte size chunks
#define GPU_GFX_SUBFUNC_FILLMEM8 3 // Fills memory with low byte of data16, 1 bytes at a time

// normal commands
#define GPU_GFX_TILE_MAP_R (2+GPU_GFX_BASE_ID) // Reads 16-bit tile map ptr which points to the current tile map displayed.
#define GPU_GFX_TILE_MAP_W (3+GPU_GFX_BASE_ID) // Writes 16-bit tile map ptr which points to the current tile map displayed.

#define GPU_GFX_DISPLAY_PTR_R (4+GPU_GFX_BASE_ID) // Reads 16-bit tile map ptr which points to the current display ptr into the tile map, low
// level print functions use this pointer.
#define GPU_GFX_DISPLAY_PTR_W (5+GPU_GFX_BASE_ID) // Writes 16-bit tile map ptr which points to the current display ptr into the tile map, low
// level print functions use this pointer.

#define GPU_GFX_TERM_PTR_R (6+GPU_GFX_BASE_ID) // Reads 16-bit tile map ptr which points to the current terminal ptr into the tile map,
// terminal level print functions use this pointer.
#define GPU_GFX_TERM_PTR_W (7+GPU_GFX_BASE_ID) // Writes 16-bit tile map ptr which points to the current terminal ptr into the tile map,
// terminal level print functions use this pointer.

#define GPU_GFX_BITMAP_R (8+GPU_GFX_BASE_ID) // Reads 16-bit tile bitmaps ptr which points to the bitmaps indexed by the tilemap.
#define GPU_GFX_BITMAP_W (9+GPU_GFX_BASE_ID) // Reads 16-bit tile bitmaps ptr which points to the bitmaps indexed by the tilemap.

#define GPU_GFX_PALETTE_R (10+GPU_GFX_BASE_ID) // Reads 16-bit palette array ptr which points to the palettes in use for the tilemap.
#define GPU_GFX_PALETTE_W (11+GPU_GFX_BASE_ID) // Writes 16-bit palette array ptr which points to the palettes in use for the tilemap.

#define GPU_GFX_TOP_OVERSCAN_COL_R (12+GPU_GFX_BASE_ID) // Reads top overscan color drawn on the screen.
#define GPU_GFX_TOP_OVERSCAN_COL_W (13+GPU_GFX_BASE_ID) // Writes top overscan color drawn on the screen.

#define GPU_GFX_BOT_OVERSCAN_COL_R (14+GPU_GFX_BASE_ID) // Reads top overscan color drawn on the screen.
#define GPU_GFX_BOT_OVERSCAN_COL_W (15+GPU_GFX_BASE_ID) // Writes top overscan color drawn on the screen.

#define GPU_GFX_HSCROLL_FINE_R (16+GPU_GFX_BASE_ID) // Reads current fine horizontal scroll register (0..7) NOTE: ( NOT implemented yet)
#define GPU_GFX_HSCROLL_FINE_W (17+GPU_GFX_BASE_ID) // Writes current fine horizontal scroll register (0..7) NOTE: ( NOT implemented yet)

#define GPU_GFX_VSCROLL_FINE_R (18+GPU_GFX_BASE_ID) // Reads current fine vertical scroll register (0..7)
#define GPU_GFX_VSCROLL_FINE_W (19+GPU_GFX_BASE_ID) // Writes current fine vertical scroll register (0..7)

#define GPU_GFX_SCREEN_WIDTH_R (20+GPU_GFX_BASE_ID) // Reads screen width value, 0=16 tiles, 1=32 tiles, 2=64 tiles, etc.
#define GPU_GFX_SCREEN_WIDTH_W (21+GPU_GFX_BASE_ID) // Writes screen width value, 0=16 tiles, 1=32 tiles, 2=64 tiles, etc.

#define GPU_GFX_SRC_ADDR_R (22+GPU_GFX_BASE_ID) // Reads 16-bit source address for GPU operations.
#define GPU_GFX_SRC_ADDR_W (23+GPU_GFX_BASE_ID) // Writes 16-bit source address for GPU operations.

#define GPU_GFX_DEST_ADDR_R (24+GPU_GFX_BASE_ID) // Reads 16-bit destination address for GPU operations.
#define GPU_GFX_DEST_ADDR_W (25+GPU_GFX_BASE_ID) // Writes 16-bit destination address for GPU operations.

#define GPU_GFX_NUM_BYTES_R (26+GPU_GFX_BASE_ID) // Reads 16-bit number representing the number of bytes for a GPU operation Sub-Function.
#define GPU_GFX_NUM_BYTES_W (27+GPU_GFX_BASE_ID) // Writes 16-bit number representing the number of bytes for a GPU operation Sub-Function.

#define GPU_GFX_DATA_R (28+GPU_GFX_BASE_ID) // Reads 16-bit data word uses for GPU operations or memory access operations.
#define GPU_GFX_DATA_W (29+GPU_GFX_BASE_ID) // Writes 16-bit data word uses for GPU operations or memory access operations.

#define GPU_GFX_RAM_PORT8_R (30+GPU_GFX_BASE_ID) // Reads 8-bit data pointed to by the currently addressed memory location in the GPU pointed to
// by the src_addr_low|hi.
// After the operation, the src_addr_ptr is incremented as per the GPU configuration.
#define GPU_GFX_RAM_PORT8_W (31+GPU_GFX_BASE_ID) // Writes 8-bit data pointed to by the currently addressed memory location in the GPU pointed
// to by the src_addr_low|hi.
// After the operation, the src_addr_ptr are incremented as per the GPU configuration.

#define GPU_GFX_RAM_PORT16_R (32+GPU_GFX_BASE_ID) // Reads 16-bit data pointed to by the currently addressed memory location in the GPU pointed
// to by the src_addr_low|hi.
// After the operation, the src_addr_ptr is incremented as per the GPU configuration.
#define GPU_GFX_RAM_PORT16_W (33+GPU_GFX_BASE_ID) // Writes 16-bit data pointed to by the currently addressed memory location in the GPU pointed
// to by the src_addr_low|hi.
// After the operation, the src_addr_ptr are incremented as per the GPU configuration.

#define GPU_GFX_RASTER_LINE_R (34+GPU_GFX_BASE_ID) // Reads the current raster line being drawn from 0..191, or whatever the GPU's line resolution
// is set to.

// gpu configuration registers
#define GPU_GFX_SET_AUTO_INC_R (35+GPU_GFX_BASE_ID) // Reads current memory auto increment setting for read/write operations lower 4-bits (0..15),
// default 0
#define GPU_GFX_SET_AUTO_INC_W (36+GPU_GFX_BASE_ID) // Writes the current memory auto increment setting for read/write operations lower 4-bits
// (0..15), default 0

```

And that's about it for #defines, let's move on to the API roadmap.

19.4 API Listing Reference

The API listing for the “SPI and I²C” module **CHAM_TWI_SPI_V010.c** supports a thin “*wrapper*” layer over the hardware functionality. The SPI API is more developed since the I²C tends to have a lot “*state*” in it, functions have to return with errors and other information, hence, you will probably want to write your own functions in the future. Nonetheless, I have used the thin wrapper to show how to control an I²C peripheral (a digital potentiometer) which we will cover in the **Mechatronics** section for reference. So the functions work, but you have to call them at a fairly low level. Table 19.3 below lists the “SPI and I²C” API functions categorized by functionality.

Table 19.3 – “SPI and I²C” module API functions listing.

Function Name	Description
TWI / I2C related functions	
int TWI_Init(long twi_clock);	Initializes the TWI / I2C hardware.
void TWI_Error();	Place holder for error handling ¹ .
void TWI_Status();	Place holder for status codes ¹ .
void TWI_Success(int code);	Place holder for success codes ¹ .
char TWI_Action(unsigned char command);	Sends an “action” to the TWI hardware.
SPI related functions	
int SPI_Init(unsigned char spi_rate);	Initializes the primary SPI unit (connected to SD).
int SPI_Set_Speed(unsigned char spi_rate);	Sets/resets the SPI speed.
unsigned char SPI_Write(unsigned char data8);	Writes a byte to the SPI interface.
unsigned char SPI_Read(unsigned char data8);	Reads a byte from the SPI interface.
unsigned char SPI_WriteRead(unsigned char data8);	Writes and reads a byte to/from the SPI interface.
unsigned char SPI_Send_Recv_Byte(unsigned char data8);	Writes and reads a byte to/from the SPI interface.
SPI command function to communicate with Propeller	
long SPI_Prop_Send_Cmd(int cmd, int data, int status);	Sends a 3-byte command packet to Propeller.
int SPI_Prop_Print_String(int device, char *string);	Legacy function, uses terminal NTSC/VGA driver to print strings on the display.
Note 1: Not implemented yet, just a place holder.	

19.5 API Functional Declarations

The following lists each function, a description, comments, and example usage of the function.

Function Prototype:

```
int TWI_Init(long twi_clock);
```

Description: *TWI_Init(...)* takes a single parameter which is the speed in Hz that you wish the TWI hardware interface to be initialized to. 100KHz is usually used, although some devices can support 400KHz as well as 1MHz, read your target I²C device for details. Recall, that the **TWI** (two wire interface) **is** the **I²C** interface for our purposes since we are setting it up to conform to I²C. Returns 1 always.

Example(s): Initialize the TWI / I²C interface to 400KHz.

```
TWI_Init( 400000 );
```

Function Prototype:

```
char TWI_Action(unsigned char command);
```

Description: The *TWI_Action(...)* function is very powerful since it really doesn't do much other than pass the

parameter to the TWI hardware "**control register**" (listed as **TWCR** in the datasheet). Then the function waits for the operation to complete by blocking on the **TWINT** flag. Normally, you use the **TWI_Action(...)** function to perform any and all operations by logically OR'ing various TWI flags together. The flags can be found in the AVR 328P datasheet. Returns the TWI hardware module's status register **TWSR**, this can be tested against common TWI status and success values to determine the results of any action.

Example(s): The complete listing of the digital potentiometer function excerpted from the Mechatronics library.

```
void POT_Set(int pot_value)
{
    // UPDATE TWI POT
    // currently supports an Analog Devices AD5242, PDF datasheet can be found on DVD here:
    // Manufacture page here:
    // http://www.analog.com/en/digital-to-analog-converters/digital-potentiometers/ad5241/products/product.html
    // this function requires the use of the "CHAM_AVR_TWI_SPI_DRV_V010.c|h" function library

    // send start command
    if ( TWI_Action( SET(TWINT) | SET(TWEN) | SET(TWSTA) )!= TW_START) TWI_Error();

    // now set slave address and high bit of 17-bit address "page bit" P0, selects which 64K page
    TWDR = 0x58; // 0 1 0 1 | 1 AD1 AD0 R/W' -> 0101_1000 for write, 0101_1001 for read

    if ( TWI_Action( SET(TWINT) | SET(TWEN) )!= TW_MT_SLA_ACK) TWI_Error();

    // now write instruction byte, reset, select pot A, 00 on digital outputs
    TWDR = 0x00; // A'/B RS SD 01 02 X X X -> 0 1 0 0 | 0 0 0 0

    if ( TWI_Action( SET(TWINT) | SET(TWEN) )!= TW_MT_DATA_ACK) TWI_Error();

    // now write pot value 0..255
    TWDR = pot_value;

    if ( TWI_Action( SET(TWINT) | SET(TWEN) )!= TW_MT_DATA_ACK) TWI_Error();

    /* send stop condition */
    TWCR = SET(TWINT) | SET(TWEN) | SET(TWSTO);

    // wait until stop condition is executed and bus released
    while(TWCR & (1<<TWSTO));

    // END TWI WRITE
} // end POT_Set
```

As you can see, using the **TWI_Action(...)** function must be interleaved with the protocol of the target device. The low level I²C protocol is handled by each call to **TWI_Action(...)** via the hardware, but the actual communication bytes, setup, and control of the target device (a digital pot in this case) is a completely other issue. Reviewing the code, you can see that the first step is to send the START and address to the device, then start writing data to control the pot. Also, notice the testing of the return value from each call to **TWI_Action(...)**. Depending on the results you can take what action you must. In this case, the empty function **TWI_Error()** are called, which you can insert any code you wish.

Function Prototype(s):

```
void TWI_Error();
void TWI_Status();
void TWI_Success(int code);
```

Description(s): These functions are not implemented and are simply place holders or wrappers for future expansion. As noted in the primer on I2C, the protocol has a lot of state and you might want to add these functions to the API and fill them out. Currently, a much simpler approach is taken where the single "action" function (listed above) performs both any action as well as returns status and errors.

Example(s): None.

Function Prototype:

```
int SPI_Init(unsigned char spi_rate);
```

Description: *SPI_Init(...)* initializes the primary SPI hardware on the AVR 328 to master and sets it up so we can transmit and receive. Also, it's important to recall with SPI whenever you send a byte you receive a byte as well due to the circular buffering of output and input. You can review the function yourself for details, but it sets the clock polarity and phase polarity to default mode 0 in both cases. Also, the single parameter sent *spi_rate* is used to set the speed of the SPI interface in the "**SPI Control Register 0**" speed control bits. The speed control bits control the "divisor" to the SPI hardware where the clock source is the input frequency of the AVR 328P, thus, the control bits set the divisor allowing a number of possible SPI clock rates. Writing to the SPI Control Register is straightforward, but some bit manipulation and shifting is needed to obtain the desired results, thus the constants declared in the .H file below can be used to set the clock speed:

```
// SPI rate bit settings
#define SPI_FOSC_2   0b100
#define SPI_FOSC_4   0b000
#define SPI_FOSC_8   0b101
#define SPI_FOSC_16  0b001
#define SPI_FOSC_32  0b110
#define SPI_FOSC_64  0b010
#define SPI_FOSC_128 0b011
```

Finally, the function de-asserts the *SPI_CS* line, so any SPI device connected to the expansion interface will be de-selected (if it's using the *SPI_CS* line as the chip select, recall that you can use any I/O line you wish to enable/disable a SPI slave, but bottom line is only a single SPI device better be on the MISO, MOSI, SCLK lines at any time). Considering this, when you write code to control the SPI interface for example, before you make calls to SPI write and read functions you will need to asserts the proper *SPI_CS* or appropriate SPI mux chip selects *SPI_CS0*, *SPI_CS1* to your SPI device while making sure all others are de-asserted. Returns 1 always.

Example(s): Example 1: Initialize the SPI hardware and set speed to main clock divided by 128 (the slowest rate).

```
SPI_Init( SPI_FOSC_128 );
```

Programming Sidebar

It is important to remember that the SPI's built in CS line is not used to select anything. Instead a pair of digital I/O lines are used as SPI "select" selection bits and are multiplexed to select 1 of 4 devices named 0...3. The signals in the Chameleon design are on Port C

```
#define SPI_CS0   PC2
#define SPI_CS1   PC3
```

To select device 0...3 you simply place 00_b, 01_b, 10_b, 11_b on these pins. 11_b (3) selects the FLASH, 10_b (2) selects the Propeller as destination. The other two unused selectors are exported off the I/O headers to support other SPI devices you might connect. For example, to select the Propeller chip you would use the following code:

```
// set CS to SPI select, select Prop SPI channel = 2
SPI_CS_PORT = (1 << SPI_CS1) | (0 << SPI_CS0);
```

And to de-select the Propeller and select device 0 (null device), you would use :

```
// set CS to SPI select channel 0 (null)
SPI_CS_PORT = (0 << SPI_CS1) | (0 << SPI_CS0);
```

Function Prototype:

```
int SPI_Set_Speed(unsigned char spi_rate);
```

Description: *SPI_Set_Speed(...)* simply updates the speed of the SPI interface in real-time, allowing you to "*throttle*"

the SPI interface. In many cases, SPI protocols demand that the slowest speed is used to initiate communications with the SPI hardware, then you can throttle the speed up and/or read maximum speed supported from the SPI device itself. In any event, this function gives you the ability to change speed on the fly (basically updates the SPI speed setting control bits). The parameter ***spi_rate*** is the same used in the initialization function, so are the possible parameter values. Returns 1 always.

Example(s): Assuming the SPI interface has already been initialized, reset the speed to the maximum rate which is 1/2 the clock rate or 14 MHz roughly (most SPI devices should easily be able to handle this speed).

```
SPI_Set_Speed( SPI_FOSC_2 );
```

Function Prototype:

```
unsigned char SPI_Send_Recv_Byte(unsigned char data8)
unsigned char SPI_WriteRead(unsigned char data8);
```

Description: Both of these functions are aliases for each other and have the exact same body, thus we will refer to both as ***SPI_WriteRead(...)*** since it's a little shorter to type. Recalling, that the SPI protocol always writes 8-bits at a time and reads 8-bits at a time, each write operation really is a read operation, and each read a write. Confusing huh? Well, its due to the circular buffer the system uses. You might ask, what if you want to read something, how can you write something when you are trying to read? The answer is that when you want to read data, you simply put a dummy value (usually **0xFF**) in the transmit buffer which the Slave will ignore. Same thing when you are writing data and not reading. The receive buffer will have some data in it after you write a byte, but you can ignore it. Thus, in operations that write and read at the **same time**, full duplex in other words, you get 2 pieces of data at the same time, this is the beauty of SPI in full duplex mode, 2x the clock rate for intent purposes. But, if you are only writing or reading at any time then this feature is irrelevant. Moving along, the function takes a single parameter ***data8*** which is the data you want written to the SPI interface. The function returns the data received **after** the data you sent is transmitted. In other words, the function ***waits*** for the 8-bits you sent to shift out as it shifts in the new 8-bits from the Slave. Thus, the data in the receiver buffer from the previous transaction ***is lost***. Moreover, the function is a **"blocking"** function and the transaction must complete before the function returns. Of course, you can code your own variations that just dump the data into the transmit buffer and return. But, then you would need a status function to determine if the transmitter is complete and or receiver has shifted in results before writing more data. The AVR SPI hardware supports this kind of system as well as full interrupt based systems. Finally, the function returns the 8-bit data shifted back to the Master from the Slave.

Example(s): Example 1: Write a value to the SPI interface channel 1 (assumes its been initialized).

```
// set CS to SPI select, select device channel 1
SPI_CS_PORT = (0 << SPI_CS1) | (1 << SPI_CS0);

// now write the data
dummy = SPI_WriteRead( power );

// and disable the chip select lines, set to NULL device 0
SPI_CS_PORT = (0 << SPI_CS1) | (0 << SPI_CS0);
```

Notice that we always assert then de-assert the chip select line, so that the SPI device comes online. In some case, you might want to keep the SPI device selected (enabled) at all times as long as you don't have other SPI devices on the bus that can contend with the SD SPI interface.

Example 2: Read 512 bytes from the SPI interface device 1 (assumes its been initialized somehow) and store in an array.

```
// set CS to SPI select, select device channel 1
SPI_CS_PORT = (0 << SPI_CS1) | (1 << SPI_CS0);

// read data, note that a dummy value of 0xFF is sent each time
for (index = 0; index < 512; index++)
    data[ index ] =SPI_WriteRead( 0xFF );
```

```
// and disable the chip select lines, set to NULL device 0
SPI_CS_PORT = (0 << SPI_CS1) | (0 << SPI_CS0);
```

Additionally, there might be "timing constraints" on how fast you can read bytes of data from the SPI device you are dealing with. Thus, you might have to put a delay after each read, etc.

Function Prototype:

```
unsigned char SPI_Write(unsigned char data8);
unsigned char SPI_Read(unsigned char data8);
```

Description: These functions are currently empty place holders, you might want to fill them up with special non-blocking versions of the write and read functionality or something else.

Example(s): None.

Function Prototype:

```
long SPI_Prop_Send_Cmd(int cmd, int data, int status);
```

Description: *SPI_Prop_Send_Cmd(...)* is the workhorse of the entire inter-processor communications system. This single function is what is used to send and receive information to and from the Propeller chip. Each command has 3-byte always. The first byte **cmd** is the command 0..255 that represents what you want to do. These are the same commands that are listed in the header file and the **CHAM_DEFAULT2_DRV_V112.SPIN** message dispatcher is listening for. The second byte **data** is the first 8-bit data operand, the 3rd byte **status** is the 2nd 8-bit data operand. For commands that require 16-bits, then **data** and **status** are concatenated to form a 16-bit operand like this:

```
msb [status, data] lsb
```

Thus, **data** is the lower 8-bits and **status** is the upper 8-bits.

For commands that only require 8-bit operands set **status** to 0. If you recall, SPI is a "circular" buffer system, so each byte we send out, we receive a byte back, thus when you call this function, 3 bytes are sent, 3 bytes are received. Currently, the function returns a 16-bit value representing the first two returned bytes, the 3rd byte is used internally for now. Also, since this is such an important function, let's take a look at the source for it:

```
// this function waits for any previous command to complete and then initiates a new command
long SPI_Prop_Send_Cmd(int cmd, int data, int status)
{
    // wait for previous command to complete
    while (SPI_Prop_Send_Cmd2(CMD_NULL, 0, 0) & 0x00ff0000)
        ;

    // initiate the new command
    return SPI_Prop_Send_Cmd2(cmd, data, status) & 0x0000ffff;
} // end SPI_Prop_Send_Cmd
```

As you can see the *SPI_Prop_Sen_Cmd(...)* function waits on a flag while calling a hidden function that actually does the work. Let's talk about that for a moment. In multiprocessor systems, when once processor requests another processor to do something, the whole point of multiprocessors, is that the originating processor can continue to do more work in "parallel". Thus, we need code to consider this such that say processor A (the AVR) sends a command to the processor B (the Propeller). Now, processor A goes about its business while processor B executes the command. But, the processor A finishes something and then requests processor B to do something else – here is the problem. Processor B can't perform another function until it completes the last requested command, thus we need a mechanism that blocks processor A and makes it wait for processor B when this set of conditions occur. That's what the code above does. So if processor A, the AVR, ask for too much, too fast, processor B, the Propeller, says "hold on a minute, let me finish the last thing!".

The function that actually sends the command is listed below:

```
// this function is used to send SPI commands to the Propeller slave processor
long SPI_Prop_Send_Cmd2(int cmd, int data, int status)
{
    long status_low, status_high, status_busy;

    // set CS to SPI select, select Prop SPI channel = 2
1:   SPI_CS_PORT = (1 << SPI_CS1) | (0 << SPI_CS0);

    // send command byte and retrieve low byte of result
2:   status_low = SPI_WriteRead( cmd );
3:   _delay_us(SPI_PROP_CMD_DELAY_US);

    // send command byte and retrieve high byte of result
4:   status_high = SPI_WriteRead( data );
5:   _delay_us(SPI_PROP_CMD_DELAY_US);

    // send command byte and retrieve busy byte of result
6:   status_busy = SPI_WriteRead( status );
7:   _delay_us(SPI_PROP_CMD_DELAY_US);

    // set CS to SPI select channel 0 (null)
8:   SPI_CS_PORT = (0 << SPI_CS1) | (0 << SPI_CS0);
9:   _delay_us(SPI_PROP_CMD_DELAY_US);

10:  return( (status_busy << 16) | (status_high << 8) | (status_low) );
} // end SPI_Prop_Send_Cmd2
```

Even though you don't directly call this function, its where all the action happens. Let's go thru it line by line (I have added manual line numbers in the listing to help). The function uses the built in SPI hardware and makes calls to the SPI functions we already developed, so that makes things a lot easier. Here's how it works:

Propeller Chip Selection

Step 1: Line 1 – here we select channel 2 with the SPI mux, this chip selects the Propeller chip as the target for SPI traffic.

Data Transmission

Step 2: Lines 2,3 – here we simply send the 1st byte of the command packet and retrieve the 1st byte back in a buffer.

Step 3: Lines 4,5 – here we simply send the 2nd byte of the command packet and retrieve the 2nd byte back in a buffer.

Step 4: Lines 6,7 – here we simply send the 3rd byte of the command packet and retrieve the 3rd byte back in a buffer.

Propeller Chip De-Selection

Step 5: Lines 8,9 – here we de-select channel 2 with the SPI mux and select channel 0 (the NULL channel).

Results

Step 6: Line 10 – We return the 8-16 bit result to caller.

This is how all commands are sent to the Propeller chip.

Example(s): Send command to NTSC driver to clear the screen. So first thing we do is look in the header file at the command list in **CHAM_AVR_TWI_SPI_DRV_V010.h** (also the Propeller driver **CHAM_DEFAULT2_DRV_112.SPIN** has the same list) and look up the command we want; **GFX_CMD_NTSC_CLS**, and now send the command with a function call.

```
// clear the NTSC screen, this command takes no parameters, thus send 0,0 for data bytes.
SPI_Prop_Send_Cmd( GFX_CMD_NTSC_CLS, 0x00, 0x00);
```

Function Prototype:

```
int SPI_Prop_Print_String(int device, char *string);
```

Description: ***SPI_Prop_Print_String(...)*** is a legacy function that uses terminal NTSC/VGA driver to print strings on the display. Typically, you will use the NTSC/VGA API to print to the terminals, but this was created as a quick and dirty function during testing, you might find it useful. The first parameter is the device you want to print on; **DEVICE_NTSC** or **DEVICE_VGA**, and the second parameter is the ASCII string you want printed. Returns the last byte received from the SPI transmission of all the commands.

Example(s): Print "Hello World!" on both the NTSC and VGA monitors.

```
SPI_Prop_Print_String( DEVICE_NTSC, "Hello world!");
SPI_Prop_Print_String( DEVICE_VGA, "Hello world!");
```

Note: in this case, we are directly using the SPI command function to send commands to the Propeller, so you don't need the NTSC API source files.

20.0 NTSC Library Module Primer

The basic premise of the Chameleon design is that it leverages drivers running on the Propeller chip to do all the media and graphics. Thus, whatever features the particular driver running on the Propeller side is the only features we can access via the AVR side. That doesn't mean we can't abstract functionality and add higher level functions that build on the sub-functions, however, this probably isn't productive since you will want to change drivers, re-write the Propeller driver and so forth. Thus, for the majority of the "media" drivers all the API functions do (including the NTSC) is expose some of the base functionality in nice function calls so you don't have to send SPI messages yourself. In the case of the enhanced NTSC driver (**CHAM_DEFAULT2_DRV_112.SPIN**), we are using the following Propeller Object for the NTSC tile engine and text display:

CHAM_GFX_DRV_001_TB_001.SPIN – NTSC tile engine with support for terminal mode graphics as well as tile graphics, smooth scrolling, color manipulation, large playfields, fixed font 8x8 with 32x24 characters on the screen at one time. In essence, the mode acts like a "console" or terminal, you can print to it, it will scroll when you print the last line, etc. Additionally, it supports more advanced "graphics" like functions for gaming applications, but we will discuss that in the next section.

If your TV/LCD has trouble displaying this driver's output then you can use the other default driver that uses the generic Parallax NTSC tile driver (not as powerful, but more stable signal). The name of that driver is **CHAM_DEFAULT1_DRV_112.SPIN**. Then it loads the following driver for NTSC:

TV_Text_Half_Height_011.spin – NTSC tile engine with 40x30 characters max resolution, limited scrolling, no redefinition of characters.

Hopefully, you can use the standard default2 driver which includes the more advanced graphics tile engine which supports game like applications. But, again, the point is, you can change the Propeller driver and re-write it as you wish, we simply give you a starting point to get going with. So, if you want to know what the driver supports, you can always peek into the driver itself **CHAM_GFX_DRV_001_TB_001.SPIN** and to see the messages that are getting passed to it, you always look into the **CHAM_DEFAULT2_DRV_112.SPIN** driver itself.

With that in mind, if you want to use the NTSC driver then you need the following files added to your project:

CHAM_AVR_NTSC_DRV_V010.c	- Main C file source for "NTSC" module.
CHAM_AVR_NTSC_DRV_V010.h	- Header file for "NTSC" module.

**ARDUINO
TIP**

Arduino tool users you will use the "Import Library" command from the main menu to include **CHAM_AVR_NTSC_DRV_V010**, however all this really does is add the line of code:

```
#include <CHAM_AVR_NTSC_DRV_V010.h>
```

To your program, which you can just do yourself. The Arduino tool already knows where to look for other C/C++ files (since we copied them into the appropriate directories already), thus for the Arduino tool simply add the header include manually or you can let the tool do it from the main menu <Sketch -> Import Library>. The Arduino MAKE file is rather complex and knows how to add all the C/C++ files we dumped into the library directory, so there is little to do with the Arduino tool when accessing these libraries other than including the header in your program.

20.1 Sending Messages to the Propeller Directly

Before we cover the API itself, let's take a look at how we would manually send a message to the Propeller chip and what these "wrapper" functions do. This is the first time we are discussing this since the UART and SPI drivers run on the AVR side of things only, so they are local libraries. On the other hand, the NTSC, VGA, Keyboard, Mouse, Sound, and Propeller I/O Port libraries all make calls over the SPI channel to the Propeller chip. This subtle difference is very important; you do NOT need these libraries, anything you can do with the libraries, you can do directly with the **SPI_Prop_Send_Cmd(...)**, but to make life a little easier, I wrapped many of the commands in function calls for convenience. For example, to print a character to the NTSC screen you can do it directly without the NTSC API like this:

```
SPI_Prop_Send_Cmd( GFX_CMD_NTSC_PRINTCHAR, ch, 0x00);
```

Or you can call the NTSC API function to do it:

```
NTSC_Term_Char(ch);
```

But, let's dive into **NTSC_Term_Char(...)** and take a look:

```
int NTSC_Term_Char(char ch)
{
// this prints a single character to the NTSC terminal (all drivers Default1,2, etc.)
// also supports translation commands supported by the NTSC terminals
//   $00 = clear screen
//   $01 = home
//   $08 = backspace
//   $09 = tab (8 spaces per)
//   $0A = set X position (X follows)
//   $0B = set Y position (Y follows)
//   $0C = set color (color follows)
//   $0D = return
// other characters are not translated, but simply printed to the terminal
SPI_Prop_Send_Cmd( GFX_CMD_NTSC_PRINTCHAR, ch, 0x00);

// wait a bit, driver takes time to respond...
//_delay_us(SPI_PROP_DELAY_SHORT_US);

// return success
return(1);
} // end NTSC_Term_Char
```

As you can see, other than comments, the functionality of the two methods is identical! So, the point is, these API functions are very "thin" wrapper functions that more or less give function names to the commands and make them pretty. Use them if you wish. However, once you have a really nice Propeller driver written you will definitely want to create more advanced API layers on the AVR side, so you can write high level code. But, in this case, most of the functions are 1:1

with the equivalent ***SPI_Prop_Send_Cmd(...)*** code. However, in some cases, the API functions are nice if you want to do a set of operations with a single function call, and that's what APIs are all about.

20.2 Header File Contents Overview

The NTSC API module header **CHAM_NTSC_DRV_V010.h** has no globals or defines as of yet, nothing more than the function prototypes.

20.3 API Listing Reference

The API listing for the "NTSC" module **CHAM_NTSC_DRV_V010.c** is listed in Table 20.1 categorized by functionality.

Table 20.1 – "NTSC" module API functions listing.

Function Name	Description
Initialization	
int NTSC_ClearScreen(void);	Clears the NTSC screen.
Positioning	
int NTSC_GetXY(int *x, int *y);	Gets the current cursor position on the terminal.
int NTSC_SetXY(int x, int y);	Sets the current position on the terminal.
Printing	
int NTSC_Term_Char(char ch);	Prints a single character to the terminal at the current cursor position.
int NTSC_Term_Print(char *string);	Prints a NULL terminated string to the terminal.
int NTSC_Color(int col);	Sets the color of the text.

20.4 API Functional Declarations

The following lists each function, a description, comments, and example usage of the function.

Function Prototype:

```
int NTSC_ClearScreen(void);
```

Description: **NTSC_ClearScreen()** simply clears the NTSC terminal screen and fills the screen with spaces. Returns 1.

Example(s): Clear the screen.

```
NTSC_ClearScreen();
```

Function Prototype:

```
int NTSC_GetXY(int *x, int *y);
```

Description: **NTSC_GetXY(...)** returns the position (column, row) of the virtual cursor. Depending on the driver

you are using, the (x,y) values have different possible ranges. For the Default2 driver they are [0..31, 0..23]. The function takes two pointers to the integer variables you want to receive the values. Returns 1.

Example(s): Retrieve the current cursor position.

```
int x,y;
NTSC_GetXY( &x, &y );
```

Function Prototype:

```
int NTSC_SetXY(int x, int y);
```

Description: *NTSC_SetXY(...)* sets the current cursor position on the NTSC terminal screen. The (x,y) position ranges depend on the driver, but for Default2 they are [0..31, 0..23]. Returns 1.

Example(s): Read the current cursor position and then move the cursor one position to the right.

```
int x,y;
NTSC_GetXY( &x, &y);
NTSC_SetXY( x+1, y);
```

Function Prototype:

```
int NTSC_Term_Char(char ch);
```

Description: *NTSC_Term_Char(...)* prints an ASCII character to the NTSC terminal at the current cursor position, updates the current cursor position, and potentially scroll the text screen up if printing on last line of display. Additionally, this function is a sub-command conduit to the “**terminal manager**”. This functionality should be maintained for any drivers that you develop, so users can have a standard set of base “terminal” functionality. You can send the following ASCII codes (some functions are redundant):

- \$00 = clear the screen.
- \$01 = home the cursor to top left.
- \$08 = backspace.
- \$09 = tab (8 spaces per).
- \$0A = set X position (X follows).
- \$0B = set Y position (Y follows).
- \$0C = set color (color follows).
- \$0D = return (carriage return and linefeed).

Example(s): Clear the screen then print “Chameleon”.

```
NTSC_Term_Char( 0x00 );
NTSC_Term_Char( 'c' );
NTSC_Term_Char( 'h' );
NTSC_Term_Char( 'a' );
NTSC_Term_Char( 'm' );
NTSC_Term_Char( 'e' );
NTSC_Term_Char( 'l' );
NTSC_Term_Char( 'e' );
NTSC_Term_Char( 'o' );
NTSC_Term_Char( 'n' );
```

Function Prototype:

```
int NTSC_Term_Print(char *string);
```

Description: *NTSC_Term_Print(...)* prints a NULL terminated string to the screen with automatic scrolling, etc. if the string prints off the right of the screen or bottom of terminal. Returns 1.

Example(s): Print the string "Hello World!" to screen.

```
NTSC_Term_Print("Hello world!");
```

Function Prototype:

```
int NTSC_Color(int col);
```

Description: *NTSC_Color(...)* sets the color of the text and background (depending on driver). The **col** parameter should be from [0..7] and selects one of a number of different foreground background color schemes for the characters printed. This function has different behavior between the Default1 driver and the Default2 driver. The Default2 driver is more for graphics and gaming, and thus the idea is to control the color more directly with palette manipulation. Returns 1.

Example(s): Print the character 'X' in all colors.

```
int index;
for (index = 0; index < 8; index++)
{
    NTSC_Color( index );
    NTSC_Term_Char( 'X' );
} // end for index
```

21.0 VGA Library Module Primer

The basic premise of the Chameleon design is that it leverages drivers running on the Propeller chip to do all the media and graphics. Thus, whatever features the particular driver running on the Propeller side is the only features we can access via the AVR side. That doesn't mean we can't abstract functionality and add higher level functions that build on the sub-functions, however, this probably isn't productive since you will want to change drivers, re-write the Propeller driver and so forth. Thus, for the majority of the "media" drivers all the API functions do (including the VGA) is expose some of the base functionality in nice function calls so you don't have to send SPI messages yourself. In the case of the enhanced NTSC driver (**CHAM_DEFAULT2_DRV_112.SPIN**), we are using the following Propeller Object for the VGA tile engine and text display:

VGA_Text_010.spin – VGA tile engine with support for terminal mode graphics with a tile resolution of 32x15. Very stable, but limited.

Unlike the NTSC driver, we are using the same VGA terminal driver for both Default1 and Default2 Propeller drivers, so it doesn't matter which you are using the same rules apply. The VGA standard is much higher resolution, thus this driver is much less capable than the NTSC driver. However, you can always use a more advanced VGA driver and modify the Propeller message dispatcher, but we choose this one to get you started since it requires only a single processing core and many of the more advanced VGA drivers require 2 or more cores to generate higher resolutions and capabilities.

But, again, the point is, you can change the Propeller driver and re-write it as you wish, we simply give you a starting point to get going with. So, if you want to know what the driver supports, you can always peek into the driver itself

VGA_Text_010.spin (which calls sub-objects) and to see the messages that are getting passed to it, you always look into the **CHAM_DEFAULT2_DRV_112.SPIN** driver itself.

With that in mind, if you want to use the NTSC driver then you need the following files added to your project:

CHAM_AVR_VGA_DRV_V010.c - Main C file source for "VGA" module.
CHAM_AVR_VGA_DRV_V010.h - Header file for "VGA" module.

ARDUINO TIP

Arduino tool users you will use the "Import Library" command from the main menu to include **CHAM_AVR_VGA_DRV_V010**, however all this really does is add the line of code:

```
#include <CHAM_AVR_VGA_DRV_V010.h>
```

To your program, which you can just do yourself. The Arduino tool already knows where to look for other C/C++ files (since we copied them into the appropriate directories already), thus for the Arduino tool simply add the header include manually or you can let the tool do it from the main menu <Sketch -> Import Library>. The Arduino MAKE file is rather complex and knows how to add all the C/C++ files we dumped into the library directory, so there is little to do with the Arduino tool when accessing these libraries other than including the header in your program.

As you will see the VGA terminal mode functionality is nearly identical to the NTSC. This is by design, so when you are developing "terminal" applications that print out to the NTSC/VGA screens basic information, you don't have to worry about details, each system is like a mini-VT100 terminal with limited abilities, but at least easy to work with.

21.1 Header File Contents Overview

The VGA API module header **CHAM_VGA_DRV_V010.h** has no globals or defines as of yet, nothing more than the function prototypes.

21.2 API Listing Reference

The API listing for the "VGA" module **CHAM_VGA_DRV_V010.c** is listed in Table 21.1 categorized by functionality.

Table 21.1 – "VGA" module API functions listing.

Function Name	Description
Initialization	
int VGA_ClearScreen(void);	Clears the VGA screen.
Positioning	
int VGA_GetXY(int *x, int *y);	Gets the current cursor position on the terminal.
int VGA_SetXY(int x, int y);	Sets the current position on the terminal.
Printing	
int VGA_Term_Char(char ch);	Prints a single character to the terminal at the current cursor position.
int VGA_Term_Print(char *string);	Prints a NULL terminated string to the terminal.
int VGA_Color(int col);	Sets the color of the text.

21.3 API Functional Declarations

The following lists each function, a description, comments, and example usage of the function.

Function Prototype:

```
int VGA_ClearScreen(void);
```

Description: *VGA_ClearScreen()* simply clears the VGA terminal screen and fills the screen with spaces. Returns 1.

Example(s): Clear the screen.

```
VGA_ClearScreen();
```

Function Prototype:

```
int VGA_GetXY(int *x, int *y);
```

Description: *VGA_GetXY(...)* returns the position (column, row) of the virtual cursor. Depending on the driver you are using, the (x,y) values have different possible ranges. For the Default2 driver they are [0..31, 0..23]. The function takes two pointers to the integer variables you want to receive the values. Returns 1.

Example(s): Retrieve the current cursor position.

```
int x,y;
VGA_GetXY( &x, &y );
```

Function Prototype:

```
int VGA_SetXY(int x, int y);
```

Description: *VGA_SetXY(...)* sets the current cursor position on the VGA terminal screen. The (x,y) position ranges depend on the driver, but for Default2 they are [0..31, 0..23]. Returns 1.

Example(s): Read the current cursor position and then move the cursor one position to the right.

```
int x,y;
VGA_GetXY( &x, &y);
VGA_SetXY( x+1, y);
```

Function Prototype:

```
int VGA_Term_Char(char ch);
```

Description: *VGA_Term_Char(...)* prints an ASCII character to the VGA terminal at the current cursor position, updates the current cursor position, and potentially scroll the text screen up if printing on last line of display. Additionally, this function is a sub-command conduit to the "terminal manager". This

functionality should be maintained for any drivers that you develop, so users can have a standard set of base "terminal" functionality. You can send the following ASCII codes (some functions are redundant):

\$00 = clear the screen.
 \$01 = home the cursor to top left.
 \$08 = backspace.
 \$09 = tab (8 spaces per).
 \$0A = set X position (X follows).
 \$0B = set Y position (Y follows).
 \$0C = set color (color follows).
 \$0D = return (carriage return and linefeed).

Example(s): Clear the screen then print "Chameleon".

```
VGA_Term_Char( 0x00 );
VGA _Term_Char( 'c' );
VGA _Term_Char( 'h' );
VGA _Term_Char( 'a' );
VGA _Term_Char( 'm' );
VGA _Term_Char( 'e' );
VGA _Term_Char( 'l' );
VGA _Term_Char( 'e' );
VGA _Term_Char( 'o' );
VGA _Term_Char( 'n' );
```

Function Prototype:

```
int VGA_Term_Print(char *string);
```

Description: *VGA_Term_Print(...)* prints a NULL terminated string to the screen with automatic scrolling, etc. if the string prints off the right of the screen or bottom of terminal. Returns 1.

Example(s): Print the string "Hello World!" to screen.

```
VGA_Term_Print("Hello world!");
```

Function Prototype:

```
int VGA_Color(int col);
```

Description: *VGA_Color(...)* sets the color of the text and background (depending on driver). The *col* parameter should be from [0..7] and selects one of a number of different foreground background color schemes for the characters printed. Returns 1.

Example(s): Print the character 'X' in all colors.

```
int index;
for (index = 0; index < 8; index++)
{
  VGA_Color( index );
  VGA_Term_Char( 'x' );
} // end for index
```

22.0 GFX Library Module Primer

The basic premise of the Chameleon design is that it leverages drivers running on the Propeller chip to do all the media and graphics. Thus, whatever features the particular driver running on the Propeller side is the only features we can access via the AVR side. That doesn't mean we can't abstract functionality and add higher level functions that build on the sub-functions, however, this probably isn't productive since you will want to change drivers, re-write the Propeller driver and so forth. Thus, for the majority of the "media" drivers all the API functions do (including the GFX) is expose some of the base functionality in nice function calls so you don't have to send SPI messages yourself. In the case of the NTSC driver (**CHAM_DEFAULT2_DRV_112.SPIN**), we are using the following Propeller Object for the NTSC tile engine and text display:

CHAM_GFX_DRV_001_TB_001.spin – This is the enhanced NTSC only text/tile engine with support for terminal mode graphics with a tile resolution of 32x24. However, the tile engine has extra features for more graphically oriented tile graphics and gaming applications.

Now, keep in mind the GFX library functions are only supported with the Default2 driver(s) and only work on the NTSC screen. I wrote this driver myself to fit into a single Propeller core, but give you some decent control of the tile maps, bitmaps, palettes, scrolling, and colors. The tile engine has two layers of functionality. The topmost layer is the "console" or "terminal" layer which acts like a simple VT100 terminal, you print to it, it scrolls, you can clear the screen etc. This functionality give you the base abilities to write applications that only need basic text output. Moreover, we selected the NTSC and VGA drivers in both the Default1 and Default2 example drivers to support exactly the same commands. So applications written to drive the VGA terminal work exactly the same on Default1, Default2, or with a simple change of function calls from "VGA" to "NTSC" they will work on the NTSC screen. That said, if you want to do "gaming" tile based graphics, right now, as is, you have to use the Default2 driver **CHAM_DEFAULT2_DRV_112.SPIN**, and only the NTSC screen output is supported.

But, again, the point is, you can change the Propeller driver and re-write it as you wish, we simply give you a starting point to get going with. So, if you want to know what the driver supports, you can always peek into the driver itself **CHAM_GFX_DRV_001_TB_001.spin** and see the messages that are getting passed to it, you always look into the **CHAM_DEFAULT2_DRV_112.SPIN** driver itself.

With that in mind, if you want to use the GFX NTSC driver then you need add both the base NTSC driver along with the GFX API as well to your project:

CHAM_AVR_NTSC_DRV_V010.c	- Main C file source for "NTSC" module.
CHAM_AVR_NTSC_DRV_V010.h	- Header file for "VGA" module.
CHAM_AVR_GFX_DRV_V010.c	- Main C file source for "GFX" module.
CHAM_AVR_GFX_DRV_V010.h	- Header file for "VGA" module.

ARDUINO TIP

Arduino tool users you will use the "Import Library" command from the main menu to include **CHAM_AVR_NTSC_DRV_V010** as well as **CHAM_AVR_GFX_DRV_V010**, however all this really does is add the line of code:

```
#include <CHAM_AVR_NTSC_DRV_V010.h>
#include <CHAM_AVR_GFX_DRV_V010.h>
```

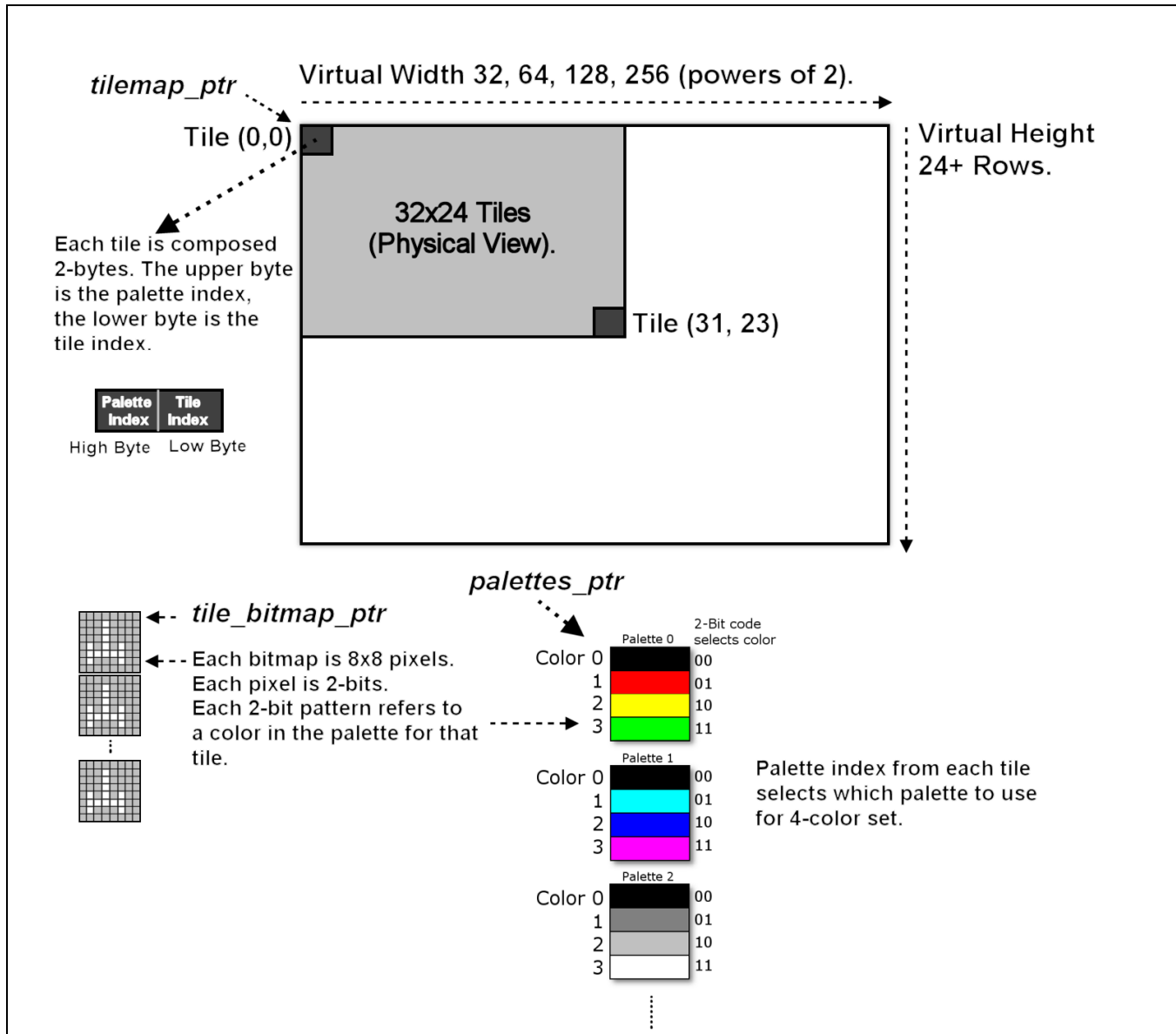
To your program, which you can just do yourself. The Arduino tool already knows where to look for other C/C++ files (since we copied them into the appropriate directories already), thus for the Arduino tool simply add the header include manually or you can let the tool do it from the main menu <Sketch -> Import Library>. The Arduino MAKE file is rather complex and knows how to add all the C/C++ files we dumped into the library directory, so there is little to do with the Arduino tool when accessing these libraries other than including the header in your program.

22.1 GFX Driver Architectural Overview

The idea of the Chameleon is that you use other people's drivers, make your own and change them as you need them on the Propeller side. Thus, we don't want to spend too much time explaining how any particular driver works since you will

probably change it. However, the NTSC GFX API is worth covering for a little, so if you do want to create a game or something more graphical, you can. Please refer to Figure 22.1 for the following paragraphs, I will assume you have some experience in tile mapped graphics and game programming for the following explanations.

Figure 22.1 – GFX Architectural Overview Flow Diagram.



The GFX NTSC tile engine is relatively simple as tiles engines go. The single driver file contains both Spin and ASM language. The driver file name is **CHAM_GFX_DRV_001_TB_001.SPIN**, I highly recommend you sit down and read it line by line if you want to understand how it works. However, it has 2 major parts to it; the SPIN message dispatcher and the ASM code graphics driver. When the driver is started, it loads the ASM code into a Propeller core and that starts drawing graphics on the NTSC screen immediately. Then the SPIN code has a lot of functionality to print characters, strings, numbers, and emulation of the crude VT100 like terminal that all the other NTSC and VGA terminal drivers require. In addition to the terminal functionality, there is a set of "virtual registers" that are used to communicate with the ASM driver with. These registers control everything from the top and bottom overscan, tile map pointers, fine scrolling, etc. We will see some of these constants in the following sections when we look at the header files. But, for now, the whole point is that when the GFX driver is started, the SPIN code launches the ASM driver, it starts immediately drawing the tiles in the 32x24 tile map. Then we can make changes to the tile map using the high level "**terminal**" functions or the low level graphics functions (which is what this section is all about).

Tile Map and Playfield Size

Now, let's take a look at the specifics of this tile map engine. Referring to Figure 22.1, the tile map starts off 32x24 displayed on the NTSC screen. This is the physical resolution and never changes. That is, we can never display more than 32x24 tiles on the physical screen. However, the "playfield" can be much larger to facilitate scrolling or page flipping. The engine supports horizontal tile pitches of 32, 64, 128, and 256. Thus, horizontally, the playfield can be quite large. Vertically, there are no power of 2 restraints, so you can make the vertical pitch anything you want (as long as you don't run out of memory). The tile map itself starts at the declaration **tile_maps** (make sure to take a look yourself) in the driver portion of code and is controlled by changing the global variable **tile_map_base_ptr_parm** in the register interface.

The tile map represents what's on the screen physically and the entire playfield virtually, which might be much larger than the physical screen of 32x24. The tile map pointer above points to the start of the tile map, but how the tiles are interpreted, that is, how many per row is controlled by a register that controls the setup of the tile mode. This variable in the driver is called **tile_map_control_parm** and controls a number of interesting things, here's the initial default setup code for this variable on driver start up:

```
tile_map_control_parm := $AC_AC_00_00 ' format $bc_tc_hv_ww, tc = top overscan color,
                                         ' bc= bottom overscan color,
                                         ' h=horizontal scroll (0..7), v=vertical scroll (0..7),
                                         ' ww = width: 0 = 32 tiles, 1 = 64 tiles, 2 = 128 tiles, 3 = 256 tiles
                                         ' green overscan with $AC
```

So the format of this 4-byte value is

[bottom overscan color | top overscan color | horizontal and vertical smooth scroll | playfield width]

The top and bottom overscan colors (these are the colors on the top and bottom of your TV screen, usually black) are in a simple luma/chroma format which we will discuss in the section below on color palettes. The horizontal and vertical smooth scroll are encoded as 4-bits each, but only vertical smooth scrolling works currently and the valid values are 0..7. Finally, the playfield width is controlled with this register as well. It can be 32, 64, 128, or 256 tiles wide, a subset of powers of 2.

Now, this register is accessed via a number of separate "shadow" registers in the global register interface, but I wanted you to see this in the code, so when we discuss the registers they are familiar.

Tile Map Entry Format

Each tile map entry is a 2-bytes In the following format:

(msb)[palette_index | tile_index](lsb)

The low byte is the actual tile character index 0..255, and the high byte is the palette index 0..255. However, in most cases you won't have this many tiles or palettes. In fact, initially there are only 16 palettes 0..15 and there are 144 bitmaps extracted from file font file **c64_font_05_16b.bmp** followed by 16 blanks for user definition of course you can overwrite these as well with direct memory access if you wish to. The font file was hand made and based on the Commodore 64 and Atari 800 fonts as well as some extra characters for fun. The font file is located on the CD here:

DVD-ROM:\ CHAM_AVR \ TOOLS \ GRAPHICS \ c64_font_05_16b.bmp

Figure 22.2 shows a screen shot of the font file for reference.

Tile Bitmap Format

Figure 22.2 the tile engine font (I added a couple Space Invaders at the end for fun!).



The font bitmap was converted using a tool I wrote which you can find here:

DVD-ROM:\CHAM_AVR\TOOLS\GRAPHICS\BMP2SPIN.EXE

The tool takes as an input a BMP file and outputs SPIN compatible data statements. Read the C source file for the control parameters and to understand how it works. For example, the output of the tool for the "!" character looks like this:

```
' Extracted from file "c64_font_05_16b.bmp" at tile=(1, 0), size=(8, 8), palette index=(1)
c64_font_05__tx1_ty0_bitmap      WORD
WORD  %1_1_1_2_2_1_1_1
WORD  %1_1_1_2_2_1_1_1
WORD  %1_1_1_2_2_1_1_1
WORD  %1_1_1_2_2_1_1_1
WORD  %1_1_1_1_1_1_1_1
WORD  %1_1_1_1_1_1_1_1
WORD  %1_1_1_2_2_1_1_1
WORD  %1_1_1_1_1_1_1_1
```

The tool outputs a text line description of the location and file where the tile was extracted from then encodes the tile in 2-bit pixels where each 2-bit pixel value represents one of four colors from the tile palette to use. Thus, each tile has its own palette of 4 colors. Also, you may notice the funny “%%” character sequence this is how SPIN indicates binary representation, but “%%” means to use 2-bit encoding to simplify your typing thus instead of writing 00_b, 01_b, 10_b, 11_b, we can write 1,2,3,4 – which is convenient. Summing up, each tile is a 8x8 pixels. Each pixel is 2-bits, therefore, each tile bitmap row is a single 16-bit WORD, and there are 8 WORDs that make up a tile. Each 2-bit pixel represents 1 of 4 colors from the palette that is indexed from the tile entry’s high byte. So each tile has its own palette.

TIP

The tile engine is very optimized and thus needs the tile bitmaps in a specific format. That is reversed. So the 2-bit pixels are reversed when output, that is the bitmaps you enter must be mirrored, so that when displayed they are correct. Normally, bitmaps in tile engines might be left to right as they are on the screen, but in this case the left most pixels will display on the right, etc. So keep that in mind when making your own tiles.

The tile bitmaps themselves are defined by the declaration ***_tile_bitmaps*** in the driver and you can change the pointer to them with the global ***tile_bitmaps_base_ptr_parm*** in the register interface. However, this is rare unless you want to point the ***tile_bitmaps_base_ptr_parm*** to another character set. If you move the pointer only one bitmap in length, then it would have the effect of shifting all the character, so “A” would be “B” etc. not very useful.

Palette Entry and Color Format

The palette entries are very simple. Each tile in the tile map character is actually 2-bytes (as discussed above). The low byte is the character which is a direct index into the character bitmaps and the 2nd byte is the color attribute which is an index into the palettes. Thus, a palette entry of 0 means use palette 0, a 1 means use palette 1 and so forth. Each palette consists of 4 colors. Each color encoded as a single byte (which I will discuss the encoding in a moment). Currently the driver has 16 palettes allocated, nothing special, just made up colors, black, white, blue, basic colors kind of thing. You will want to define your own palettes and your own colors as you define your bitmaps, art, and level designs for your games. Below is a peek at the palette definition in the driver:

```
' palettes extracted from file "c64_font_04_8b.bmp" using palette color map 1
```

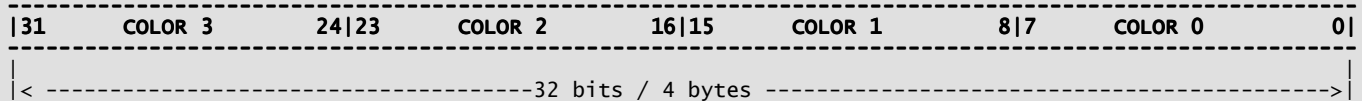
```
' only palette index 0,1 are used right now for character font, palette index 0 makes characters normal
' palette index 1 makes them look inverse video
_tile_palette_map      LONG

c64_font_04__palette_map      LONG
      LONG $07_07_0C_02 ' palette index 0
      LONG $07_0C_07_02 ' palette index 1

      LONG $07_0C_07_02 ' palette index 2
      LONG $07_0C_07_02 ' palette index 3
      LONG $07_0C_07_02 ' palette index 4
      LONG $07_0C_07_02 ' palette index 5
      LONG $07_0C_07_02 ' palette index 6
      LONG $07_0C_07_02 ' palette index 7
      LONG $07_0C_07_02 ' palette index 8
      LONG $07_0C_07_02 ' palette index 9
      LONG $07_0C_07_02 ' palette index 10
      LONG $07_0C_07_02 ' palette index 11
      LONG $07_0C_07_02 ' palette index 12
      LONG $07_0C_07_02 ' palette index 13
      LONG $07_0C_07_02 ' palette index 14
      LONG $07_0C_07_02 ' palette index 15
```

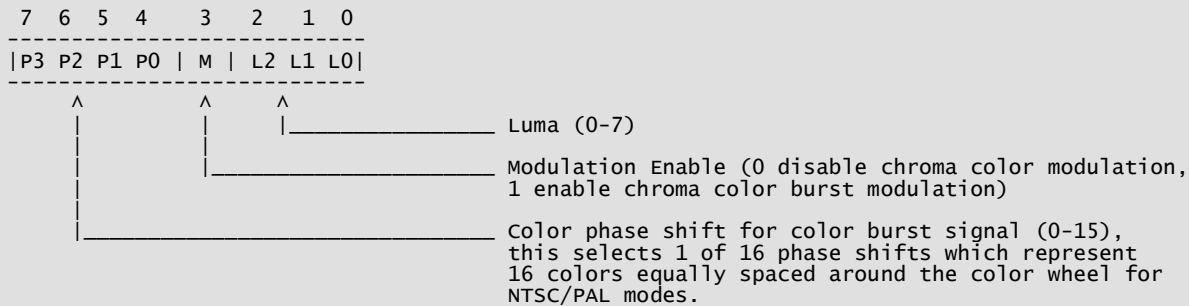
As you can see, the palettes are defined at the label ***_tile_pallette_map*** and each entry is s a LONG. The register you access the palettes from in the global register interface is called ***tile_palettes_base_ptr_parm***. You can read or write this pointer and thus modify palette entries. Now, let's talk about the format of each color.

The color format is always the same, the 32-bits of each palette entry represent 4 possible colors that are "indexed" by the 2-bit pixel data, the colors are encoded as a single byte each in the following way:



Where each color byte is encoded as follows:

Composite Video Color Byte Format



The color byte is used to represent ***everything*** you do with Propeller video, for example; ***sync***, ***black***, ***color burst***, and ***active video***. By setting different bits you can create "colors" that represent these signals. For example, Table 22.1 below shows how to create black, sync, color burst, shades of gray, and colors with any intensity.

Table 22.1 - Color encoding for various color values.

Desired Signal	Bit encoding								
	P3	P2	P1	P0	M	L2	L1	L0	
Sync	0	0	0	0	0	0	0	0	Control values Group
Black	0	0	0	0	0	0	1	0	Grayscale values Group
Dark Grey	0	0	0	0	0	0	1	1	
Grey	0	0	0	0	0	1	0	0	
Light Grey	0	0	0	0	0	1	0	1	

Bright Grey		0	0	0	0	0	1	1	0	
White		0	0	0	0	0	1	1	1	
Blue	0.0'	0	0	0	0	1	x	x	x	(where 011 <= xxx <= 110) (i.e. 3 <= xxx <= 6)
	22.5'	0	0	0	1	1	x	x	x	
Purple	45.0'	0	0	1	0	1	x	x	x	
Magenta	67.5'	0	0	1	1	1	x	x	x	
	90.0'	0	1	0	0	1	x	x	x	
Red	112.5'	0	1	0	1	1	x	x	x	
Orange	135.0'	0	1	1	0	1	x	x	x	
Brown	157.5'	0	1	1	1	1	x	x	x	Color Values Group
Yellow	180.0'	1	0	0	0	1	x	x	x	
Yell/Grn	202.5'	1	0	0	1	1	x	x	x	
Green	225.0'	1	0	1	0	1	x	x	x	
	247.5'	1	0	1	1	1	x	x	x	
	270.0'	1	1	0	0	1	x	x	x	
Cyan	292.5'	1	1	0	1	1	x	x	x	
	315.0'	1	1	1	0	1	x	x	x	
	337.5'	1	1	1	1	1	x	x	x	

NOTE 1: Colors without names are simply linear mixes of the colors before and after.
NOTE 2: When chroma/color is enabled the luma signal is modulated +-1;

Referring to the table, the highest **"safe"** value to use for brightness/luma is 6 and the lowest **"safe"** value is 3, so keep your luma in the range of [3,6] when using chroma, otherwise, the system won't have enough freedom to encode the color signal modulation since a +-1 modulates your luma signal and you don't want it going out of range. The color format is not something I made up, but a direct result of the underlying Propeller video hardware's color format that it needs to process pixels, so we are stuck with it. However, other than the out of range luma values, and the color modulation bit, it's a fairly common way to encode color on 8/16-bit graphics hardware. They rarely use RGB.

Tile Map Review

That about sums up the tile engine design. The physical tile map on the screen always displays 32x24 characters. You can define larger virtual tile maps for scrolling that are 32, 64, 128, 256 wide, any height. Each tile is 2-bytes representing the tile index (character) and the palette to use for the tile (4 colors). Each tile bitmap is 8x8 pixels, each pixel 2-bits. Each 2-bit code refers to 1 of 4 colors in the "palette" for that tile.

The tile engine supports course scrolling by changing the pointer to the start of the current tile map, or page flipping by moving the pointer an entire tile map length. The engine also supports "smooth" scrolling in the vertical direction with the vertical scroll register (0..7). The tile maps, tile bitmaps and palettes are all in memory and can be accessed by reading their base address memory locations and then writing memory. This is done via the register interface (which we will discuss in a moment). Finally, you can control the top and bottom overscan color of the NTSC screen with a 8-bit register for each.

That's about all there is to it, but this tile engine is more than enough to write 90% of the games from the 90's on 8/16-bit machines!!!

22.1.1 GFX Driver Register Interface

Before jumping into the header file contents overview, I want to re-enforce that all communications to the GFX driver are thru a set of "virtual" registers. These registers can be read/written (in most cases) and each register usually has a bit of code that "executes" the read/write and does what needs to be done to any low level variables or data structures in the ASM video driver itself. Thus, much of the functionality of the register interface isn't directly built into the driver, but created via code, the driver, and other added elements to give the user an "interface" that is more robust.

22.2 Header File Contents Overview

Since the GFX driver is a special case, I want to re-enforce that it's worth reviewing the Propeller side driver as well, so you can see what's really going on. We are only going to cover the C/AVR side of things here, but all these registers, functions, API, etc. are all handled at some point on the Propeller running the enhanced graphics driver for NTSC tile graphics. With that in mind, let's crack open the C/C++ header file named: **CHAM_AVR_GFX_DRV_V010.h** and see

what's inside. If you look inside the file, you will see nothing other than a conditional compilation section and the prototypes for the function API, so where are all the register interface defines? Well, remember we have to keep all the SPI commands in a single place, so the register interface are actually message ids for the SPI messaging system! Thus, if you open up **CHAM_AVR_TWI_SPI_DRV_V010.h** and scroll down then you will get to the section with the register interface messages for the GFX driver itself, they are:

```
// advanced GFX commands for GFX tile engine
#define GPU_GFX_BASE_ID 192 // starting id for GFX commands to keep them away from normal command set
#define GPU_GFX_NUM_COMMANDS 37 // number of GFX commands

#define GPU_GFX_SUBFUNC_STATUS_R (0+GPU_GFX_BASE_ID) // Reads the status of the GPU, writes the GPU Sub-Function register and issues a high level
// command like copy, fill, etc.

#define GPU_GFX_SUBFUNC_STATUS_W (1+GPU_GFX_BASE_ID) // Writes status of the GPU, writes the GPU Sub-Function register and issues a high level
// command like copy, fill, etc.

// sub-function constants that are executed when the GPU_GFX_SUBFUNC_STATUS_W command is issued
#define GPU_GFX_SUBFUNC_COPYMEM16 0 // Copies numbytes from src -> dest in wordsize chunks
#define GPU_GFX_SUBFUNC_FILLMEM16 1 // Fills memory with data16, 2 bytes at a time

#define GPU_GFX_SUBFUNC_COPYMEM8 2 // Copies numbytes from src -> dest in byte size chunks
#define GPU_GFX_SUBFUNC_FILLMEM8 3 // Fills memory with low byte of data16, 1 bytes at a time

// normal commands
#define GPU_GFX_TILE_MAP_R (2+GPU_GFX_BASE_ID) // Reads 16-bit tile map ptr which points to the current tile map displayed.
#define GPU_GFX_TILE_MAP_W (3+GPU_GFX_BASE_ID) // Writes 16-bit tile map ptr which points to the current tile map displayed.

#define GPU_GFX_DISPLAY_PTR_R (4+GPU_GFX_BASE_ID) // Reads 16-bit tile map ptr which points to the current display ptr into the tile map, low
// level print functions use this pointer.

#define GPU_GFX_DISPLAY_PTR_W (5+GPU_GFX_BASE_ID) // Writes 16-bit tile map ptr which points to the current display ptr into the tile map, low
// level print functions use this pointer.

#define GPU_GFX_TERM_PTR_R (6+GPU_GFX_BASE_ID) // Reads 16-bit tile map ptr which points to the current terminal ptr into the tile map,
// terminal level print functions use this pointer.

#define GPU_GFX_TERM_PTR_W (7+GPU_GFX_BASE_ID) // Writes 16-bit tile map ptr which points to the current terminal ptr into the tile map,
// terminal level print functions use this pointer.

#define GPU_GFX_BITMAP_R (8+GPU_GFX_BASE_ID) // Reads 16-bit tile bitmaps ptr which points to the bitmaps indexed by the tilemap.
#define GPU_GFX_BITMAP_W (9+GPU_GFX_BASE_ID) // Reads 16-bit tile bitmaps ptr which points to the bitmaps indexed by the tilemap.

#define GPU_GFX_PALETTE_R (10+GPU_GFX_BASE_ID) // Reads 16-bit palette array ptr which points to the palettes in use for the tilemap.
#define GPU_GFX_PALETTE_W (11+GPU_GFX_BASE_ID) // Writes 16-bit palette array ptr which points to the palettes in use for the tilemap.

#define GPU_GFX_TOP_OVERSCAN_COL_R (12+GPU_GFX_BASE_ID) // Reads top overscan color drawn on the screen.
#define GPU_GFX_TOP_OVERSCAN_COL_W (13+GPU_GFX_BASE_ID) // Writes top overscan color drawn on the screen.

#define GPU_GFX_BOT_OVERSCAN_COL_R (14+GPU_GFX_BASE_ID) // Reads top overscan color drawn on the screen.
#define GPU_GFX_BOT_OVERSCAN_COL_W (15+GPU_GFX_BASE_ID) // Writes top overscan color drawn on the screen.

#define GPU_GFX_HSCROLL_FINE_R (16+GPU_GFX_BASE_ID) // Reads current fine horizontal scroll register (0..7) NOTE: (NOT implemented yet)
#define GPU_GFX_HSCROLL_FINE_W (17+GPU_GFX_BASE_ID) // Writes current fine horizontal scroll register (0..7) NOTE: (NOT implemented yet)

#define GPU_GFX_VSCROLL_FINE_R (18+GPU_GFX_BASE_ID) // Reads current fine vertical scroll register (0..7)
#define GPU_GFX_VSCROLL_FINE_W (19+GPU_GFX_BASE_ID) // Writes current fine vertical scroll register (0..7)

#define GPU_GFX_SCREEN_WIDTH_R (20+GPU_GFX_BASE_ID) // Reads screen width value, 0=16 tiles, 1=32 tiles, 2=64 tiles, etc.
#define GPU_GFX_SCREEN_WIDTH_W (21+GPU_GFX_BASE_ID) // Writes screen width value, 0=16 tiles, 1=32 tiles, 2=64 tiles, etc.

#define GPU_GFX_SRC_ADDR_R (22+GPU_GFX_BASE_ID) // Reads 16-bit source address for GPU operations.
#define GPU_GFX_SRC_ADDR_W (23+GPU_GFX_BASE_ID) // Writes 16-bit source address for GPU operations.

#define GPU_GFX_DEST_ADDR_R (24+GPU_GFX_BASE_ID) // Reads 16-bit destination address for GPU operations.
#define GPU_GFX_DEST_ADDR_W (25+GPU_GFX_BASE_ID) // Writes 16-bit destination address for GPU operations.

#define GPU_GFX_NUM_BYTES_R (26+GPU_GFX_BASE_ID) // Reads 16-bit number representing the number of bytes for a GPU operation Sub-Function.
#define GPU_GFX_NUM_BYTES_W (27+GPU_GFX_BASE_ID) // Writes 16-bit number representing the number of bytes for a GPU operation Sub-Function.

#define GPU_GFX_DATA_R (28+GPU_GFX_BASE_ID) // Reads 16-bit data word uses for GPU operations or memory access operations.
#define GPU_GFX_DATA_W (29+GPU_GFX_BASE_ID) // Writes 16-bit data word uses for GPU operations or memory access operations.

#define GPU_GFX_RAM_PORT8_R (30+GPU_GFX_BASE_ID) // Reads 8-bit data pointed to by the currently addressed memory location in the GPU pointed to
// by the src_addr_low|hi.
// After the operation, the src_addr_ptr is incremented as per the GPU configuration.

#define GPU_GFX_RAM_PORT8_W (31+GPU_GFX_BASE_ID) // Writes 8-bit data pointed to by the currently addressed memory location in the GPU pointed
// to by the src_addr_low|hi.
// After the operation, the src_addr_ptr are incremented as per the GPU configuration.

#define GPU_GFX_RAM_PORT16_R (32+GPU_GFX_BASE_ID) // Reads 16-bit data pointed to by the currently addressed memory location in the GPU pointed
// to by the src_addr_low|hi.
// After the operation, the src_addr_ptr is incremented as per the GPU configuration.

#define GPU_GFX_RAM_PORT16_W (33+GPU_GFX_BASE_ID) // Writes 16-bit data pointed to by the currently addressed memory location in the GPU pointed
// to by the src_addr_low|hi.
// After the operation, the src_addr_ptr are incremented as per the GPU configuration.

#define GPU_GFX_RASTER_LINE_R (34+GPU_GFX_BASE_ID) // Reads the current raster line being drawn from 0..191, or whatever the GPU's line resolution
// is set to.

// gpu configuration registers
#define GPU_GFX_SET_AUTO_INC_R (35+GPU_GFX_BASE_ID) // Reads current memory auto increment setting for read/write operations lower 4-bits (0..15),
// default 0

#define GPU_GFX_SET_AUTO_INC_W (36+GPU_GFX_BASE_ID) // Writes the current memory auto increment setting for read/write operations lower 4-bits
// (0..15), default 0
```

Each register is actually a message to the SPI driver, so we cleverly pass these register access requests right thru and tunnel them thru the SPI interface. Now, there are a lot of registers (37 to be exact currently), but they are all more or less self explanatory if you read the comments. To start, we see a couple lines at the top:

```
// advanced GFX commands for GFX tile engine
#define GPU_GFX_BASE_ID 192 // starting id for GFX commands to keep them away from normal command set
#define GPU_GFX_NUM_COMMANDS 37 // number of GFX commands
```

These are important since they give the base offset for these commands 192, and there are 37. So from 192 to (192+37-1) we know are graphics commands. Thus, in the graphics driver, it will get these messages since the SPI message dispatcher is looking for this range. If we open up the Default2 driver, we can see this as shown below:

```
' now process command to determine what user is client is requested via spi link
case ( g_cmd )

    GFX_CMD_NULL:

        ' GFX GPU TILE ENGINE COMMANDS
        //////////////////////////////////////

        ' catch all commands right here
        GPU_GFX_BASE_ID..(GPU_GFX_BASE_ID + GPU_GFX_NUM_COMMANDS - 1):

            ' call single processing function...
            g_spi_result := gfx_ntsc.GPU_GFX_Process_Command( g_cmd, g_data16 )

        ' // NTSC GFX/TILE SPECIFIC COMMANDS
        //////////////////////////////////////
        ' we only expose a subset of the commands the driver supports, you can add/subtract more commands as desired
        ' some commands like PRINTCHAR for example internally support a number of sub-commands that we don't
        ' need to expose at this level unless we want to add functionality
        GFX_CMD_NTSC_PRINTCHAR:
            ' this command pipes right to the out() function of the driver which supports the following
            ' sub-commands already
            '
            '     $00 = clear screen
            '     $01 = home
            '     $08 = backspace
            '     $09 = tab (8 spaces per)
            '     $0A = set X position (X follows)
            '     $0B = set Y position (Y follows)
            '     $0C = set color (color follows)
            '     $0D = return
            '     anything else, prints the character to terminal

            gfx_ntsc.Out_Term( g_data )

        .
        .
        .
```

The yellow highlighted code is in the Default2 SPI message dispatcher, as you can see this highlighted section “catches” all the graphics function calls/messages and pipes them to yet another function call inside

CHAM_GFX_DRV_001_TB_001.SPIN which is the actual graphics driver! Now, let's take a quick peek into that:

```
PUB GPU_GFX_Process_Command( g_cmd, g_data16 ) : g_spi_result
    ' this function can be called with parameters to execute various commands to the register model of the GPU
    ' the SPI client running on the control COG with the virtual SPI interface will typically catch the GPU
    ' command and then pass it along here for efficient processing.

    ' BEGIN REGISTER INTERFACE CASE
    //////////////////////////////////////

    ' process command with proper handler
    case (g_cmd)

        ' GPU
        SUBFUNCTION/STATUS////////////////////////////////////

        GPU_GFX_SUBFUNC_STATUS_R: ' Reads the status of the GPU OR Writes the GPU Sub-Function register and
                                ' issues a high level command like copy, fill, etc.

        GPU_GFX_SUBFUNC_STATUS_W:

            ' extract API subfunction and data
            GPU_GFX_subfunc := g_data16 & $FF ' lower 8-bits defines sub-function
            GPU_GFX_data    := g_data16 >> 8 ' upper 8-bits defines data for sub-function

            case (GPU_GFX_subfunc)

                GPU_GFX_SUBFUNC_COPYMEM16: ' 0 - Copies numbytes from src -> dest in wordsize chunks
                    wordmove( tile_dest_addr_parm, tile_src_addr_parm, tile_numbytes_parm)

                GPU_GFX_SUBFUNC_FILLMEM16: ' 1 - Fills memory with data16, 2 bytes at a time
                    wordfill( tile_dest_addr_parm, tile_data16_parm, tile_numbytes_parm >> 1)
```

```

GPU_GFX_SUBFUNC_COPYMEM8: ' 2 - Copies numbytes from src -> dest in byte size chunks
                           bytemove( tile_dest_addr_parm, tile_src_addr_parm, tile_numbytes_parm >> 1)

GPU_GFX_SUBFUNC_FILLMEM8: ' 3 - Fills memory with low byte of data16, 1 bytes at a time
                           bytefill( tile_dest_addr_parm, tile_data16_parm & $FF, tile_numbytes_parm)

' END CASE GPU_GFX_subfunc
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
' TILE MAP POINTER
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

GPU_GFX_TILE_MAP_R:      ' Reads or writes the 16-bit tile map ptr which points to the current tile map
                           ' displayed.
    g_spi_result := tile_map_base_ptr_parm

GPU_GFX_TILE_MAP_W:
    tile_map_base_ptr_parm := g_data16

' TILE DISPLAY POINTER
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

GPU_GFX_DISPLAY_PTR_R:   ' Reads or writes the 16-bit tile display ptr which points to the location
                           ' where low level printing is displayed
    g_spi_result := tile_display_ptr

GPU_GFX_DISPLAY_PTR_W:
    tile_display_ptr := g_data16

' TILE TERMINAL POINTER
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

GPU_GFX_TERM_PTR_R:      ' Reads or writes the 16-bit tile terminal ptr which points to the location where
                           ' high level terminal mode printing is displayed
    g_spi_result := tile_term_ptr

GPU_GFX_TERM_PTR_W:
    tile_term_ptr := g_data16

' BITMAP POINTER
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

GPU_GFX_BITMAP_R:        ' Reads or writes the low byte of the 16-bit tile bitmaps ptr which points to the
                           ' bitmaps indexed by the tilemap.
    g_spi_result := tile_bitmaps_base_ptr_parm

GPU_GFX_BITMAP_W:
    tile_bitmaps_base_ptr_parm := g_data16
.
.
.

```

I have highlighted the register access code fragments at the top of each section. So the idea is the AVR/PIC sends the GFX command message via SPI, the SPI message dispatcher catches these messages and passes all of them to the function, then this function, cases on each possible register access message and processes it – viola! Of course, the above code is a partial listing, the complete function is really long and you can look it yourself in the Propeller source file.

GPU Functions Overview

These functions deserve a bit of explanation. I am using the word “GPU” pretty freely here and its not meant to infer there is a hidden GPU somewhere, but rather than indicate that the hardware or driver is capable of doing large operations via a single command. Two things that we do a lot of in computer graphics is copying data and moving data. Copy, move, copy, move, copy, move. 90% of operations are copy, move! So, one thing you want you GPU/graphics engine, software, whatever to do is copy and move – **fast!**

Thus, one of the simple additions to the tile driver is the ability to copy and fill memory. So, I added some sub-functions that I call “GPU functions” these are macro level functions that can fill and copy memory 8/16 bits at a time. If you take a look at the listing above at the very top you will see the processing of GPU sub-functions. First, we use the SPI command itself to identify a GPU function, the register messages are **GPU_GFX_SUBFUNC_STATUS_R** and **GPU_GFX_SUBFUNC_STATUS_W**. The “R” or read message currently does nothing, but the “W” or write message executes the GPU command which there are currently 4 of as shown in the header code below:

```

' extract API subfunction and data
GPU_GFX_subfunc := g_data16 & $FF ' lower 8-bits defines sub-function
GPU_GFX_data    := g_data16 >> 8  ' upper 8-bits defines data for sub-function

case (GPU_GFX_subfunc)

```

```

GPU_GFX_SUBFUNC_COPYMEM16: ' 0 - Copies numbytes from src -> dest in wordsize chunks
wordmove( tile_dest_addr_parm, tile_src_addr_parm, tile_numbytes_parm)

GPU_GFX_SUBFUNC_FILLMEM16: ' 1 - Fills memory with data16, 2 bytes at a time
wordfill( tile_dest_addr_parm, tile_data16_parm, tile_numbytes_parm >> 1)

GPU_GFX_SUBFUNC_COPYMEM8: ' 2 - Copies numbytes from src -> dest in byte size chunks
bytemove( tile_dest_addr_parm, tile_src_addr_parm, tile_numbytes_parm >> 1)

GPU_GFX_SUBFUNC_FILLMEM8: ' 3 - Fills memory with low byte of data16, 1 bytes at a time
bytefill( tile_dest_addr_parm, tile_data16_parm & $FF, tile_numbytes_parm)

```

We will see how to call all these in the API section below, but the point is, I want you to see the “big picture” and see how all this stuff works, not so much, so you can use it, but so that you might use this template, model, or pattern to create better, faster drivers. In any event, when a message is sent with **GPU_GFX_SUBFUNC_STATUS_W** then we assume that a GPU command is being issued and the first data byte of the SPI message is used as the sub-function ID (lower 8-bits) and the upper 8-bits or second byte is used as the operand or data.

Now, the GPU functions are too complex to request in a single SPI message, so we have to “set them up”. Thus, we have to initialize pointers to the source, destination, and length of bytes or words to process then we call the GPU command to actually do the work. For example, the sub-function **GPU_GFX_SUBFUNC_FILLMEM16** requires a pointer to the destination and the number of WORDS to write along with data WORD. So how do we set all those? Easy, we use other GFX messages to set the destination address pointer, data WORD, and number of bytes for “operation”, then the GPU sub-function uses those data for the operation! Simple!

At this point, you should have a good idea of how the enhanced NTSC tile engine works, and the entire communication paths from the AVR/PIC to the Propeller driver itself. So, let’s move onto the API itself.

```

/* default baud rate */
#define UART_DEFAULT_BAUDRATE      115200 // slow down to 57600, 38400 if you have trouble

// transmit and receive buffer constants
#define UART_TX_BUFF_SIZE          32
#define UART_RX_BUFF_SIZE          32

```

The #defines are trivial; simply a constant for a convenient baud rate as well as the size of the circular buffers. You can make these any size you wish, but memory is at a premium, so keep it in mind. If you are going to be using a lot of serial communications and know you are going to send large strings or receive large strings and not be able to interrogate the buffer functions then you might want to increase the buffers to 64, 128, 256 etc. But, for now 32/32 seems to work find in all examples.

Next are all the global data structures and variables used by the API functions:

```

// global that holds UART current baud rate
extern int g_baudrate;

// receiver and transmitter head and tail indices for circular buffers
extern int g_uart_rx_head;
extern int g_uart_rx_tail;
extern int g_uart_tx_head;
extern int g_uart_tx_tail;

// buffers for uart interrupt driver
extern UCHAR g_uart_buffer_rx[UART_RX_BUFF_SIZE]; // receiver buffer for incoming characters
extern UCHAR g_uart_buffer_tx[UART_RX_BUFF_SIZE]; // transmit buffer for outgoing characters

```

First we have a global baudrate, pointers to the circular buffers, and finally the buffers themselves. These are all declared in the C file of course, these are simply the external references.

22.3 API Listing Reference

The API listing for the “NTSC GFX” driver module **CHAM_GFX_DRV_V010.c** is listed in Table 22.2 categorized by functionality.

Table 22.2 – “GFX” module API functions listing.

Function Name	Description
GPU sub-functions	
int GFX_GPU_Fill_Mem16(unsigned int dest_ptr, unsigned int value, unsigned int num_bytes);	Fills memory 16-bits at a time.
int GFX_GPU_Copy_Mem16(unsigned int dest_ptr, unsigned int src_ptr, unsigned int num_bytes);	Copies memory 16-bits at a time.
int GFX_GPU_Fill_Mem8(unsigned int dest_ptr, unsigned int value, unsigned int num_bytes);	Fills memory 8-bits at a time.
int GFX_GPU_Copy_Mem8(unsigned int dest_ptr, unsigned int src_ptr, unsigned int num_bytes);	Copies memory 8-bits at a time.
Palette pointer access functions	
unsigned int GFX_Read_Palette_Ptr(void);	Reads the tile map palette pointer.
unsigned int GFX_Write_Palette_Ptr(unsigned int palette_ptr);	Writes the tile map palette pointer.
Tilemap pointer access functions	
unsigned int GFX_Read_Tilemap_Ptr(void);	Reads the pointer to the top of tilemaps.
unsigned int GFX_Write_Tilemap_Ptr(unsigned int tilemap_ptr);	Writes the pointer to the top of tilemaps.
Bitmap pointer access functions	
unsigned int GFX_Read_Bitmap_Ptr(void);	Reads the pointer to tile bitmaps.
unsigned int GFX_Write_Bitmap_Ptr(unsigned int bitmap_ptr);	Writes the pointer to the tile bitmaps.
Scrolling functions	
int GFX_Get_HScroll(void);	Not implemented.
int GFX_Get_VScroll(void);	Retrieves the vertical smooth scroll value.
int GFX_Set_HScroll(void);	Not implemented yet.
int GFX_Set_VScroll(int vscroll);	Sets the vertical smooth scroll value.
Memory port functions	
unsigned int GFX_Read_Mem_Port8(void);	Reads Propeller memory 8-bit port.
int GFX_Write_Mem_Port8(unsigned int data);	Writes to the Propeller memory 8-bit port.
unsigned int GFX_Read_Mem_Port16(void);	Reads the Propeller memory 16-bit port.
unsigned int GFX_Write_Mem_Port16(unsigned int data);	Writes the Propeller memory 16-bit port.
Source and destination pointer access	
unsigned int GFX_Get_Src_Ptr(void);	Retrieves the “source” GPU operation pointer.
unsigned int GFX_Get_Dest_Ptr(void);	Retrieves the “destination” GPU operation pointer.
unsigned int GFX_Set_Src_Ptr(unsigned int src_ptr);	Sets the “source” GPU operation pointer.
unsigned int GFX_Set_Dest_Ptr(unsigned int dest_ptr);	Sets the “destination” GPU operation pointer.
Tilemap width functions	
int GFX_Get_Tilemap_Width(void);	Retrieves the width of the tilemap in encoded format.
int GFX_Set_Tilemap_Width(int width);	Sets the width of the tilemap in encoded format.
Top and bottom overscan control functions	
int GFX_Get_Bottom_Overscan_Color(void);	Retrieves the bottom overscan color.
int GFX_Get_Top_Overscan_Color(void);	Retrieves the top overscan color.

int GFX_Set_Bottom_Overscan_Color(int col);	Sets the bottom overscan color.
int GFX_Set_Top_Overscan_Color(int col);	Sets the top overscan color.

22.4 API Functional Declarations

The following lists each function, a description, comments, and example usage of the function.

Function Prototype:

```
int GFX_GPU_Fill_Mem16(unsigned int dest_ptr, unsigned int value, unsigned int num_bytes);
```

Description: **GFX_GPU_Fill_Mem16(...)** copies a number of bytes from source to destination in the GPU memory space in WORD side chunks, or in other words in the Prop's local memory of 32K, you can use this to overwrite SPIN code, so watch out! The parameters are the destination address of the fill **dest_ptr**, the 16-bit data **value**, and finally, the number of bytes (always a multiple of 2) **num_bytes**. Typically, you will use this function to clear tile map, bitmap memory, and other repetitive operations that require the same value to be written. Returns 1.

Example(s): Fill 100 WORDs starting at address \$3456 with the value \$AAFF.

```
GFX_GPU_Fill_Mem16( 0x3456, 0xAAFF, 100*2 );
```

Function Prototype:

```
int GFX_GPU_Copy_Mem16(unsigned int dest_ptr, unsigned int src_ptr, unsigned int num_bytes);
```

Description: **GFX_GPU_Copy_Mem16(...)** copies a number of bytes from source to destination in the GPU memory space in WORD side chunks, or in other words in the Prop's local memory of 32K, you can use this to overwrite SPIN code, so watch out! The parameters are the destination address to copy to **dest_ptr**, the source address to copy from **src_ptr**, and finally, the number of bytes (always a multiple of 2) **num_bytes** to copy. Typically, you will use this function to copy tile maps, scroll, page flip, perform various animations, etc. Returns 1.

Example(s): Assuming that the current tile map is 32x24 physically and virtually, and there are two tile maps, one located at \$1000 and the other at \$2000, copy the later to the former.

```
GFX_GPU_Copy_Mem16( 0x1000, 0x2000, 32*24*2 );
```

Function Prototype:

```
int GFX_GPU_Fill_Mem8(unsigned int dest_ptr, unsigned int value, unsigned int num_bytes);
```

Description: **GFX_GPU_Fill_Mem8(...)** copies a number of bytes from source to destination in the GPU memory space in BYTE side chunks, or in other words in the Prop's local memory of 32K, you can use this to overwrite SPIN code, so watch out! The parameters are the destination address of the fill **dest_ptr**, the 8-bit data **value**, and finally, the number of bytes **num_bytes**. Typically, you will use this function to clear tile map, bitmap memory, and other repetitive operations that require the same value to be written. Returns 1.

Example(s): Fill 100 BYTES starting at address \$3456 with the value \$AF.

```
GFX_GPU_Fill_Mem8( 0x3456, 0xAF, 100 );
```

Function Prototype:

```
int GFX_GPU_Copy_Mem8(unsigned int dest_ptr, unsigned int src_ptr, unsigned int num_bytes);
```

Description: ***GFX_GPU_Copy_Mem8(...)*** copies a number of bytes from source to destination in the GPU memory space in BYTE side chunks, or in other words in the Prop's local memory of 32K, you can use this to overwrite SPIN code, so watch out! The parameters are the destination address to copy to **dest_ptr**, the source address to copy from **src_ptr**, and finally, the number of bytes **num_bytes** to copy. Typically, you will use this function to copy tile maps, scroll, page flip, perform various animations, etc. Returns 1.

Example(s): Assuming that the current tile map is 32x24 physically and virtually, and there are two tile maps, one located at \$1000 and the other at \$2000, copy the later to the former.

```
GFX_GPU_Copy_Mem8( 0x1000, 0x2000, 32*24*2 );
```

Function Prototype:

```
unsigned int GFX_Read_Palette_Ptr(void);
```

Description: ***GFX_Read_Palette_Ptr()*** reads the 16-bit palette pointer that points to the set of palettes referred to by the palette index in each tile entry. This pointer is of course relative to the Propeller's 32K memory space and an absolute address. Each palette entry is a single 32-bit long that represents 4 colors in the format: **[color 3 | color 2 | color 1 | color 0]** that are represented by the bitmaps as each 2-bit pair indexing as color 0,1,2,3. Returns the 16-bit pointer value/address.

Example(s): Read the palette pointer.

```
unsigned int palette_ptr_prop;
palette_ptr_prop = GFX_Read_Palette_Ptr();
```

Function Prototype:

```
unsigned int GFX_Write_Palette_Ptr(unsigned int palette_ptr);
```

Description: ***GFX_Write_Palette_Ptr(...)*** writes the 16-bit palette pointer **palette_ptr** that points to the set of palettes referred to by the palette index in each tile entry each palette entry is a single 32-bit long that represents 4 colors **[color 3 | color 2 | color 1 | color 0]** that are represented by the bitmaps as each 2-bit pair indexing as color 0,1,2,3. When this operation is performed, the palettes pointed to must be valid colors, otherwise sync issues could occur since the color words can represent both color and sync. So make sure that when you do alter the palette pointer you have built up value palette entries and that all potential palette indices in your tile map are represented by a palette. Returns 1.

Example(s): Read the palette pointer and then shift it by 4-bytes, in essence by a single palette of 4 colors down the list.

```
GFX_Write_Palatte_Ptr( GFX_Read_Palette_Ptr() + 4 );
```

Function Prototype:

```
unsigned int GFX_Read_Tilemap_Ptr(void);
```

Description: ***GFX_Read_Tilemap_Ptr()*** reads the 16-bit tilemap pointer that points to the current tile map data displayed. Each tile map entry is 2 bytes [**palette index** | **character index**], and the current tile map width decides the pitch per row. The tile map pointer is of course relative to the Propeller's 32K memory space and an absolute address. Returns the 16-bit pointer value/address.

Example(s): Read the tile map pointer.

```
unsigned int tilemap_ptr;
tilemap_ptr = GFX_Read_Tilemap_Ptr() ;
```

Function Prototype:

```
unsigned int GFX_Write_Tilemap_Ptr(unsigned int tilemap_ptr);
```

Description: ***GFX_Write_Tilemap_Ptr(...)*** writes the 16-bit tilemap pointer that points to the current tile map data displayed. Each tile map entry is 2 bytes [**palette index** | **character index**], and the current tile map width decides the pitch. The pointer must be a valid address and within open memory or part of memory declared for the driver itself, otherwise you could crash the Propeller, so watch out. Scrolling, page flipping, and other memory effects can be achieved by changing the pointer. For example, say you want to scroll the tilemap one row down? Assuming the current tilemap is 32x24 then we know there are 32 tiles per row and each tile is 2-bytes, thus 64 bytes per row. Therefore to scroll down a single row, we simply need to add 64 to the current tile map pointer. The example that follows will show this. Returns 1.

Example(s): Scroll down a line. Assumes 32x24 tile map (important thing is the row pitch, this defines how many bytes per row).

```
GFX_Write_Tilemap_Ptr( GFX_Read_Tilemap_Ptr() + 32*2 ) ;
```

Function Prototype:

```
unsigned int GFX_Read_Bitmap_Ptr(void);
```

Description: ***GFX_Read_Bitmap_Ptr()*** reads the 16-bit bitmap pointer that points to the first bitmap indexed by tile 0. Each bitmap is 8x8 pixels, 2-bits per pixel, 16-bytes each. The bitmap pointer is of course relative to the Propeller's 32K memory space and an absolute address. Returns the 16-bit pointer value/address.

Example(s): Read the current bitmap pointer.

```
unsigned int bitmap_ptr;
bitmap_ptr = GFX_Read_Bitmap_Ptr() ;
```

Function Prototype:

```
unsigned int GFX_Write_Bitmap_Ptr(unsigned int bitmap_ptr);
```

Description: ***GFX_Write_Bitmap_Ptr(...)*** writes the 16-bit bitmap pointer ***bitmap_ptr*** that points to the first bitmap indexed by tile 0. Each bitmap is 8x8 pixels, 2-bits per pixel, 16-bytes each. The pointer must be a valid address and within open memory or part of memory declared for the driver itself, otherwise you could crash the Propeller, so watch out. Typically, you will adjust the bitmap pointer by one or more bitmaps to create animation effects of the tiles on screen or to swap out character sets on the fly. Returns 1.

Example(s): Adjust the bitmap pointer down by 16 bitmap tiles (16 * 16 = 256 bytes).

```
GFX_Write_Bitmap_Ptr( GFX_Read_Bitmap_Ptr() + 256 );
```

Function Prototype:

```
int GFX_Get_VScroll(void);
```

Description: ***GFX_Get_VScroll()*** returns the current fine vertical scroll value from 0..7. The tile engine supports smooth vertical scrolling via the vertical fine scroll register. This function reads that register.

Example(s): Read the vertical scroll register.

```
int fine_scroll ;
fine_scroll = GFX_Get_Vscroll() ;
```

Function Prototype:

```
int GFX_Set_VScroll(int vscroll);
```

Description: ***GFX_Set_VScroll(...)*** sets the vertical fine scroll register with the sent value ***vscroll*** (0..7). Use this function and register to control smooth scrolling. Each tile is 8 pixels high, thus you can scroll the tilemap up and down a single pixel at a time with this register/call. To perform continuous smooth scrolling, first fine scroll 0..7 then reset to 0, and course scroll a single row. Returns 1.

Example(s): Fine scroll 8 times in about a second from 0..7, a tile height.

```
for (int scroll = 0; scroll < 8; scroll++)
{
    GFX_Set_VScroll( scroll );
    delay_ms(100);
} // end for scroll
```

Function Prototype:

```
unsigned int GFX_Read_Mem_Port8(void);
```

Description: ***GFX_Read_Mem_Port8()*** reads a 8-bit value from **prop_memory[source_ptr]**, note source pointer is always used for memory port operations (read or write). In the Propeller driver **source_ptr** is the variable **tile_src_addr_parm** and is set using ***GFX_Set_Src_Ptr(unsigned int src_ptr)*** defined below. The idea of this function is that it provides a simple mechanism to access the Propeller's RAM. Returns the 8-bit value read directly from the Propeller's 32K RAM. This function also updates the source pointer automatically anticipating another read or write. The actual code looks like:

```
' update src pointer based on auto increment value
tile_src_addr_parm += tile_mem_autoinc_parm
```

Thus, the variable **tile_mem_autoinc_parm** is used as the memory increment stride each read/write operation. This comes in handy and reduces address updates when you access continuous memory. There is of course a function to change **tile_mem_autoinc_parm**.

Example(s): Read the byte of memory currently pointed to by the source memory pointer.

```
data = GFX_Read_Mem_Port8() ;
```

Function Prototype:

```
int GFX_Write_Mem_Port8(unsigned int data);
```

Description: ***GFX_Write_Mem_Port8()*** writes a 8-bit value to **prop_memory[source_ptr] = data**, note source pointer is always used for memory port operations (read or write). In the Propeller driver **source_ptr** is the variable **tile_src_addr_parm** and is set using ***GFX_Set_Src_Ptr(unsigned int src_ptr)*** defined below. The idea of this function is that it provides a simple mechanism to access the Propeller's RAM. This function also updates the source pointer automatically anticipating another read or write. The actual code looks like:

```
' update src pointer based on auto increment value
tile_src_addr_parm += tile_mem_autoinc_parm
```

Thus, the variable **tile_mem_autoinc_parm** is used as the memory increment stride each read/write operation. This comes in handy and reduces address updates when you access continuous memory. There is of course a function to change **tile_mem_autoinc_parm**.

Finally, the function always returns 1.

Example(s): Write \$FF to the byte of memory currently pointed to by the source memory pointer.

```
GFX_Write_Mem_Port8( 0xFF ) ;
```

Function Prototype:

```
unsigned int GFX_Read_Mem_Port16(void);
```

Description: ***GFX_Read_Mem_Port16()*** reads a 16-bit value from **prop_memory[source_ptr]**, note source pointer is always used for memory port operations. In the Propeller driver **source_ptr** is the variable **tile_src_addr_parm** and is set using ***GFX_Set_Src_Ptr(unsigned int src_ptr)*** defined below. The idea of this function is that it provides a simple mechanism to access the Propeller's RAM. Returns the 16-bit

value read directly from the Propeller's 32K RAM. This function also updates the source pointer automatically anticipating another read or write. The actual code looks like:

```
' update src pointer based on auto increment value
tile_src_addr_parm += tile_mem_autoinc_parm
```

Thus, the variable ***tile_mem_autoinc_parm*** is used as the memory increment stride each read/write operation. This comes in handy and reduces address updates when you access continuous memory. There is of course a function to change ***tile_mem_autoinc_parm***.

Example(s): Read the 16-bit WORD of memory currently pointed to by the source memory pointer.

```
data16 = GFX_Read_Mem_Port16() ;
```

Function Prototype:

```
unsigned int GFX_Write_Mem_Port16(unsigned int data);
```

Description: ***GFX_Write_Mem_Port16()*** writes a 16-bit value to **prop_memory[source_ptr] = data**, note source pointer is always used for memory port operations (read or write). In the Propeller driver ***source_ptr*** is the variable ***tile_src_addr_parm*** and is set using ***GFX_Set_Src_Ptr(unsigned int src_ptr)*** defined below. The idea of this function is that it provides a simple mechanism to access the Propeller's RAM. This function also updates the source pointer automatically anticipating another read or write. The actual code looks like:

```
' update src pointer based on auto increment value
tile_src_addr_parm += tile_mem_autoinc_parm
```

Thus, the variable ***tile_mem_autoinc_parm*** is used as the memory increment stride each read/write operation. This comes in handy and reduces address updates when you access continuous memory. There is of course a function to change ***tile_mem_autoinc_parm***.

Finally, the function always returns 1.

Example(s): Write \$1234 to the 16-bit WORD of memory currently pointed to by the source memory pointer.

```
GFX_Write_Mem_Port16( 0x1234 ) ;
```

Function Prototype:

```
unsigned int GFX_Get_Src_Ptr(void);
```

Description: ***GFX_Get_Src_Ptr()*** retrieves the 16-bit source pointer in the tile GPU for memory transfer and fill operations. This address is absolute in the Propeller's main RAM memory. Returns the pointer.

Example(s): Retrieve the source pointer.

```
unsigned int src_ptr;
src_ptr = GFX_Get_Src_Ptr();
```

Function Prototype:

```
unsigned int GFX_Get_Dest_Ptr(void);
```

Description: ***GFX_Get_Dest_Ptr()*** retrieves the 16-bit destination pointer in the tile GPU for memory transfer and fill operations. This address is absolute in the Propeller's main RAM memory. Returns the pointer.

Example(s): Retrieve the destination pointer.

```
unsigned int dest_ptr;
dest_ptr = GFX_Get_Dest_Ptr();
```

Function Prototype:

```
unsigned int GFX_Set_Src_Ptr(unsigned int src_ptr);
```

Description: ***GFX_Set_Src_Ptr(...)*** sets the 16-bit source pointer for tile GPU fill and memory operations. This is an absolute address in Propeller RAM memory. Returns 1.

Example(s): Read the source pointer and add 4 to it and write it back.

```
GFX_Set_Src_Ptr( GFX_Get_Src_Ptr() + 4);
```

Function Prototype:

```
unsigned int GFX_Set_Dest_Ptr(unsigned int dest_ptr);
```

Description: ***GFX_Set_Dest_Ptr(...)*** sets the 16-bit destination pointer for tile GPU fill and memory operations. This is an absolute address in Propeller RAM memory. Returns 1.

Example(s): Read the destination pointer and add 4 to it and write it back.

```
GFX_Set_Dest_Ptr( GFX_Get_Dest_Ptr() + 4);
```

Function Prototype:

```
int GFX_Get_Tilemap_Width(void);
```

Description: ***GFX_Get_Tilemap_Width()*** retrieves the 8-bit the tile map width encoded as 0,1,2,3 which means (32, 64, 128, 256) tiles.

Example(s): Read the tilemap width and store it.

```
int tilemap_width = GFX_Get_Tilemap_width();
```

Function Prototype:

```
int GFX_Set_Tilemap_Width(int width);
```

Description: **GFX_Set_Tilemap_Width(...)** sets the width of the tile map to 32, 64, 128 or 256 which are encoded as 0,1,2,3 with the parameter **width**. The idea of this function call is to set the virtual playfield of the tilemap to large than its physical size which is always 32 tiles. Doing so allows for horizontal scrolling (course). Returns 1.

Example(s): Set the tilemap width to 128 tiles which is 4 complete screenfull's of tiles for horizontal scrolling.

```
GFX_Set_Tilemap_width( 2 );
```

Function Prototype:

```
int GFX_Get_Bottom_Overscan_Color(void);
```

Description: **GFX_Get_Bottom_Overscan_Color()** retrieves the bottom overscan color of the NTSC driver. The bottom overscan is a few lines at the bottom of the screen that no tiles are drawn into. Usually this area is black on normal broadcasts of TV programs, but the driver allows you to change the color. The returned color value will be in Propeller color format [**chroma 4-bit | chroma enable 1-bit | luma 3-bit**].

Example(s): Retrieve the current bottom overscan color.

```
int bottom_overscan_color = GFX_Get_Bottom_Overscan_color();
```

Function Prototype:

```
int GFX_Get_Top_Overscan_Color(void);
```

Description: **GFX_Get_Top_Overscan_Color()** retrieves the top overscan color of the NTSC driver. The top overscan is a few lines at the top of the screen that no tiles are drawn into. Usually this area is black on normal broadcasts of TV programs, but the driver allows you to change the color. The returned color value will be in Propeller color format [**chroma 4-bit | chroma enable 1-bit | luma 3-bit**].

Example(s): Retrieve the current top overscan color.

```
int top_overscan_color = GFX_Get_Top_Overscan_color();
```

Function Prototype:

```
int GFX_Set_Bottom_Overscan_Color(int col);
```

Description: **GFX_Set_Bottom_Overscan_Color(...)** sets the bottom overscan color to **col** where **col** is in normal

Propeller color format [**chroma 4-bit | chroma enable 1-bit | luma 3-bit**]. Returns 1.

Example(s): Read the current color and increment the chroma portion of the color (upper 4 bits).

```
// retrieve the color
int color = GFX_Get_Bottom_Overscan_Color();

// now add 1 to the upper 4 bit nibble, allow wrap around rather than clamp on color so 15 -> 0
color = color + (1 << 4);

// finally write it back
GFX_Set_Bottom_Overscan_Color( color );
```

Function Prototype:

```
int GFX_Set_Top_Overscan_Color(int col);
```

Description: **GFX_Set_Top_Overscan_Color(...)** sets the top overscan color to **col** where **col** is in normal Propeller color format [**chroma 4-bit | chroma enable 1-bit | luma 3-bit**]. Returns 1.

Example(s): Read the current color and increment the chroma portion of the color (upper 4 bits).

```
// retrieve the color
int color = GFX_Get_Bottom_Overscan_Color();

// now add 1 to the upper 4 bit nibble, allow wrap around rather than clamp on color so 15 -> 0
color = color + (1 << 4);

// finally write it back
GFX_Set_Bottom_Overscan_Color( color );
```

23.0 Sound Library Module Primer

The basic premise of the Chameleon design is that it leverages drivers running on the Propeller chip to do all the media and graphics. Thus, whatever features the particular driver running on the Propeller side is the only features we can access via the AVR side. That doesn't mean we can't abstract functionality and add higher level functions that build on the sub-functions, however, this probably isn't productive since you will want to change drivers, re-write the Propeller driver and so forth. Thus, for the majority of the "media" drivers all the API functions do (including the sound) is expose some of the base functionality in nice function calls so you don't have to send SPI messages yourself.

For sound, we decided to re-use a driver from the HYDRA development that was designed for both PWM and PCM (samples) with multiple channels. The same sound driver is used in both Default1 and Default2, so the same idea apply to either driver you load on the Propeller (**CHAM_DEFAULT1_DRV_112.SPIN** or **CHAM_DEFAULT2_DRV_112.SPIN**). The actual driver object used for sound is:

NS_sound_drv_052_11khz_16bit.spin – This is an 11Khz, 16bit sound driver with 9 independent audio channels! It's pretty cool.

However, I only exposed a portion of its abilities thru the current SPI messages since sound is something that you will probably want either a little or a lot, no need to waste messages. You can always add more. Therefore, the abilities you see exposed in the following API are just a taste of what it can do. For example, to save commands, I mashed sound commands into a single SPI packet, so you only can control 4 channels with limited frequency and control. In other words, if you need more than simple 4 channel sound, then you will have to add messages to the SPI driver and add them yourself.

With that in mind, if you want to use the sound driver then you need the following files added to your project:

CHAM_AVR_SOUND_DRV_V010.c - Main C file source for "Sound" module.
CHAM_AVR_SOUND_DRV_V010.h - Header file for "Sound" module.

23.1 Header File Contents Overview

The "Sound" module header **CHAM_AVR_SOUND_DRV_V010.h** has nothing in it except the prototypes for the handful of functions. Rather, it makes more sense to show you the SPI header messages relating to the sound driver:

```
// sound commands
#define SND_CMD_PLAYSOUNDFM 40 // plays a sound on a channel with the sent frequency at 90% volume
#define SND_CMD_STOPSOUND 41 // stops the sound of the sent channel
#define SND_CMD_STOPALLSOUNDS 42 // stops all channels
#define SND_CMD_SETFREQ 43 // sets the frequency of a playing sound channel
#define SND_CMD_SETVOLUME 44 // sets the volume of the playing sound channel
#define SND_CMD_RELEASESOUND 45 // for sounds with infinite duration, releases the sound and it enters
// the "release" portion of ADSR envelope
```

The above messages are located in **CHAM_AVR_TWI_SPI_DRV_V010.H** as well as in the main SPI driver running on the Propeller **CHAM_DEFAULT2_DRV_V112.spin** or **CHAM_DEFAULT1_DRV_V112.spin**. The above messages only expose rudimentary abilities of the sound driver, but they are enough to start and stop sounds, play music, sound fx, etc.

23.2 API Listing Reference

The API listing for the "Sound" module **CHAM_SOUND_DRV_V010.c** is listed in Table 23.1 categorized by functionality.

Table 23.1 – "Sound" module API functions listing.

Function Name	Description
Playing a Sound	
int Sound_Play(int channel, int frequency, int duration);	Plays a sound on one of the 4 channels.
Stopping Sounds	
int Sound_Stop(int channel);	Stops a sound and silences the channel.
int Sound_StopAll(void);	Stops all sound channels and silences them.
Updating a Sound	
int Sound_Update(int channel, int frequency, int volume);	Updates a currently playing sound.

23.3 API Functional Declarations

The following lists each function, a description, comments, and example usage of the function.

Function Prototype:

```
int Sound_Play(int channel, int frequency, int duration);
```

Description: **Sound_Play(...)** instructs the driver to play a single tone on one of the sound channels. The parameters

are **channel** (0..3), **frequency** (0...2000) Hz, **duration** (0..7) seconds where 0 means to play forever. Returns 1.

Example(s): Instruct sound channel 0 to play 440hz for 3 seconds.

```
Sound_Play(0, 440, 3);
```

Function Prototype:

```
int Sound_Stop(int channel);
```

Description: **Sound_Stop(...)** stops the sound playing on channel. Returns 1.

Example(s): Stop all channels manually.

```
for (int ch = 0; ch < 4; ch++)
    Sound_Stop( ch );
```

Function Prototype:

```
int Sound_StopAll(void);
```

Description: **Sound_StopAll()** stops all sound channels and silences them. Returns 1.

Example(s): Stop all playing sounds.

```
Sound_StopAll();
```

Function Prototype:

```
int Sound_Update(int channel, int frequency, int volume);
```

Description: **Sound_Update(...)** allows you to update a sound channel (0..3) "on the fly" without interrupting it. You can modify channel's current frequency and volume. Of course, the parameters must be in range; **frequency** (0...2000) and **volume** (0..255), where 0 means to leave as is. Returns 1.

Example(s): Start channel 0 at 100 Hz and sweep to 1000 Hz, at 10 msec intervals.

```
Sound_Play(0, 100, 0);
for (int f = 100; f < 1000; f++)
{
    Sound_Update( 0, f, 200);
    _delay_ms(10);
} // end for
```

24.0 Keyboard Library Module Primer

The basic premise of the Chameleon design is that it leverages drivers running on the Propeller chip to do all the media and graphics. Thus, whatever features the particular driver running on the Propeller side is the only features we can access via the AVR side. That doesn't mean we can't abstract functionality and add higher level functions that build on the sub-functions, however, this probably isn't productive since you will want to change drivers, re-write the Propeller driver and so forth. Thus, for the majority of the "media/IO" drivers all the API functions do (including the Keyboard) is expose some of the base functionality in nice function calls so you don't have to send SPI messages yourself. Now, the one thing about keyboards (and mice) is they are really SLOW, thus its typically a problem reading them, since your CPU has to slow down to do so; therefore, its really nice for a core on the Propeller to completely handle the keyboard and queue up key presses for you.

For the keyboard, we decided to use a base driver from the original Propeller objects development that was designed to communicate to a standard PS/2 keyboard, nothing fancy. The same driver is used in both Default1 and Default2, so the same idea apply to either driver you load on the Propeller (**CHAM_DEFAULT1_DRV_112.SPIN** or **CHAM_DEFAULT2_DRV_112.SPIN**). The actual driver object used for the keyboard is:

keyboard_010.spin – Basic PS/2 keyboard driver originally written by Parallax.

However, I only exposed a portion of its abilities thru the current SPI messages, no need to waste messages. You can always add more. Therefore, the abilities you see exposed in the following API are just a taste of what it can do. In other words, if you need more functions that the driver supports then you will have to add messages to the SPI driver and add them yourself.

With that in mind, if you want to use the keyboard driver then you need the following files added to your project:

CHAM_AVR_KEYBOARD_DRV_V010.c	- Main C file source for "Keyboard" module.
CHAM_AVR_KEYBOARD_DRV_V010.h	- Header file for "Keyboard" module.

24.1 Header File Contents Overview

The "Keyboard" module header **CHAM_KEYBOARD_DRV_V010.h** has no #defines or globals itself since it literally nothing more than wrapper functions that call thru to the Propeller side driver. But, its worth while to take a look at some of the constants in the actual Spin/ASM code keyboard driver – **KEYBOARD_010.spin**. There are two very large tables at the end of the driver that define the keys in terms of scan codes and ASCII codes. The tables are too long to list, but here's a excerpt from the first table:

		ascii	scan	extkey	regkey	()=keypad
'	Lookup table					
,						
.						
table	word	\$0000	'00			
	word	\$00D8	'01		F9	
	word	\$0000	'02			
	word	\$00D4	'03		F5	
	word	\$00D2	'04		F3	
	word	\$00D0	'05		F1	
	word	\$00D1	'06		F2	
	word	\$00DB	'07		F12	
	word	\$0000	'08			
	word	\$00D9	'09		F10	
	word	\$00D7	'0A		F8	
	word	\$00D5	'0B		F6	
	word	\$00D3	'0C		F4	
	word	\$0009	'0D		Tab	
	word	\$0060	'0E			
	word	\$0000	'0F			
	word	\$0000	'10			
	word	\$F5F4	'11	Alt-R	Alt-L	
	word	\$00F0	'12		Shift-L	
	word	\$0000	'13			
	word	\$F3F2	'14	Ctrl-R	Ctrl-L	
	word	\$0071	'15		q	
	word	\$0031	'16		1	
	word	\$0000	'17			
	word	\$0000	'18			
	word	\$0000	'19			
	word	\$007A	'1A		z	
	word	\$0073	'1B		s	

```

''
''      Key Codes
''
''      00..DF = keypress and keystate
''      E0..FF = keystate only
''
''
''      09      Tab
''      0D      Enter
''      20      Space
''      21      !
''      22      "
''      23      #
''      24      $
''      25      %
''      26      &
''      27      '
''      28      (
''      29      )
''      2A      *
''      2B      +
''      2C      ,
''      2D      -
''      2E      .
''      2F      /
''      30      0..9
''      3A      :
''      3B      ;
''      3C      <
''      3D      =
''      3E      >
''      3F      ?
''      40      @
''      41..5A  A..Z
''      5B      [
''      5C      \
''      5D      ]
''      5E      ^
''      5F      _
''      60      `
''      61..7A  a..z
''      7B      {
''      7C      |
''      7D      }
''      7E      ~
''
''
''

```

24.2 API Listing Reference

198

Table 24.1 – “Keyboard” module API functions listing.

Function Name	Description
Keyboard Driver Control	
int Keyboard_Unload(void);	Unloads the keyboard driver on the Propeller.
int Keyboard_Load(void);	Loads the keyboard driver on the Propeller.
Keyboard Reading Functions	
int Keyboard_GotKey(void);	Tests if a key is queued up.
int Keyboard_Key(void);	Reads the next key from the queue.
int Keyboard_State(int key);	Tests if a particular key is down.

24.3 API Functional Declarations

The following lists each function, a description, comments, and example usage of the function.

Function Prototype:

```
int Keyboard_Unload(void);
```

Description: **Keyboard_Unload()** unloads the keyboard driver on the Propeller and frees up a processing core. Typically, you will make this call before loading the mouse driver or to simply free a core for another process. Returns 1.

Example(s): Unload the keyboard driver to save power.

```
keyboard_Unload();
```

Function Prototype:

```
int Keyboard_Load(void);
```

Description: **Keyboard_Load()** simply loads the keyboard driver into a core on the Propeller. If a keyboard driver is already running, this function will have no effect. Returns 1.

Example(s): Start the keyboard driver.

```
keyboard_Load();
```

Function Prototype:

```
int Keyboard_GotKey(void);
```

Description: **Keyboard_GotKey()** tests if a key is available for reading, returns 1 (key ready) 0 (no key ready).

Example(s): Test if a key has been pressed?

```
if (keyboard_GotKey()==1)
{
  // do something...
} // end if
```

Function Prototype:

```
int Keyboard_Key(void);
```

Description: Keyboard_Key() reads the next keycode out of the keyboard buffer, returns 0 if no key is available.

Example(s): Read the key out of the queue and do something

```
if (keyboard_GotKey())
{
  // read key
  key = Keyboard_Key();
  // do something with key
} // end if key ready
```

Function Prototype:

```
int Keyboard_State(int key);
```

Description: **Keyboard_State(...)** tests if keycode **key** is depressed (1) or released (0) and returns the values.

Example(s): Test if right arrow is down?

```
// test for scancode right arrow
if (Keyboard_State( 0xC1E6 ))
{
  // do something...
} // end if
```

25.0 Mouse Library Module Primer

The basic premise of the Chameleon design is that it leverages drivers running on the Propeller chip to do all the media and graphics. Thus, whatever features the particular driver running on the Propeller side is the only features we can access via the AVR side. That doesn't mean we can't abstract functionality and add higher level functions that build on the sub-functions, however, this probably isn't productive since you will want to change drivers, re-write the Propeller driver and so forth. Thus, for the majority of the "media/IO" drivers all the API functions do (including the Mouse) is expose some of the base functionality in nice function calls so you don't have to send SPI messages yourself. Now, the one thing about mice (and keyboards) is they are really SLOW, thus its typically a problem reading them, since your CPU has to slow down to do so; therefore, its really nice for a core on the Propeller to completely handle the mouse for you.

For the mouse, we decided to use a base driver from the original Propeller objects development that was designed to communicate to a standard PS/2 mouse, nothing fancy. The same driver is used in both Default1 and Default2, so the same idea apply to either driver you load on the Propeller (**CHAM_DEFAULT1_DRV_112.SPIN** or **CHAM_DEFAULT2_DRV_112.SPIN**). The actual driver object used for the mouse is:

mouse_010.spin – Basic PS/2 mouse driver originally written by Parallax.

However, I only exposed a portion of its abilities thru the current SPI messages, no need to waste messages. You can always add more. Therefore, the abilities you see exposed in the following API are just a taste of what it can do. In other words, if you need more functions that the driver supports then you will have to add messages to the SPI driver and add them yourself.

With that in mind, if you want to use the mouse driver then you need the following files added to your project:

CHAM_AVR_MOUSE_DRV_V010.c - Main C file source for “Mouse” module.
CHAM_AVR_MOUSE_DRV_V010.h - Header file for “Mouse” module.

25.1 A Brief Mouse Primer

Mice typically have two axes and a number of buttons. The two axes (X and Y) are decoded as they move either mechanically (roller mice) or optically. Inside the mouse is a microprocessor that actually handles all the decoding and sending of the information. There are two ways that mice send messages; absolute and relative. In absolute mode, the X-Y position is accumulated, and in relative mode, the mouse sends the current change from the last positions. Additionally, mouse have “sensitivity” settings that you can control that slow or speed the relative mouse message values to the surface they are moving on. Finally, some mice have a “Z-axis” which is usually the scrub wheel, this is just like the X-Y.

Finally, the mice have a number of buttons, these are encoded by the mouse as simple switches. There is a whole mouse message API that most mice respond to, but for our purposes we will let the Propeller driver deal with it. All our API needs to do is call the driver over the SPI link, get absolute, relative, position and the buttons – simple.

25.2 Header File Contents Overview

The “Mouse” module header **CHAM_MOUSE_DRV_V010.h** actually has nothing in it once again (I told you this stuff was easy!). However, the mouse does use a data structure located in the global system file **CHAM_AVR_SYSTEM_V010.h**, the data structure is shown below:

```
// generic input device data record type, helps get back data from various multitdata calls
typedef struct gid_event_type
{
    int x,y,z;    // position data
    int buttons;  // buttons, bit encoded
                // bit4 = right-side button
                // bit3 = left-side button
                // bit2 = center/scrollwheel button
                // bit1 = right button
                // bit0 = left button

    } gid_event, *gid_event_ptr;
```

Its nice to think of input devices in all the same way and thus wrap a “container” around them. This data structure was created with that in mind for future expansion. So all mouse messages use this container. The first ints are the x,y position, the 3rd (Z) is usually the scrub wheel. The next int, **buttons** is a bit encoded representation of all the buttons on the mouse up to 5 different standard buttons.

25.3 API Listing Reference

The API listing for the “Mouse” module **CHAM_MOUSE_DRV_V010.c** is listed in Table 25.1 categorized by functionality.

Table 25.1 – “Mouse” module API functions listing.

Function Name	Description
Mouse Driver Control	
int Mouse_Load(void);	Loads the mouse driver.
int Mouse_Unload(void);	Unloads the mouse driver.

Mouse Reading	
int Mouse_Read(gid_event_ptr mouse);	Reads the mouse message.

25.3 API Functional Declarations

The following lists each function, a description, comments, and example usage of the function.

Function Prototype:

```
int Mouse_Load(void);
```

Description: *Mouse_Load()* loads the mouse driver up on a free Propeller core. If the driver is already loaded this function has no effect. Returns 1.

Example(s): Load the mouse driver.

```
Mouse_Load();
```

Function Prototype:

```
int Mouse_Unload(void);
```

Description: *Mouse_Unload()* unloads the mouse driver freeing a core on the Propeller. Typically, you will do this when you want to load the keyboard driver. Returns 1.

Example(s): Load the mouse driver.

```
Mouse_Load();
```

Function Prototype:

```
int Mouse_Read(gid_event_ptr mouse);
```

Description: *Mouse_Read(...)* reads the current mouse event from the driver. In this case, it only reads "relative" events and thus the data is in delta format. You send a pointer to a gid_event structure and the function will write the latest mouse deltas and button states into the structure. Returns 1.

Example(s): Read the mouse position and test if left button is down.

```
gid_event mouse;
Mouse_Read(&mouse);
// test bit 3, left button?
if (mouse.buttons & (0b01000))
{
    // do stuff...
} // end if
```

26.0 Propeller Local I/O Port Module Primer

The Propeller chip on the Chameleon is more or less “peripheraless” if that’s a word? In other words, if you want serial, SPI, A/D, D/A, etc. you use a core and write one yourself using software. Thus, the philosophy of the Propeller chip is a lot of I/O pins and a lot of processors, what you do with them is up to you. The Chameleon uses most of the I/O pins for the video, audio, VGA, and PS/2 port, but we were able to tuck 8 I/O pins away and export them out to what is called the “Propeller Local Port”. With this you can hook up devices that are designed to interface to the Propeller, use it for switches, a D/A, A/D, even another video or VGA port! That said, what I decided to do was write a very rudimentary interface to it, so you can control the direction of each port bit (input or output) and then write/read data to and from the port from the AVR. So, if you run out of digital I/Os on the AVR headers you can use the Propeller port, or if you just want to connect something to the Propeller that takes 8 or less I/O pins that someone has developed a Propeller object for. Either way, in this section, we will see the API for the Propeller Local Port.

One detail before we begin, the Propeller Port does NOT have a driver object running on another core. It’s so simple that we wrote the code right into the master control dispatch program object. There is a complete listing of the code below excerpted from **CHAM_DEFAULT2_DRV_112.spin**.

```
' // Propeller local port specific commands //////////////////////////////////////
' these are handled locally in SPIN on the interface driver's COG

PORT_CMD_SETDIR:      ' sets the 8-bit I/O pin directions for the port 1=output, 0=input
    DIRA[7..0] := g_data

PORT_CMD_READ:        ' reads the 8-bit port pins, outputs are don't cares
    g_spi_result := (INA[7..0] & $FF) ' data is now in g_spi_result,
                                     ' client must perform a general READ_CMD to get it back

PORT_CMD_WRITE:       ' writes the 8-bit port pins, port pins set to input ignore data
    OUTA[7..0] := g_data
```

As you can see, there are 3 commands; set direction, read, and write. These are directly interfaced to SPIN native functions and port I/O control. The AVR side API is nothing more than 3 commands to set the direction, read, and write.

The name of the API file is:

CHAM_AVR_PROP_PORT_DRV_V010.c – Contains the API functions for the Propeller local port functionality.

And once again, this functionality is available in all versions of the main Propeller driver Default1 and 2.

26.1 Header File Contents Overview

The “Propeller Local Port” module header **CHAM_AVR_PROP_PORT_DRV_V010.h** has nothing in it, but the prototypes for the API functions, so nothing to show you there.

26.2 API Listing Reference

The API listing for the “Propeller Local Port” module **CHAM_AVR_PROP_PORT_DRV_V010.c** is listed in Table 26.1 categorized by functionality.

Table 26.1 – “Propeller Local Port” module API functions listing.

Function Name	Description
int PropPort_SetDir(int dirbits);	Sets the port bit direction bits individually.
int PropPort_Read(void);	Read the port input buffer.
int PropPort_Write(int data8);	Writes to the port output buffer.
Note: When reading or writing, I/O pins that aren’t set the proper direction will have invalid data on them, thus if you have set the lower 4-bits as inputs and read the port, only count on the lower 4-bits having valid data. Similarly when writing data, only pins that are outputs will drive current. The only will sink it as inputs. Thus, any data written to them is ignored (as is should be).	

26.3 API Functional Declarations

The following lists each function, a description, comments, and example usage of the function.

Function Prototype:

```
int PropPort_SetDir(int dirbits);
```

Description: *PropPort_SetDir(...)* sets the direction of the 8 I/O bits on the Propeller local port. 1=output, 0=input, eg: 11110000 = out, out, out, out, in, in, in, in. Physically on the port header bit 0 is on the right, bit 7 on the left, so its LSB to MSB right to left. Returns 1.

Example(s): Set the lower 4-bits to outputs, upper 4-bits to inputs.

```
PropPort_SetDir( 0x0F );
```

Function Prototype:

```
int PropPort_Read(void);
```

Description: *PropPort_Read()* reads the pins of the input port that are set as inputs, pins set as outputs are invalid. Returns the pin data.

Example(s): Set the port to all input and read the port.

```
PropPort_SetDir( 0x00 );
inputs = PropPort_Read();
```

Function Prototype:

```
int PropPort_Write(int data8);
```

Description: *PropPort_Write(...)* writes **data8** to the port. I/Os that are set as inputs ignore the bits in those positions. Returns 1.

Example(s): Write the binary patterns for 0..255 on the port at 10 millisecond intervals.

```
PropPort_SetDir( 0xFF );
for (data = 0; data <= 255; data++)
{
    PropPort_Write( data );
    _delay_ms( 10 );
} // end for data
```

27.0 Chameleon Hands-On Programming Tutorials and Demos

Either you spent the past couple days reading up to this section or you cheated and skipped here! No matter how you got here, finally it's time to see some actual code and the Chameleon AVR 8-bit in action! First, some ground rules.

- Every tutorial was designed to show off some aspect of the Chameleon AVR's hardware or a particular library.
- Tutorials are not designed to be mind blowing or amazing, they are designed to show you how to use each system and are very simple to get you started.
- The Chameleon's AVR's API is far from bug free. There are probably lots of little bugs that I haven't found, so beware. However, the APIs don't really do much, but send messages to the objects running on the Propeller chip, so ultimately if something doesn't work it's probably the driver!
- Each tutorial demo follows the same outline; the demo will be introduced, a screen shot (if applicable), compilation instructions for both the Arduino mode of operation and straight AVRStudio.
- Finally, the tutorials are organized in order of various sub-systems like keyboard, sound, graphics, etc. And each demo might have a couple versions with different video drivers and/or VGA/NTSC support. I will tend to review only one version of the demo since the others are usually variants. However, compilation files for each demo version will be included.

After experimenting with all the tutorial demos you will have a command of all the sub-systems of the Chameleon AVR 8-bit as well as be able to use the provided APIs to get things done. Then I suggest you start with the demos as templates to get your own programs up and running before trying to do things from scratch.

27.1 Setup to Compile the Demos and Tutorials

To compile and run any of the tutorial demos you will need to have one of two things setup:

1. An AVRStudio project open and ready to go along with an AVRISP MKII programmer to directly download to the Chameleon's AVR programming port.
2. An Arduino "Sketch" project along with the Arduino bootloader previously FLASHED into the AVR 328P processor (this is how the Chameleon AVR ships).

In a nutshell, if you just got your Chameleon AVR, it comes pre-FLASHED with a the Arduino bootloader and you can use method 2 to run all the experiments. If on the other hand, you prefer a more robust tool like the Atmel AVRStudio IDE and you have purchased the AVR ISPMKII programmer (or similar) then you can compile and program the Chameleon AVR via the 6-pin ISP port. Both methods and setups were covered earlier in the manual. But, let's briefly review a couple details about the differences between the AVRStudio and Arduino version of each demo.

27.1.1 Differences Between the AVRStudio and Arduino Demos and General Porting Strategies

Before we get into the pre-setup of AVRStudio and the Arduino tool, let's take a step back and discuss the differences between the two tools and software as you write it.

First off, with AVRStudio and the Arduino tool, you program in C/C++. Technically, AVRStudio you can program in straight C while the Arduino tool you can use C/C++. That said, we made all programs C, so they work on both tool chains. So assuming we start with a simple C program on AVRStudio then porting it to Arduino requires only a few changes relating a number of topic areas that we will cover below:

Include Files

The Arduino tool isn't as flexible as AVRStudio. The whole point of the Arduino tool is to insulate the user from the complexities of compilers, linkers, make files, etc. Thus, you have to code in the "**template**" that Arduino provides, to this end, the search paths are a little different. When programming in AVRStudio, your header files will typically be included like this:

AVRStudio Include Example

```
#include "myfile.h"
```

The quotes indicate to the compiler to look in the working directory of the project for the header. However, with the Arduino tool, there really aren't "header" files for your projects anymore that aren't part of an official Arduino "**library**" file. And library files are located in the **hardware \ libraries** \ directory of your Arduino install directory. In other words, the location of your library C/C++ code and the header are *in the search path* of the compiler not the project; therefore, like any system header `stdio.h`, etc. you use angle brackets to include your header files:

Arduino Include Example

```
#include <myfile.h>
```

But, usually the only reason you include your own header file is that you have created an Arduino library called, "myfile" and therefore there is a **MYFILE** \ directory inside the **hardware \ libraries** \ and there is a `myheader.c/cpp` file along with a `myfile.h`.

Now, sometimes, you just want to add a header file to your sketch, but you don't want to create a library with C/C++ code, nor do you need it. You just want a header maybe with some macros, defines, data, whatever. There is a trick to doing this.

Step 1: Add the header file to your sketch headers with the following syntax (notice the `../`) and that quotes are used to enclose the header file this time:

```
#include "../headerfile.h"
```

Step 2: Copy the header file into your sketch directory.

You can learn more about include files, libraries and the Arduino environment here:

<http://arduino.cc/en/Reference/Libraries>

Renaming the `main()` Function and `setup()`

The Arduino "environment" builds a final C/C++ program with your source "sketch" along with a number of other files. In fact, your program does need a ***main()***. The Arduino build provides a ***main()*** and inside of it calls a couple functions; one as a `setup`, the other as the main loop which more or less replaces ***main()***.

This is best illustrated with an example, below are two complete programs for "Hello World". One will compile with AVRStudio, the other will compile with the Arduino tool.

AVRStudio Version

```

////////////////////////////////////
// INCLUDES //////////////////////////////////////
////////////////////////////////////

#define __AVR_ATmega328P__

// include everything
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <avr/eeprom.h>
#include <avr/io.h>
#include <avr/sleep.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>
#include <inttypes.h>
#include <compat/twi.h>
#include <avr/wdt.h>

// include headers required for demo

```

```

#include "CHAM_AVR_SYSTEM_V010.h" // you need this one always
#include "CHAM_AVR_TWI_SPI_DRV_V010.h" // you need this always
#include "CHAM_AVR_NTSC_DRV_V010.h" // needed for NTSC driver
#include "CHAM_AVR_VGA_DRV_V010.h" // needed for VGA driver

////////////////////////////////////
// DEFINES AND MACROS //////////////////////////////////////
////////////////////////////////////

// define CPU frequency in MHZ here if not defined in Makefile or other includes,
// compiler will throw a warning, ignore
#ifndef F_CPU
#define F_CPU 16000000UL // 28636360UL, 14318180UL, 21477270 UL
#endif

////////////////////////////////////
// MAIN //////////////////////////////////////
////////////////////////////////////

int main(void)
{
// initialize and set SPI rate
SPI_Init( SPI_DEFAULT_RATE );

// clear screens
NTSC_ClearScreen();
VGA_ClearScreen();

// enter infinite loop...
while(1)
{
// print on NTSC terminal screen
NTSC_Color(0);
NTSC_Term_Print("Hello world! ");

// print on VGA terminal screen
VGA_Color(0);
VGA_Term_Print("Hello world! ");

// slow things down a bit, so we can read the text!
_delay_ms(10);

} // end while
} // end main

```

Take note of the highlight header files and the **main()** function, now take a look at the Arduino version, its nearly the same with a slight syntactic change to the header files and **main()** has been replaced with **setup()** and **loop()**:

Arduino Sketch Version

```

////////////////////////////////////
// INCLUDES //////////////////////////////////////
////////////////////////////////////

#define __AVR_ATmega328P__

// include everything
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <avr/eeprom.h>
#include <avr/io.h>
#include <avr/sleep.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>
#include <inttypes.h>
#include <compat/twi.h>
#include <avr/wdt.h>

// include headers required for demo, notice the use of <> brackets for the arduino version
// since it searches for headers
// differently than AVRStudio
#include <CHAM_AVR_SYSTEM_V010.h> // you need this one always
#include <CHAM_AVR_TWI_SPI_DRV_V010.h> // you need this always
#include <CHAM_AVR_NTSC_DRV_V010.h> // needed for NTSC driver
#include <CHAM_AVR_VGA_DRV_V010.h> // needed for VGA driver

////////////////////////////////////
// DEFINES AND MACROS //////////////////////////////////////
////////////////////////////////////

// define CPU frequency in MHZ here if not defined in Makefile or other includes,

```

```
// compiler will throw a warning, ignore
#ifndef F_CPU
#define F_CPU 16000000UL // 28636360UL, 14318180UL, 21477270 UL
#endif

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// MAIN //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// SETUP CALL (ARDUINO SPECIFIC: initialization call)
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void setup() // run once, when the sketch starts
{

} // end setup

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// LOOP CALL (ARDUINO SPECIFIC: main entry point)
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void loop() // run over and over again
{
// initialize and set SPI rate
SPI_Init( SPI_DEFAULT_RATE );

// clear screens
NTSC_ClearScreen();
VGA_ClearScreen();

// enter infinite loop...
while(1)
{
// print on NTSC terminal screen
NTSC_Color(0);
NTSC_Term_Print("Hello world!  ");

// print on VGA terminal screen
VGA_Color(0);
VGA_Term_Print("Hello world!  ");

// slow things down a bit, so we can read the text!
_delay_ms(10);
} // end while
} // end main
```

Reviewing both files, 90% of porting all programs from AVRStudio to Arduino is simply making sure header files have angle brackets <> and that **main()** is replaced with **loop()** and that a function **setup()** is added to your program. The **setup()** function can be empty, but will be called before **loop()** is, thus it's a nice place to put initialization code. Personally, I just think of **loop()** as **main()** and leave it at that.

Serial I/O Libraries

The last little porting tip has to do with serial communications. We wrote serial UART libraries for the Chameleon AVR that are stored in **CHAM_AVR_UART_DRV_V010.C|H**. You have seen these in the API overview section. However, Arduino has a built in serial class that you can review here:

<http://arduino.cc/en/Reference/Serial>

Thus, you can use our library or the Arduino library, if you decide to use one or the other then porting is a simple matter of changing a couple function calls, below in Table 27.1 is a quick list of the function call mappings for the serial calls.

Table 27.1 – Chameleon AVR vs. Arduino Serial UART Library Mapping.

CHAM_AVR_UART_DRV_V010.C	Arduino Serial Library Version
Function Name	Function Name
int UART_Init(long baudrate)	Serial.begin(baudrate);

void UART_Print_String(char *string)	Serial.print(data);
	Serial.println(data);
int UART_putc(unsigned char ch)	Serial.print(data);
int UART_getc(void);	Serial.read()

You will notice that on the right side (Arduino functions) that there are no data types as the parameters. This is because this is an overloaded C++ class with overloaded data types. Thus, you can send characters, strings, numbers, etc. to the **print** and **println** functions. Therefore, to port in either direction, you just replace the functions with their counterparts and maybe a cast or two and you're done!

27.1.2 Setup for AVRStudio Version of Demos

The AVRStudio of the demos are relatively straightforward to setup. We have already covered how to setup AVRStudio, to compile, build a FLASH image and download with the AVRISP MKII programmer. Thus, for each of the demos, all you need to do is add the source C files to the project along with the main program and build the program and download.

All the source API libraries for the demos and the demos themselves are located on the DVD ROM in the usual directory:

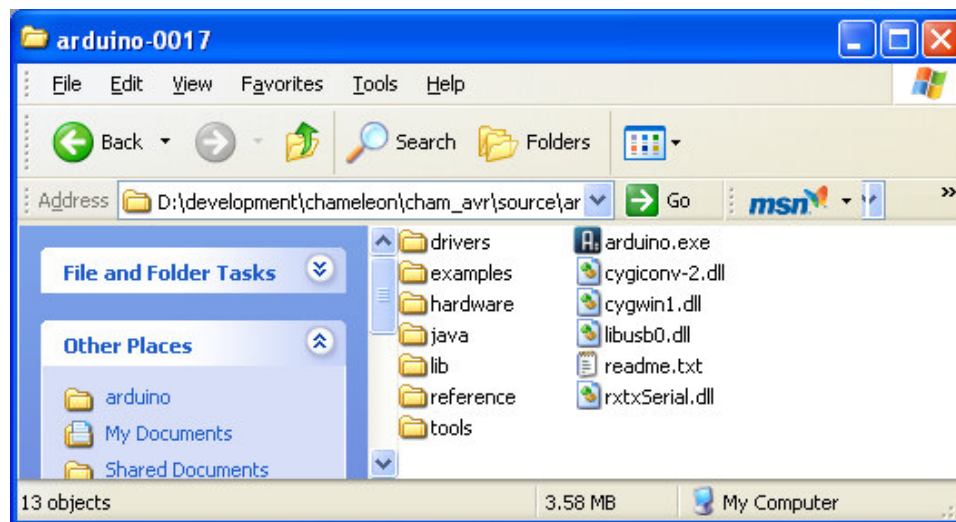
DVD-ROM:\ CHAM_AVR \ SOURCE \ *.*

During initial setup of the tool you should have copied this entire directory to your hard drive.

27.1.3 Setup for Arduino Version of Demos

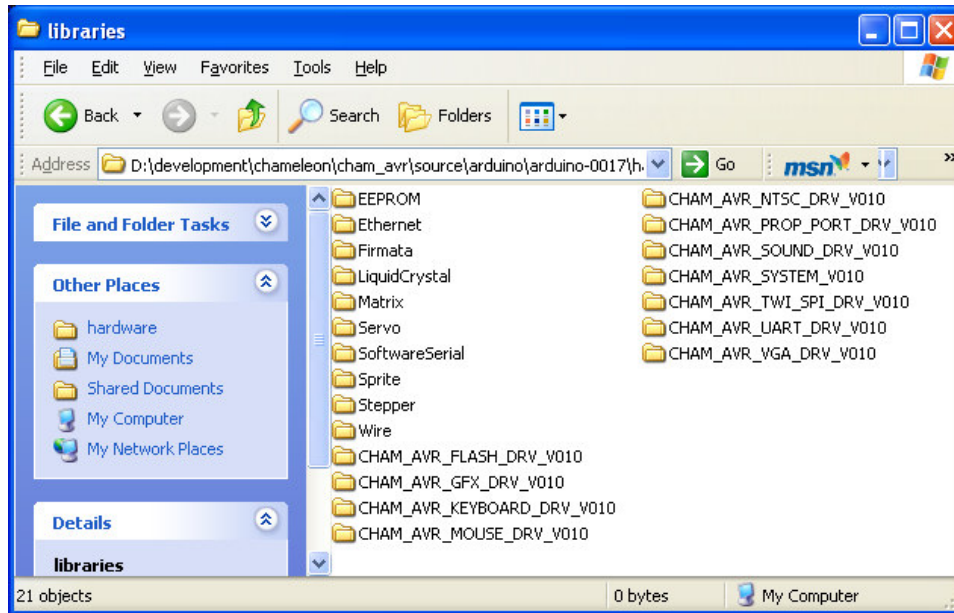
The setup for the Arduino version of the demos is a little different than the AVRStudio version since Arduino deals with "libraries" and "sketches" rather than straight C/C++ files. Thus, there is a little setup. The first thing is that you should have already copied all the Libraries from the DVD ROM into your Arduino installation directory. If you recall, after you install the Arduino tool it creates a number of directories from the root installation directory. They look something like Figure 27.1.

Figure 27.1 – The root Arduino installation directory.



Then if you drill down into the \HARDWARE sub-directory, you will find the \LIBRARIES sub-directory, the contents of that directory are shown in Figure 27.2

Figure 27.2 – The Libraries directory which contains both the default Arduino libraries as well as our custom libraries.



If you haven't copied the new libraries into this directory then you should do so now. They are located on the DVD ROM in the following location:

DVD-ROM:\ CHAM_AVR \ TOOLS \ ARDUINO \ LIBRARIES

TIP

If the Arduino tool is running make sure to quit the application and restart, so the tool can scan the Libraries directory and add them to the IDE's **<Sketch → Import Library>** menu option.

Assuming, you have the libraries directory setup with all the new Chameleon specific libraries for NTSC, VGA, sound, FLASH, serial, keyboard, mouse, etc. The next thing we need to do is actually copy the pre-made "Sketches" into your local sketch directory, so you don't have to build each sketch yourself.

Copying the Sketches from DVD

To save you time for each demo construction, I have pre-built a sketch for each demo. In essence, I took the program as I developed it in AVRStudio, then create a sketch with the same root name, then saved it. Of course, I performed any porting operations like changing header include syntax, changing **main()** to **loop()** etc. But, more or less just a couple minutes each to port each program in most cases. All the ported sketches are located on the DVD here:

DVD-ROM:\ CHAM_AVR \ TOOLS \ ARDUINO \ SKETCHES \ *.*

Simply copy the contents of this directory (which contains a number of sub-directories, each representing a sketch) into your local hard drive where you installed the Arduino tool and its **Sketches** directory. After copying, re-start the Arduino tool to make sure they are scanned.

Now, you are ready to go for the Arduino versions as well.

27.1.3 Setting the Chameleon Hardware Up

Finally, let's take a brief moment to remind ourselves what's going on with the hardware. If you are using AVRStudio, then you are going to build a project for each demo, or change a single project, re-build and FLASH the file to the Chameleon AVR with the AVR ISP MKII programmer. If you don't own a programmer, then you will use the Chameleon in Arduino mode, and thus, you need the bootloder on the Chameleon (which we deliver the unit with). In this case, make sure you

have the serial select switch on the Chameleon in the **down** position, so the PC can get to the serial port of the AVR since this is how the bootloader works via serial.

28.1 Graphics Demos

The following graphics demos illustrate examples in both NTSC and VGA modes as well as use of the advanced tile modes of the NTSC driver.

Each example will typically consist of the primary source file for the demo as well as:

- The System API library module **CHAM_AVR_SYSTEM_V010.c|h**.
- The main SPI API library module **CHAM_AVR_TWI_SPI_DRV_V010.c|h**.
- The NTSC, VGA, GFX driver or all of them.

And any other ancillary drivers for sound, keyboard, etc.

All the required files can of course be found in the **\Source** directory on the DVD (which you should have already copied onto your hard drive) located here:

**DVD-ROM:\ CHAM_AVR \ SOURCE **

AVRStudio TIP

In general, you will include all the sources in your project (.C files) and make sure all the .H header files are in the same working directory, so the compiler can find them. You do NOT include the .H header files in the compilation list of source files, the compiler will do this for you. Only include .C and .S files in your source file link list in your project file list to the left of the tool.

The Arduino version will be in **"Sketch"** form already, so all you need to do is load the Sketch in and upload and you're in business!

28.1.1 NTSC Printing Demo

This demo simply uses the NTSC terminal API to print "Hello World" to the NTSC screen. Figure 28.1 shows the demo in action.

Figure 28.1 – NTSC terminal printing demo.



Compiling and Running the AVRStudio Version

Demo Version Description: NTSC version prints "Hello World" to screen.

Main Source File: CHAM_AVR_NTSC_PRINT_DEMO_01.C

Additional Files Required:

- CHAM_AVR_SYSTEM_V010.c|h
- CHAM_AVR_TWI_SPI_DRV_V010.c|h
- CHAM_AVR_NTSC_DRV_V010.c|h

General Requirements: Video port connected to NTSC TV monitor.

Controls Required: None.

Compiling and Running the Arduino Version

Demo Version Description: NTSC version prints "Hello World" to screen.

Main Sketch Source File: CHAM_AVR_NTSC_PRINT_DEMO

Additional Library Files Required (imported automatically by sketch)

- CHAM_AVR_SYSTEM_V010
- CHAM_AVR_TWI_SPI_DRV_V010
- CHAM_AVR_NTSC_DRV_V010

General Requirements: Video port connected to NTSC TV monitor.

Controls Required: None.

Technical Overview

The demo simply uses the NTSC API to print "Hello World" on the screen. When the printing reaches the bottom of the screen the terminal scrolls the screen image up one line. The program is so short, let's take a quick peek at only the contents of the *main()* or *loop()* in the Arduino version:

```

// MAIN
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
int main(void)
{
  // initialize and set SPI rate
  SPI_Init( SPI_DEFAULT_RATE );

  // give Prop a moment before sending it commands, the boot process is about a second
  // if you like you can speed it up, by removing the Prop driver's "LED blink" sequence in the driver
  // or speed it up, but commands that are sent before the Prop driver is done booting will be ignored.
  // therefore, if you have a lot of set up work to do, then you don't require a delay, but if you jump right
  // into commands, then you need a good 1.5 - 2 second delay
  _delay_ms( 2500 );

  // clear screens
  NTSC_ClearScreen();

  // enter infinite loop...
  while(1)
  {
    // print on NTSC terminal screen
    NTSC_Color(0);
    NTSC_Term_Print("Hello world!  ");

    // slow things down a bit, so we can read the text!
    _delay_ms(10);

  } // end while
} // end main

```

First, it should be amazing to you that in a few lines of code you are generating a NTSC color text display! Reviewing the code, the program starts, waits for the Propeller to boot (don't forget to do this), then the call to clear the screen, and we enter the main *while* loop. Here a redundant call to set the color is made and the text is printed to the NTSC terminal with *NTSC_Term_Print(...)* – that's it!

Summary

This demo shows how easy it is to get NTSC terminal like text output going on the Chameleon.

28.1.2 NTSC Glowing Top/Bottom Overscan Demo

This demo simply uses the NTSC tile graphics engine extensions to glow the top and bottom overscan colors on the display. The default2 driver uses a high performance tile engine that we developed that supports scrolling, 4 colors per tile, large playfields and a lot of more. One of its simple features is the ability to control the top and bottom over scan colors on the screen. This demo simply cycles thru those colors and shows you how to make calls to update the overscan color registers in the tile engine running on the Propeller chip. Figure 28.2 shows the demo in action.

Figure 28.2 – NTSC overscan color animation program demo.



Compiling and Running the AVRStudio Version

Demo Version Description: NTSC version prints animates overscan colors.

Main Source File: CHAM_AVR_NTSC_GLOW_DEMO_01.C

Additional Files Required:

- CHAM_AVR_SYSTEM_V010.c|h
- CHAM_AVR_TWI_SPI_DRV_V010.c|h
- CHAM_AVR_NTSC_DRV_V010.c|h
- CHAM_AVR_GFX_DRV_V010.c|h

General Requirements: Video port connected to NTSC TV monitor.

Controls Required: None.

Compiling and Running the Arduino Version

Demo Version Description: NTSC version animates overscan colors.

Main Sketch Source File: CHAM_AVR_NTSC_GLOW_DEMO_01

Additional Library Files Required (imported automatically by sketch)

- CHAM_AVR_SYSTEM_V010
- CHAM_AVR_TWI_SPI_DRV_V010

- CHAM_AVR_NTSC_DRV_V010
- CHAM_AVR_GFX_DRV_V010

General Requirements: Video port connected to NTSC TV monitor.

Controls Required: None.

Technical Overview

The demo simply uses the NTSC GFX to animate the top and bottom overscan colors (this is only supported with the default2 version of the driver and the NTSC system). Below is an excerpt of the code from the main **while** loop that performs the animation.

```
// enter infinite loop...glowing the top and bottom overscan, tile engine functions required for this
while(1)
{
    // glow thru every color, every brightness
    for (col = 0; col <= 15 ; col++)
    {
        for (bright = 2; bright < 6; bright++)
        {
            // build up Propeller compatible color [chroma:4 | chroma bit:1 | luma:3]
            color = (col << 4) | (0b00001000) | (bright);

            // set top overscan
            GFX_Set_Top_Overscan_Color( color & 0xFF );

            // set bottom overscan
            GFX_Set_Bottom_Overscan_Color( color & 0xFF );

            // slow things down a bit, so we can read the text!
            _delay_ms(100);

        } // end for bright
    } // end for color
} // end while
```

The code simply formats a Propeller compliant color and then calls the API function to change the top and bottom overscan color, that's it.

Summary

This demo shows how easy it is to use the advanced GFX tile engine API functions, in particular to animate the overscan color on the NTSC display.

28.1.3 NTSC Smooth Scrolling Tilemap Demo

This demo illustrates the smooth scrolling feature of the **default2** versions of the Propeller driver. The NTSC tile engine supports **course** horizontal and vertical scrolling simply by changing the tile map base pointer. However, the engine also supports pixel smooth scrolling in the **vertical** direction. The allows you to smoothly scroll the tile map any number of rows you wish. First, you smooth scroll 0...7 pixels then you reset the smooth scroll register and course scroll and repeat. To the user this will look like one continuous scrolling action. Figure 28.3 shows the demo in action.

Figure 28.3 – NTSC smooth scrolling tilemap demo.



Compiling and Running the AVRStudio Version

Demo Version Description: NTSC version smooth scrolls array of space invaders.

Main Source File: CHAM_AVR_NTSC_SMOOTH_SCROLL_DEMO_01.C

Additional Files Required:

- CHAM_AVR_SYSTEM_V010.c|h
- CHAM_AVR_TWI_SPI_DRV_V010.c|h
- CHAM_AVR_NTSC_DRV_V010.c|h
- CHAM_AVR_GFX_DRV_V010.c|h

General Requirements: Video port connected to NTSC TV monitor.

Controls Required: None.

Compiling and Running the Arduino Version

Demo Version Description: NTSC version smooth scrolls array of space invaders.

Main Sketch Source File: CHAM_AVR_NTSC_SMOOTH_SCROLL_DEMO

Additional Library Files Required (imported automatically by sketch)

- CHAM_AVR_SYSTEM_V010
- CHAM_AVR_TWI_SPI_DRV_V010
- CHAM_AVR_NTSC_DRV_V010
- CHAM_AVR_GFX_DRV_V010

General Requirements: Video port connected to NTSC TV monitor.

Controls Required: None.

Technical Overview

This demo uses the vertical scroll register to scroll an array of tiles (space invader characters) up and down. If we wanted to we could scroll the image continuously by course scrolling after each smooth scroll thru the scanlines 0..7 of each character. Below is an excerpt that shows the space invader rendering code and the smooth scrolling main loop:

```
NTSC_ClearScreen();

// basic white
NTSC_Color(0);

// draw a matrix of space invaders with extended characters
for (y = 0; y < 6; y++)
{
    for (x = 0; x < NUM_SPACE_INVADERS_PER_ROW; x++)
    {
        // position next invader
        NTSC_SetXY(x*2 + SPACE_INVADERS_ORIGIN_X , y*2 + SPACE_INVADERS_ORIGIN_Y);

        // draw next invader
        NTSC_Term_Char( SPACE_INVADERS_CHAR_OFFSET + (y % 3) );
    } // end for x
} // end for y

// enter infinite loop...
while(1)
{
    // set smooth scroll vertical position register 0..7
    GFX_Set_VScroll( scroll_y );

    // change scroll position
    scroll_y += scroll_dy;

    // test for limit
    if ( (scroll_y > 7) || (scroll_y < 0) )
    {
        scroll_dy = -scroll_dy;
        scroll_y += scroll_dy;
    } // end if

    // slow things down a bit, so we can read the text!
    _delay_ms(50);
} // end while
```

I have highlighted the single API call that sets the smooth scrolling register, that's all there is to it!

Summary

This demo shows how easy it is to smooth scroll vertically. Additionally, it shows some of the hidden / special characters in our little character set to help make games!

28.1.4 VGA Printing Demo

This demo simply uses the VGA terminal API to print "Hello World" to the VGA screen. Figure 28.4 shows the demo in action.

Figure 28.4 – VGA terminal printing demo.



Compiling and Running the AVRStudio Version

Demo Version Description: VGA version prints "Hello World" to screen.

Main Source File: CHAM_AVR_VGA_PRINT_DEMO_01.C

Additional Files Required:

- CHAM_AVR_SYSTEM_V010.c|h
- CHAM_AVR_TWI_SPI_DRV_V010.c|h
- CHAM_AVR_VGA_DRV_V010.c|h

General Requirements: VGA port connected to VGA monitor.

Controls Required: None.

Compiling and Running the Arduino Version

Demo Version Description: VGA version prints "Hello World" to screen.

Main Sketch Source File: CHAM_AVR_VGA_PRINT_DEMO

Additional Library Files Required (imported automatically by sketch)

- CHAM_AVR_SYSTEM_V010
- CHAM_AVR_TWI_SPI_DRV_V010
- CHAM_AVR_VGA_DRV_V010

General Requirements: VGA port connected to VGA monitor.

Controls Required: None.

Technical Overview

The demo simply uses the VGA API to print "Hello World" on the screen. When the printing reaches the bottom of the screen the terminal scrolls the screen image up one line. The program is so short, let's take a quick peek at only the contents of the *main()* or *loop()* in the Arduino version:

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// MAIN //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

int main(void)
{
// initialize and set SPI rate
SPI_Init( SPI_DEFAULT_RATE );

// give Prop a moment before sending it commands, the boot process is about a second
// if you like you can speed it up, by removing the Prop driver's "LED blink" sequence in the driver
// or speed it up, but commands that are sent before the Prop driver is done booting will be ignored.
// therefore, if you have a lot of set up work to do, then you don't require a delay, but if you jump right
// into commands, then you need a good 1.5 - 2 second delay
_delay_ms( 2500 );

// clear screens
VGA_ClearScreen();

// enter infinite loop...
while(1)
{
// print on VGA terminal screen
VGA_Color(0);
VGA_Term_Print("Hello World!  ");

// slow things down a bit, so we can read the text!
_delay_ms(10);

} // end while
} // end main

```

First, it should be amazing to you that in a few lines of code you are generating a VGA color text display! Reviewing the code, the program starts, waits for the Propeller to boot (don't forget to do this), then the call to clear the screen, and we enter the main **while** loop. Here a redundant call to set the color is made and the text is printed to the VGA terminal with **VGA_Term_Print(...)** – that's it!

Summary

This demo shows how easy it is to get VGA terminal like text output going on the Chameleon.

28.1.5 Dual NTSC / VGA Printing Demo

One of the cool things about the NTSC and VGA terminal tile drivers is they have the exact same base features. Which means that porting your code to work on NTSC or VGA is usually a matter of changing API prefix function names with from "NTSC" to "VGA" and vice versa. Of course, the advanced "gaming" tile engine support is only for the NTSC driver; however, no one is stopping you from using or making a more advanced VGA drivers on the Propeller. I simply choose not to this time around. In any case, this demo is just a "Hello World" demo that prints both on the NTSC and VGA monitors at the same time to show you how easy this is. Figure 28.5 shows the demo in action.

Figure 28.5 – Dual NTSC / VGA terminal printing demo.



Compiling and Running the AVRStudio Version

Demo Version Description: Prints "Hello World" to both NTSC and VGA displays.

Main Source File: CHAM_AVR_NTSC_VGA_PRINT_DEMO_01.C

Additional Files Required:

- CHAM_AVR_SYSTEM_V010.c|h
- CHAM_AVR_TWI_SPI_DRV_V010.c|h
- CHAM_AVR_VGA_DRV_V010.c|h
- CHAM_AVR_NTSC_DRV_V010.c|h

General Requirements: NTSC port connect to NTSC TV monitor and VGA port connected to VGA monitor.

Controls Required: None.

Compiling and Running the Arduino Version

Demo Version Description: Prints "Hello World" to both NTSC and VGA displays.

Main Sketch Source File: CHAM_AVR_NTSC_VGA_PRINT_DEMO

Additional Library Files Required (imported automatically by sketch)

- CHAM_AVR_SYSTEM_V010
- CHAM_AVR_TWI_SPI_DRV_V010
- CHAM_AVR_VGA_DRV_V010
- CHAM_AVR_NTSC_DRV_V010

General Requirements: NTSC port connect to NTSC TV monitor and VGA port connected to VGA monitor.

Controls Required: None.

Technical Overview

The demo simply uses both the NTSC and VGA terminal API functions to print "Hello World" on both screens and scroll. Here's just the **main()** / **loop()** excerpt since this demo is nothing more than a copy of the NTSC and VGA versions respectively and merged.

```
int main(void)    // run over and over again
{
// initialize and set SPI rate
SPI_Init( SPI_DEFAULT_RATE );

// clear screens
NTSC_ClearScreen();
VGA_ClearScreen();

// enter infinite loop...
while(1)
{
    // print on NTSC terminal screen
    NTSC_Color(0);
    NTSC_Term_Print("Hello world!  ");

    // print on VGA terminal screen
    VGA_Color(0);
    VGA_Term_Print("Hello world!  ");

    // slow things down a bit, so we can read the text!
    _delay_ms(10);

} // end while
} // end main
```

The demo simply uses the same basic commands, but send them to both the NTSC and VGA monitors.

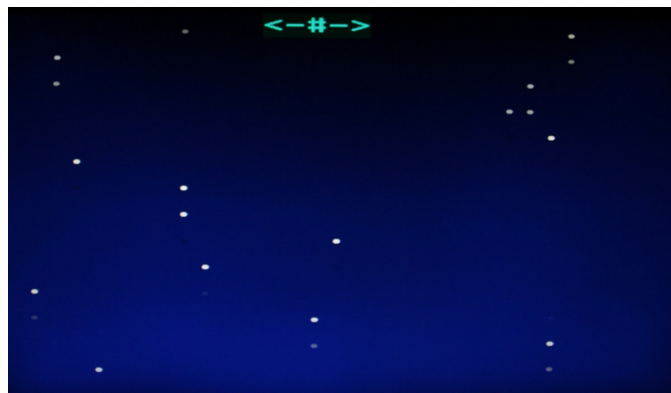
Summary

This demo shows how easy it is to get both NTSC and VGA terminal like text output going on the Chameleon.

28.1.6 VGA Star Field Demo

The default2 VGA driver isn't as impressive as the NTSC driver. In fact, its pretty boring. Its good for displaying text and acting as a terminal output, but that's about it. However, with a little ingenuity we can make it do some cool things! This little demo, draw stars on the VGA screen on the last display line, this causes the screen to scroll and presto – animated star field! Then on top of the star field a little character based ship is drawn. Figure 28.6 shows the demo in action.

Figure 28.6 – VGA terminal text graphics demo – Star Field.



Compiling and Running the AVRStudio Version

Demo Version Description: Vertically scrolls a star field and draws a little ASCII character space ship.

Main Source File: CHAM_AVR_VGA_STARS_DEMO_01.C

Additional Files Required:

- CHAM_AVR_SYSTEM_V010.c|h
- CHAM_AVR_TWI_SPI_DRV_V010.c|h
- CHAM_AVR_VGA_DRV_V010.c|h

General Requirements: VGA port connected to VGA monitor.

Controls Required: None.

Compiling and Running the Arduino Version

Demo Version Description: Vertically scrolls a star field and draws a little ASCII character space ship.

Main Sketch Source File: CHAM_AVR_VGA_STARS_DEMO

Additional Library Files Required (imported automatically by sketch)

- CHAM_AVR_SYSTEM_V010
- CHAM_AVR_TWI_SPI_DRV_V010
- CHAM_AVR_VGA_DRV_V010

General Requirements: VGA port connected to VGA monitor.

Controls Required: None.

Technical Overview

Here's just the *main()* / *loop()* excerpt which shows the entire star field generator:

```
int main(void) // run over and over again
{
    int ship_x = 12, ship_y = 0;

    // initialize and set SPI rate
    SPI_Init( SPI_DEFAULT_RATE );

    // give Prop a moment before sending it commands, the boot process is about a second
    // if you like you can speed it up, by removing the Prop driver's "LED blink" sequence in the driver
    // or speed it up, but commands that are sent before the Prop driver is done booting will be ignored.
    // therefore, if you have a lot of set up work to do, then you don't require a delay, but if you jump right
    // into commands, then you need a good 1.5 - 2 second delay
    _delay_ms( 2500 );

    // clear screens
    VGA_ClearScreen();

    // enter infinite loop...
    while(1)
    {
        // draw next star

        // set color
        VGA_Color(0);

        // position star at bottom to cause scroll
        VGA_SetXY( rand() % 32, 14 );
        VGA_Term_Print("                      ");

        // draw our little ship at top

        // set color to green (with a little forcefield :)
        VGA_Color(4);

        // position at top of screen and draw with ASCII art
        VGA_SetXY( ship_x, ship_y );
    }
}
```

```
VGA_Term_Print("<#->");
// add some logic here to move the ship!
// slow things down a bit
_delay_ms(10);
} // end while
} // end main
```

This demo should bring back memories of BASIC programming on your C64 / Atari 800 / Apple][, if you ever did any text only games and had to resort to things like "<#->" as your space ship!

Summary

This demo shows that with a little creativity even the vanilla VGA terminal driver can be used for games.

29.1 Sound Demos

The following sound demo(s) illustrate how to make calls to the sound driver on the Chameleon (running on the Propeller of course). The output of the demo displays on both the NTSC and VGA monitors.

The examples typically consist of the primary source file for the demo as well as:

- The System API library module **CHAM_AVR_SYSTEM_V010.c|h**.
- The main SPI API library module **CHAM_AVR_TWI_SPI_DRV_V010.c|h**.
- The Sound driver API library module **CHAM_AVR_SOUND_DRV_V010.c|h**.
- The NTSC, VGA, GFX driver or all of them.

And any other ancillary drivers keyboard, mouse, etc.

All the required files can of course be found in the \Source directory on the DVD (which you should have already copied onto your hard drive) located here:

**DVD-ROM:\ CHAM_AVR \ SOURCE **

AVRStudio TIP

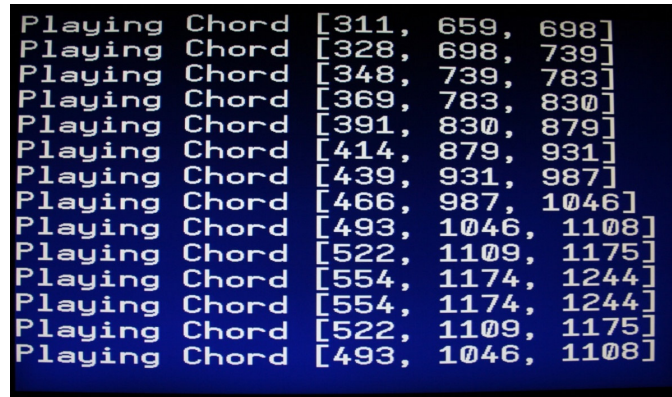
In general, you will include all the sources in your project (.C files) and make sure all the .H header files are in the same working directory, so the compiler can find them. You do NOT include the .H header files in the compilation list of source files, the compiler will do this for you. Only include .C and .S files in your source file link list in your project file list to the left of the tool.

The Arduino version will be in "Sketch" form already, so all you need to do is load the Sketch in and upload and you're in business!

29.1.1 Sound Demo

This demo uses the Sound API and driver to play some chords up and down. The chord frequencies are displayed on both the NTSC and VGA displays. Figure 29.1 shows the demo in action.

Figure 29.1 – Sound demo VGA output.



```

Playing Chord [311, 659, 698]
Playing Chord [328, 698, 739]
Playing Chord [348, 739, 783]
Playing Chord [369, 783, 830]
Playing Chord [391, 830, 879]
Playing Chord [414, 879, 931]
Playing Chord [439, 931, 987]
Playing Chord [466, 987, 1046]
Playing Chord [493, 1046, 1108]
Playing Chord [522, 1109, 1175]
Playing Chord [554, 1174, 1244]
Playing Chord [554, 1174, 1244]
Playing Chord [522, 1109, 1175]
Playing Chord [493, 1046, 1108]
  
```

Compiling and Running the AVRStudio Version

Demo Version Description: Plays sounds on the speaker.

Main Source File: CHAM_AVR_SOUND_DEMO_01.C

Additional Files Required:

- CHAM_AVR_SYSTEM_V010.c|h
- CHAM_AVR_TWI_SPI_DRV_V010.c|h
- CHAM_AVR_NTSC_DRV_V010.c|h
- CHAM_AVR_VGA_DRV_V010.c|h
- CHAM_AVR_SOUND_DRV_V010.c|h

General Requirements: Video port connected to NTSC or VGA monitors, RCA audio connected to an amplifier or the TV's audio input.

Controls Required: None.

Compiling and Running the Arduino Version

Demo Version Description: Plays sounds on the speaker.

Main Sketch Source File: CHAM_AVR_SOUND_DEMO_01.C

Additional Library Files Required (imported automatically by sketch)

- CHAM_AVR_SYSTEM_V010
- CHAM_AVR_TWI_SPI_DRV_V010
- CHAM_AVR_NTSC_DRV_V010
- CHAM_AVR_VGA_DRV_V010
- CHAM_AVR_SOUND_DRV_V010

General Requirements: Video port connected to NTSC TV monitor.

Controls Required: None.

Technical Overview

This demo uses the Sound API to play tones on the speaker. You need to hook up the Chameleon's audio out port to an amplifier or the TV's audio in. However, the program outputs to both the NTSC and VGA display for its text output. This program is the most complex yet since it deals with sound. First off, we need a scale of frequencies which is at the top of the program and shown below:

```
float adjacent_note = 1.059463094; // 2^(1/12) ratio between adjacent keys

// scale from middle A to one octave above A
int scale[] = {440, 466, 494, 523, 554, 587, 622, 659, 698, 740, 784, 831, 880 };

/*
A          440
B flat    466
B          494
C          523
C sharp   554
D          587
D sharp   622
E          659
F          698
F sharp   740
G          784
A flat    831
A          880
*/
```

The array **scale[]** holds the scale from middle A at 440Hz then from this we can compute any octave or half notes, etc. which are used to generate chords. There are all kinds of chords and all kinds of theory about what sounds good and doesn't. One method of computing chords is to play 2-3 notes that are related to each other by adjacent notes from the base note. There are formulas you can use to calculate these notes and simply experiment to see what sounds "good", "dissonant", "resonant" or just plain bad. The program plays a number of chords in an increasing fashion, then in decreasing. Below is a copy of the increasing code, so you can see the computation of the chord frequencies as well as the sound API calls to play the actual chord.

```
NTSC_Term_Print("Going up...");
NTSC_Term_Char( 0x0D );

// play the scale up
for (note = 0; note < NUM_NOTES; note++)
{
    // compute chord to play
    chord_freq_1 = scale[ note ]/2 * pow ( adjacent_note, 4);

    // play these an octave above
    chord_freq_2 = scale[ note ] * pow ( adjacent_note, 5);
    chord_freq_3 = scale[ note ] * pow ( adjacent_note, 6);

    sprintf(sbuffer,"Playing Chord [%d, %d, %d]", (int)chord_freq_1, (int)chord_freq_2, (int)chord_freq_3 );

    // play the notes (we have a freq limit of 2049 Hz with the current API interface,
    // the driver can do whatever you want though
    // so you will have to create new messages and a 16-bit format if you want to play higher frequencies
    Sound_Play(0, (int)chord_freq_1, 1);
    Sound_Play(1, (int)chord_freq_2, 1);
    Sound_Play(2, (int)chord_freq_3, 1);

    // print on NTSC terminal screen
    NTSC_Term_Print(sbuffer);
    NTSC_Term_Char( 0x0D );

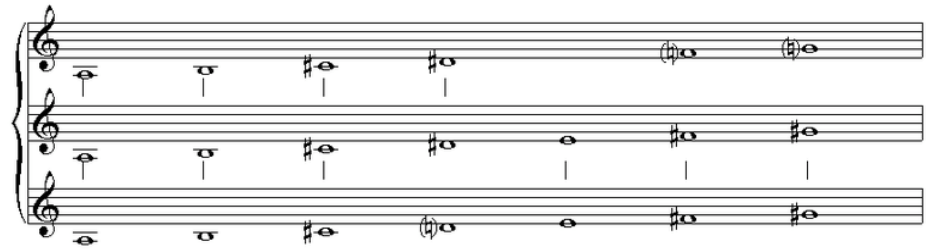
    // print on VGA terminal screen
    VGA_Term_Print(sbuffer);
    VGA_Term_Char( 0x0D );

    // slow things down a bit, so we can read the text!
    _delay_ms(500);
} // end for index
```

The comments should explain everything that is going on, but make sure to pay attention to the highlighted lines of code that compute frequencies using the **pow(...)** function. Basically, to compute the nth adjacent note on a standard diatonic scale you multiply the key/note frequency by $2^{(1/12)}$ taken to the nth power. Once the chord frequencies are computed they are issued to the sound driver to play on 3 different channels (leaving one left).

Summary

This demo shows how to play music using the sound API. However, be aware the sound driver running on the Chameleon's Propeller chip can do a lot more than we expose with the API, so if you want more features you can add them by modifying the Propeller's master driver.



30.1 Input Device Demos

The following demos illustrate the input devices supported on the Chameleon; the keyboard and mouse. The demos drive either the NTSC, VGA or both. But, if you want to add support for the other, it's simply a matter of adding a couple lines of code.

The examples typically consist of the primary source file for the demo as well as:

- The System API library module **CHAM_AVR_SYSTEM_V010.c|h**.
- The main SPI API library module **CHAM_AVR_TWI_SPI_DRV_V010.c|h**.
- The keyboard API library module **CHAM_AVR_KEYBOARD_DRV_V010.c|h**.
- The mouse API library module **CHAM_AVR_MOUSE_DRV_V010.c|h**.
- The NTSC, VGA, GFX driver or all of them.

And any other ancillary drivers keyboard, mouse, etc.

All the required files can of course be found in the **\Source** directory on the DVD (which you should have already copied onto your hard drive) located here:

**DVD-ROM:\ CHAM_AVR \ SOURCE **

AVRStudio TIP

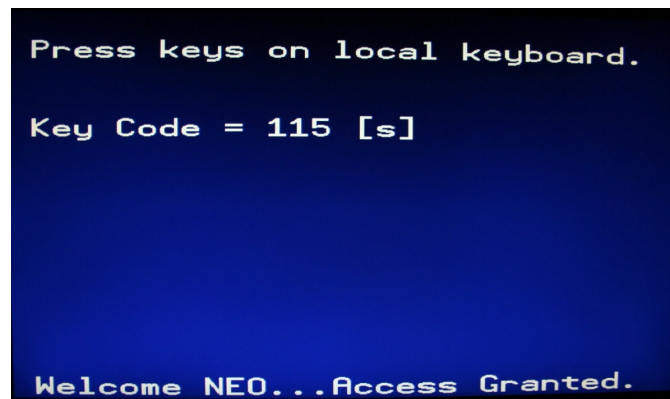
In general, you will include all the sources in your project (.C files) and make sure all the .H header files are in the same working directory, so the compiler can find them. You do NOT include the .H header files in the compilation list of source files, the compiler will do this for you. Only include .C and .S files in your source file link list in your project file list to the left of the tool.

The Arduino version will be in **"Sketch"** form already, so all you need to do is load the Sketch in and upload and you're in business!

30.1.1 Keyboard Demo

This demo uses the keyboard API to communicate with a local PS/2 keyboard plugged into the PS/2 port via the Propeller's keyboard driver. Make sure to plug your keyboard into the Chameleon and get a good connection and make sure the **<NUM-LOCK>** etc. keys are released and not in the down position. The demo prints keys on the screen as you hit them along with their codes.

Also, the demo has a little bit of fun code, it has a little password finder. The program has a state machine that is hunting for the token **"morpheus"** from the movie **"The Matrix"**, if you type this in (lower case) the program will print something out. Figure 30.1 shows the demo in action.

Figure 30.1 – Keyboard demo in action.

Compiling and Running the AVRStudio Version

Demo Version Description: Echoes the local PS/2 Chameleon keyboard to the displays.

Main Source File: CHAM_AVR_KEYBOARD_DEMO_01.C

Additional Files Required:

- CHAM_AVR_SYSTEM_V010.c|h
- CHAM_AVR_TWI_SPI_DRV_V010.c|h
- CHAM_AVR_NTSC_DRV_V010.c|h
- CHAM_AVR_VGA_DRV_V010.c|h
- CHAM_AVR_KEYBOARD_DRV_V010.h

General Requirements: Video ports connected to NTSC or VGA monitors.

Controls Required: PS/2 keyboard plugged into Chameleon, simply type into the keyboard and watch the output on the NTSC/VGA displays.

Compiling and Running the Arduino Version

Demo Version Description: Echoes the local PS/2 Chameleon keyboard to the displays.

Main Sketch Source File: CHAM_AVR_KEYBOARD_DEMO

Additional Library Files Required (imported automatically by sketch)

- CHAM_AVR_SYSTEM_V010
- CHAM_AVR_TWI_SPI_DRV_V010
- CHAM_AVR_NTSC_DRV_V010
- CHAM_AVR_VGA_DRV_V010
- CHAM_AVR_KEYBOARD_DRV_V010

General Requirements: Video ports connected to NTSC or VGA monitors.

Controls Required: PS/2 keyboard plugged into Chameleon, simply type into the keyboard and watch the output on the NTSC/VGA displays.

Technical Overview

This demo uses the single keyboard API call **Keyboard_Key()** to read the keyboard. It then displays the key both integer as well as ASCII format. The listing below is an excerpt of the **main()** program:

```

int main()          // run over and over again
{
int key;           // key that keyboard returns
char sbuffer[64]; // printing buffer

// initialize and set SPI rate
SPI_Init( SPI_DEFAULT_RATE );

// give Propeller driver time to boot
_delay_ms( 2500 );

// clear screens
NTSC_ClearScreen();
VGA_ClearScreen();

// set NTSC color
NTSC_Color(0);
NTSC_Term_Print(" PS/2 Keyboard Demo");
NTSC_Term_Char( 0x0D );
NTSC_Term_Print(" Press keys on local keyboard.");
NTSC_Term_Char( 0x0D );

// set VGA color
VGA_Color(0);
VGA_Term_Char( 0x0D );
VGA_Term_Print(" Press keys on local keyboard.");
VGA_Term_Char( 0x0D );

// load keyboard driver (this driver is already loaded by default, but this call can't hurt)
Keyboard_Load();

// enter infinite loop...
while(1)
{
    // read next key, return 0 (null if no key), wait for key press
    while ( (key = keyboard_key()) == 0);

    // build display string
    if (key >= 32)
        sprintf(sbuffer, "Key Code = %d [%c]          ", (int)key, (char)key);
    else
        sprintf(sbuffer, "Key Code = %d [non-printable] ", (int)key);

    // position cursor
    NTSC_SetXY(1, 4);

    // print on NTSC terminal screen
    NTSC_Term_Print(sbuffer);
    NTSC_Term_Char( 0x0D );

    // position cursor
    VGA_SetXY(1, 4);

    // print on VGA terminal screen
    VGA_Term_Print(sbuffer);
    VGA_Term_Char( 0x0D );

    //////////////////////////////////////
    // little string compare state machine for fun, when user types in access code
    // it displays a little message at bottom
    if (key == access_code[ access_code_index ])
    {
        // consume character, test for match?
        if (++access_code_index == strlen(access_code))
        {
            NTSC_SetXY(1, 22);
            NTSC_Term_Print("Welcome NEO...Access Granted.");

            VGA_SetXY(1, 14);
            VGA_Term_Print("Welcome NEO...Access Granted.");

            access_code_index = 0;

        } // end if match

    } // end if
    else // reset access code, start over
        access_code_index = 0;
    //////////////////////////////////////

    // slow things down a bit, so we can read the text!
    _delay_ms(10);

} // end while
} // end main

```

I have highlighted the little scanner that looks for the sequence "morpheus". Try typing it in and see if it works!

Summary

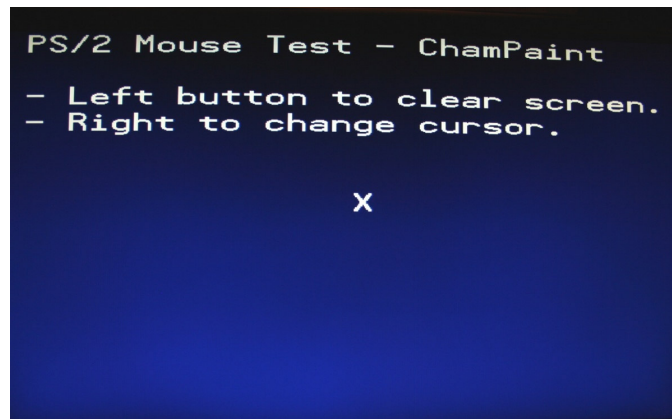
This demo shows off how easy it is to access the PS/2 keyboard. If you have done work with embedded systems, reading a PS/2 keyboard is always a challenge since it's such a slow device. Thus, the best solution is to use interrupts to detect bits sent by the keyboard. However, this usually means that any other processes that are running on a high speed interrupt will be interrupted which is tricky. However, with the Propeller doing all the work on a separate processing core, reading the keyboard is trivial and requires **no processing load** at the master's end.



30.1.2 Mouse "ASCII Paint" Demo

This demo uses the mouse API to communicate with a local PS/2 mouse plugged into the PS/2 port via the Propeller's mouse. Make sure to plug your mouse into the Chameleon and get a good connection. This crude "**ASCII Paint**" demo allows you to draw with the mouse. The right mouse button changes the ASCII drawing character and the left mouse button clears the screen. The program displays simultaneously on both the NTSC and VGA displays. Figure 30.2 shows the demo in action.

Figure 30.2 – Mouse "ASCII Paint" demo in action.



Compiling and Running the AVRStudio Version

Demo Version Description: Draws on the screen with the mouse using ASCII art.

Main Source File: CHAM_AVR_MOUSE_DEMO_01.C

Additional Files Required:

- CHAM_AVR_SYSTEM_V010.c|h
- CHAM_AVR_TWI_SPI_DRV_V010.c|h
- CHAM_AVR_NTSC_DRV_V010.c|h
- CHAM_AVR_VGA_DRV_V010.c|h
- CHAM_AVR_MOUSE_DRV_V010.h

General Requirements: Video ports connected to NTSC or VGA monitors.

Controls Required: PS/2 mouse plugged into Chameleon, draw ASCII art with the mouse. Left button to clear the screen, right button to cycle thru different drawing characters.

Compiling and Running the Arduino Version

Demo Version Description: Draws on the screen with the mouse using ASCII art.

Main Sketch Source File: CHAM_AVR_MOUSE_DEMO

Additional Library Files Required (imported automatically by sketch)

- CHAM_AVR_SYSTEM_V010
- CHAM_AVR_TWI_SPI_DRV_V010
- CHAM_AVR_NTSC_DRV_V010
- CHAM_AVR_VGA_DRV_V010
- CHAM_AVR_MOUSE_DRV_V010

General Requirements: Video ports connected to NTSC or VGA monitors.

Controls Required: PS/2 mouse plugged into Chameleon, draw ASCII art with the mouse. Left button to clear the screen, right button to cycle thru different drawing characters.

Technical Overview

This demo uses the ***Mouse_Read(...)*** API call to read the state of the mouse, it uses this to allow you to draw on the NTSC/VGA screen with ASCII art characters. The main ***while()*** loop within the ***main()*** itself is shown below, this is where all the action happens.

```
// main event loop
while(1)
{
    // read the mouse event (if one)
    Mouse_Read(&event);

    // compute position
    event.x = event.x / 10;
    event.y = event.y / 10;

    // left button hit? Clear screen?
    if (event.buttons == 0x01)
    {
        // clear the surface
        NTSC_ClearScreen();
        VGA_ClearScreen();
    } // end if

    // right button hit? update cursor character?
    if (event.buttons == 0x02)
    {
        // update the cursor character
        if (++cursor >= (32+144))
            cursor = 33;

    } // end if

    // motion clamping
    if (event.x < -2)
        event.x = -2;
    else
        if (event.x > 2)
            event.x = 2;

    if (event.y < -2)
        event.y = -2;
    else
        if (event.y > 2)
            event.y = 2;

    // update position
    x += event.x;
    y -= event.y;

    // draw on both screens
    NTSC_SetXY( x, y );
    NTSC_Term_Char( cursor );

    VGA_SetXY( x, y/2 );
    VGA_Term_Char( cursor );

    // slow things down a bit
```

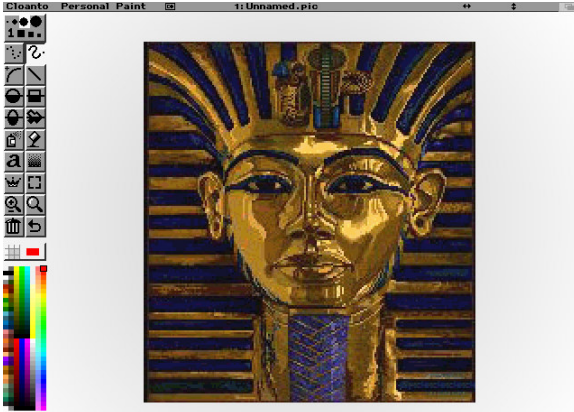
```

_delay_ms( 5 );
} // end while

```

I have highlighted the single call to the mouse read function. One thing to note is that the mouse read function takes a pointer to an **gid_event** structure, this is a generic structure that can potentially be used for other input devices such as game controllers, track balls etc.

Summary



This demo once again shows how easy it is to interface peripherals to the Chameleon with the Propeller doing all the work. Something that might be fun to do is add a bitmap NTSC or VGA driver to the master Propeller driver and then create a true paint program that allows pixel level painting, filling, etc. and create something like the famous **DPaint** that we used to use to make games in the 80's and 90's!

31.1 Serial, FLASH, and Port I/O Device Demos

The next group of demos illustrate the Serial port, FLASH memory, and Propeller Port I/O functionality. As usual, the demos try to do something graphical, so they always support NTSC output and some might support VGA output as well. But, they all usually use nothing more than text terminal graphics, so added VGA support is a couple lines of code.

The examples typically consist of the primary source file for the demo as well as:

- The System API library module **CHAM_AVR_SYSTEM_V010.c|h**.
- The main SPI API library module **CHAM_AVR_TWI_SPI_DRV_V010.c|h**.
- The serial API library module **CHAM_UART_DRV_V010.c|h**.
- The FLASH API library module **CHAM_FLASH_DRV_V010.c|h**.
- The Propeller local 8-bit port API library module **CHAM_PROP_PORT_DRV_V010.c|h**.
- The NTSC, VGA, GFX driver or all of them.

And any other ancillary drivers keyboard, mouse, etc.

All the required files can of course be found in the **\Source** directory on the DVD (which you should have already copied onto your hard drive) located here:

**DVD-ROM:\ CHAM_AVR \ SOURCE **

AVRStudio TIP

In general, you will include all the sources in your project (.C files) and make sure all the .H header files are in the same working directory, so the compiler can find them. You do NOT include the .H header files in the compilation list of source files, the compiler will do this for you. Only include .C and .S files in your source file link list in your project file list to the left of the tool.

The Arduino version will be in **"Sketch"** form already, so all you need to do is load the Sketch in and upload and you're in business!

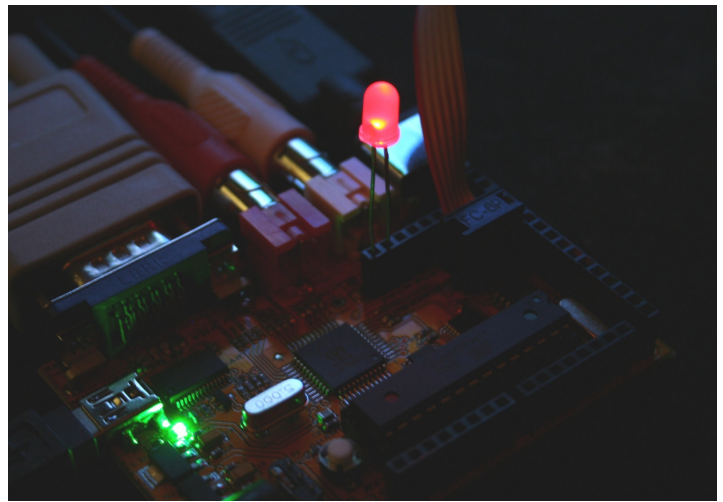
31.1.1 Propeller Local Port LED Blinker Demo

The Propeller chip on the Chameleon had 8 free I/O pins, so we decided to add an 8-bit port to it. With this, you can interface SD cards, SPI memory, control servos, whatever. Of course, the AVR itself has all its I/O headers. But, give the Propeller a direct port itself allows drivers running on the Propeller to do more stuff. Anyway, that's what the Propeller 8-bit port is all about. And this simple demo shows how to set the I/O pin directions and blink an LED. All we need is the Propeller Port API library and an LED to plug into the Propeller port – which we happen to supply with your Chameleon. Now, you might have a red, green, blue, or bi-color LED. Doesn't matter which, just plug it into the 2-left most pin headers of the Propeller Port as shown in Figure 31.1 below.

EE TIP

If you are into electronics, you might be concerned that we are plugging the LED directly into two I/O ports and it might burn the LED out? This is a valid concern, but luckily, the forward voltage of the LED is about 2.2 – 2.5V, so the 3.3V outputs of the Propeller are just enough to turn it on and the LED will draw a limited amount of current. Moreover, the Propeller I/O ports can actually source or sink 40ma, so you can even drive little light bulbs. Your second concern might be there is no ground on the 8-bit port, so how does this work? Well, when we set an I/O to digital 0, it's at a virtual ground and being pulled to GROUND by a transistor switch, so as long as the amount of current your draw thru the circuit isn't too much, the virtual ground will remain very close to 0. If you draw a lot of current then the ground will "rise" or "float" up due to the voltage drop over the transistor switch pulling the pin LOW.

Figure 31.1 – A screen shot of the NTSC output (left) and the actual LED blinking on the Chameleon (right).



Compiling and Running the AVRStudio Version

Demo Version Description: Blinks and LED on the Propeller Port.

Main Source File: CHAM_AVR_PROPPORT_DEMO_01.C

Additional Files Required:

- CHAM_AVR_SYSTEM_V010.c|h
- CHAM_AVR_TWI_SPI_DRV_V010.c|h
- CHAM_AVR_NTSC_DRV_V010.c|h
- CHAM_AVR_PROP_PORT_DRV_V010.h

General Requirements: Video port connected to NTSC TV monitors, LED plugged into the Propeller Port, leftmost 2-pins, orientation doesn't matter.

Controls Required: None.

Compiling and Running the Arduino Version

Demo Version Description: Blinks and LED on the Propeller Port.

Main Sketch Source File: CHAM_AVR_PROPPORT_DEMO

Additional Library Files Required (imported automatically by sketch)

- CHAM_AVR_SYSTEM_V010
- CHAM_AVR_TWI_SPI_DRV_V010
- CHAM_AVR_NTSC_DRV_V010
- CHAM_AVR_PROP_PORT_DRV_V010

General Requirements: Video port connected to NTSC TV monitors, LED plugged into the Propeller Port, leftmost 2-pins, orientation doesn't matter.

Controls Required: None.

Technical Overview

This demo uses the Propeller Port API library to control the pins of the 8-bit local Propeller Port and blink an LED. A couple cool things about the demo is that it toggles the pins on the I/O header +- in pairs then -+, thus no matter which way you have the LED nor what pins you have it plugged into, it will actually blink or go bi-color (if you were lucky and got a bi-color LED in your kit). Below is the entire *main()* loop of the program.

```
void main()      // run over and over again
{
  // initialize and set SPI rate
  SPI_Init( SPI_DEFAULT_RATE );

  // give Prop a moment before sending it commands, the boot process is about a second
  // if you like you can speed it up, by removing the Prop driver's "LED blink" sequence in the driver
  // or speed it up, but commands that are sent before the Prop driver is done booting will be ignored.
  // therefore, if you have a lot of set up work to do, then you don't require a delay, but if you jump right
  // into commands, then you need a good 1.5 - 2 second delay
  _delay_ms( 2500 );

  // clear screens
  NTSC_ClearScreen();
  NTSC_Color(0);

  // set direction of prop port so we have all bits output
  PropPort_SetDir( 0b11111111 ); // 1 is output, 0 input

  // enter infinite loop...
  while(1)
  {
    // print on NTSC terminal screen
    NTSC_Term_Print("Port Data ON [+ - + - + - + -]");
    NTSC_Term_Char( 0x0D );

    // turn on the LED pin 7 = HIGH, pin 6 = LOW, repeat pattern down port.
    PropPort_Write(0b10101010);

    // slow things down a bit, so we can read the text!
    _delay_ms(250);

    // print on NTSC terminal screen
    NTSC_Term_Print("Port Data OFF[- + - + - + - +]");
    NTSC_Term_Char( 0x0D );

    // turn off the LED pin 7 = LOW, pin 6 = HIGH, repeat pattern down port.
    PropPort_Write(0b01010101);

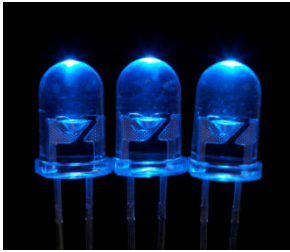
    // slow things down a bit, so we can read the text!
    _delay_ms(250);

  } // end while
} // end main
```

I have highlighted the port I/O commands. The program starts by setting the direction bits to all 1's which mean output. Then the program immediately enters into the main loop and starts toggling the I/O pins on a 500ms total cycle. The output is mimicked to the NTSC screen, you can add VGA support yourself with a few lines of code. Just don't forget to add the library and header files to your program.

Now, once thing you might wonder is **"how fast can I control these pins?"**. This is a good question since you might want to do something like talk to a SPI or I2C device. So, the real bottleneck is the SPI interface from the AVR/PIC chip to the Propeller, we can only send commands and thus toggle the I/O pins as fast as this interface. Thus, if you want to do something more complex, then I suggest that you add commands and messages to control the Propeller's onboard counters – then you can route them to the I/O pins and generate any signal you want via control from the master AVR/PIC chip.

Summary



This demos shows off a really simple way to control some of the I/O pins of the Propeller chip. Also, the code is so simple there is no processing core for it on the Propeller, it runs right on the main message dispatcher loop and just does the pin I/O on demand. Something you should try is hooking up a SPI, I2C, or other device to the port and see if you can remotely control it all the way from the AVR/PIC chip!

31.1.2 Serial RS-232 Communications Demo

This demo shows off the serial communications of the Chameleon. There are three components to this:

- The hardware UART in the AVR 328P.
- The FTDI USB to Serial UART converter.
- The software driver running on the AVR.

Considering these three elements; to get serial from the PC's USB port to the Chameleon, first we need the FTDI USB to serial UART which we have been using. Secondly, to communicate with the AVR, we need to use its internal UART (the 328P has only a single UART by the way). And finally, we need to write software to control the AVR's UART and set it up. This is what the **CHAM_AVR_UART_DRV_V010** library module is for.

TIP

Although, we have been using the USB to serial connection to the Chameleon, we don't have to use the USB port, the serial TX/RX pins from the AVR are exported to the expansion headers where you can directly connect them to a TTL level serial device. However, in most cases, the only serial device you need to connect to the Chameleon is the PC, and thus the USB to serial connection suffices for most applications.

Setting up the Terminal Program

If you are an expert at setting up serial terminals and connecting serial devices then you can skim this section, but if you're not then follow along. There are two parts to this demo:

- The physical connection from the Chameleon AVR's serial port to the PC's serial port.
- Running a VT100 serial terminal program on the PC with settings N81, 115,200 baud.

The first problem is fairly straightforward, simply connect your Chameleon AVR to your PC with the mini-USB serial cable. The FTDI driver on the PC's side virtualizes the COM port, so from the PC's point of view its just another serial port.

Once you have the physical connection made then the next step is to get a terminal program up and running on your PC. There are literally dozens if not hundreds of serial communications programs that you can find on internet, 99% of them are free for a reason -- they are junk. There are a few that are actually very good such as:

- PuTTY - <http://www.chiark.greenend.org.uk/~sgtatham/putty/>
- Zoc - <http://www.emtec.com/zoc/>

- Absolute Telnet - <http://www.celestialsoftware.net/telnet/>

You can also find them on DVD located here :

DVD-ROM:\CHAM_AVR\TOOLS

VT100 terminal running, the Chameleon's serial port connected to the PC with the following settings on your terminal:

- Data Format: "N81", no parity, 8 data bits, 1 stop bit.
- Baud Rate: 115,200.
- Handshaking: None.
- Terminal Mode: VT100.
- Screen Size: 80x24 or larger, 120x50 is a good choice.
- Local Echo: Off.
- Make sure that you have the serial selection switch in the **DOWN** position on the Chameleon (AVR mode).

What we are going to focus on here is taking full advantage of the serial UART code and making a little **"Guess My Number"** game that runs on the Chameleon AVR, but communicates via serial to a VT100 terminal running on the PC. Figure 31.2 shows the demo program running.

Figure 31.2 – "Guess My Number" running on the PC terminal (need new image)

```

COM31 - PuTTY
XGS WarGames (C) 2019

# # ##### ##### ##### ##### #####
# # # # # # # # # # # # # # #
# # ##### ##### # ## ##### # # ## #####
# # # # # # # # # # # # # # #
##### # # # # ##### # # # # #####

Hello Dr. Falkner...

Just a moment while I initialize my systems:
Holographic Memory:OK
Multiprocessor Sub-Systems:OK
Quantum Telecom Links:OK
Loading Autonomous Systems.....

That's much better.

Would you like to play a game (yes or no)?y
  
```

The screen shot doesn't do the demo justice since there are sound effects and ASCII animation that occur. When you run the demo, you might have to hit **RESET** on the Chameleon a couple times to really appreciate the **"retro"** feel of the demo.



What I did was play a bit on the 1980's classic **"WarGames"** starring Mathew Broderick. It's one of my favorite movies, and if you haven't seen it, definitely fun to watch. The movie's plot was a teenage kid tried to hack into some Silicon Valley computer game company, but instead modem'ed into the "WOPR" which stood for **"War Operations Planned Response"** which was a computer running the entire USA missile system. The computer wants to play a game and the story begins from there. The whole point of this stroll down memory lane is that computers used to be magical devices that opened up whole worlds to people in a much more personal way than they do today. And text based games used to be the conduit of much of this interaction with computers. So even with a simple terminal program you can create something with the serial port that's kinda cool If you're interested in the computer used in the movie, its too is very famous – the IMSAI, only a few left in operation. Here's a link to more information:

<http://www.imsai.net/movies/wargames.htm>

Compiling and Running the Demo(s)

There are two demos in this group; the AVRStudio version and the Arduino version. They are nearly identical except that for the AVRStudio version serial communications uses the **CHAM_AVR_UART_DRV_V010.c|h** driver that we created, but the Arduino version uses the Arduino serial class. Nearly, the same and I could have used the Chameleon serial driver in the Arduino version, but wanted to show you two fairly complex programs that use serial communications using each driver respectively.

Compiling and Running the AVRStudio Version

Demo Version Description: Serial UART demo, illustrates text output and input using a game as the platform.

Main Source File: CHAM_AVR_GUESS_NUMBER_01.c

Additional Files Required:

- CHAM_AVR_SYSTEM_V010.c|h
- CHAM_AVR_UART_DRV_V010.c|h
- CHAM_AVR_TWI_SPI_DRV_V010.c|h
- CHAM_AVR_SOUND_DRV_V010.c|h

General Requirements: Serial port connection from Chameleon AVR to PC over the USB cable, terminal programming running, RCA sound output connected to speaker or TV audio.

Controls Required: None, uses PC keyboard via terminal program.

Compiling and Running the Arduino Version

Demo Version Description: Serial UART demo, illustrates text output and input using a game as the platform.

Main Sketch Source File: CHAM_AVR_GUESS_NUMBER

Additional Library Files Required (imported automatically by sketch)

- CHAM_AVR_SYSTEM_V010.c|h
- CHAM_AVR_UART_DRV_V010.c|h
- CHAM_AVR_TWI_SPI_DRV_V010.c|h
- CHAM_AVR_SOUND_DRV_V010.c|h

General Requirements: Serial port connection from Chameleon AVR to PC over the USB cable, terminal programming running, RCA sound output connected to speaker or TV audio.

Controls Required: None, uses PC keyboard via terminal program.

The game starts by asking you if you wish to play, you can answer "yes" or "no", each has its consequences. If you do decide to play, then the computer selects a random number from 1 to 100 and asks you to guess it. Just play along and see if you can guess. Each guess you type in and hit **<RETURN>**.

Technical Overview

This demo really shows off the UART and bi-directional communications. Not only are we printing to the UART for display on the terminal on the PC, but we are getting input from the terminal and processing it, so it's a very complete two way communications demo. Also, there are some animations and fun techniques that are used to display the logon logo and other effects to "sell" the game. For example, the intro logo is an array of data that is used to print out "WarGames":

```
// the title screen bitmap as a matrix of byte, stored in FLASH for fun, then to render we will convert to
// characters: BLOCK and SPACE
// notice the use of "const" and "PROGMEM", also to access the array, we must use "pgm_read_byte( addr )"
// size of banner 52x5
const unsigned char title_string_flash[] PROGMEM = {
0,1,0,0,0,1,0,1,1,1,1,1,0,1,1,1,1,1,0,0,0,1,1,1,1,0,0,1,1,1,1,1,0,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,0,
0,1,0,0,0,1,0,1,0,0,0,1,0,1,0,0,1,1,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,1,0,0,1,0,0,1,0,0,0,0,1,0,0,0,0,0,
0,1,0,0,0,1,0,1,1,1,1,1,0,1,1,1,1,0,0,0,0,1,0,0,1,1,0,1,1,1,1,1,1,0,1,0,0,0,0,1,0,1,1,1,1,1,1,0,
0,1,0,1,0,1,0,1,0,0,0,1,0,1,0,0,1,0,0,0,0,1,0,0,0,1,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,1,0,
0,1,1,1,1,1,0,1,0,0,0,1,0,1,0,0,0,1,0,0,0,1,1,1,1,1,0,1,0,0,0,0,0,1,0,1,1,1,1,0,1,1,1,1,1,0,
};
```

The interesting thing about this array is that its defined in FLASH memory rather than SRAM, so this demo shows how you do that. Notice the keyword **PROGMEM** after the array declaration? This instructs the compiler to place the array in FLASH rather than SRAM. However, this isn't without side effects. Now that the data is in FLASH, you have to use special functions and techniques to access the FLASH data, but this is more of an inconvenience than anything else. For example, later in the code when the logo is displayed, the following function reads the data, displays it on the terminal, and makes some sounds:

```
// draw "WAR GAMES" with characters from FLASH
for (index = 0; index < 52*5; index++)
{
    // read the image from FLASH memory with special function call that uses
    // "LPM" ASM instruction to read FLASH
    c = pgm_read_byte( &title_string_flash[ index ] );

    // based on data convert to character and send to UART
    if (c == 0)
    {
        Sound_Play(0);
        UART_Print_String(" ");
    }
    else
    if (c == 1)
    {
        Sound_Play(2000);
        UART_Print_String("#");
    } // end if

    if ( (index % 52) == 0)
        UART_Print_String("\n");

    _delay_ms(50);
} // end for
```

I have highlighted the special function needed to access FLASH based storage, the **pgm_read_byte(...)** function. This is all you need to get to the bytes of the FLASH with a pointer.

The remainder of the main program is more or less printing strings with timing and a bit of conditional logic; however, there is one function that is the workhorse of the program and that's the **Get_String(...)** function. When programming, most programmers are completely oblivious to the underlying code in **printf(...)** or **scanf(...)** and so forth, but in embedded systems we can't use these calls since they usually have no meaning and we need to write them ourselves! Alas, in this case, we need some kind of simple single line text editor, so the user can type on the screen, backspace, etc. and do a few reasonable "edit" related keystrokes. The terminal program has no idea about this and nor does the UART, the UART just receives ASCII characters, thus we need to write a single line text processor to make this work. The function listing of this code is below.

```

int Get_String( char *cmd_buff, char **tokens )
{
///////////////////////////////////////////////////////////////////
// this function waits for the user to input a string on from the UART, it supports some
// editing such as backspace. When the user hits "return" the string is tokenized into an array
// of strings **tokens. The number of tokens extracted are returned by the function
// INPUTS:
//      *buffer = pointer to string storage where the raw ASCII string from the UART is placed
//      **tokens = an array of pointers where the function puts the tokenized strings
//
// OUTPUTS:
//      returns the number of tokens.
///////////////////////////////////////////////////////////////////

int c;
char token_buff[64];
int num_tokens = 0;
int next_token = 0;
int cmd_buff_index = -1;
char *token_ptr = NULL;
char *result_ptr;

// enter infinite while until valid string is entered
while(1)
{
// process next character
if ( (c = UART_getc()) != -1 )
{
// special characters
switch (c)
{
case 0x0D: // carriage return
case 0x0A: // line feed
{
cmd_buff[++cmd_buff_index] = 0;

// any command to process?
if (cmd_buff_index > 0)
{
// process command...
// upcase the string to normalize tokens
strupr( cmd_buff );

// copy cmd_buffer to token_buffer for processing
memcpy(token_buff, cmd_buff, strlen( cmd_buff ) + 1);

// initialize command processor
num_tokens = 0;

// initialize token processor
token_ptr = strtok_r( token_buff, "=", " ", &result_ptr);

while( token_ptr )
{
// insert token into token array and inc number of tokens found
tokens[ num_tokens++ ] = token_ptr;
token_ptr = strtok_r( NULL, "=", " ", &result_ptr);
} // end while

// string has been entered and tokenized, now return number of tokens
return( num_tokens );
} // end if

// empty string
return (-1);

} break;

case 0x08: // back space
{
if (cmd_buff_index >= 0 )
{
UART_putc( c );
cmd_buff_index--;
} // end if
} break;

default: // otherwise normal character, insert into command string and echo
{
UART_putc( c );
cmd_buff[++cmd_buff_index] = c;
} break;
} // end switch
} // end if
} // end while
} // end Get_String

```

This function is a bit overkill since its designed to parse the string into **"tokens"** that you might need later, but it's a good starting point for you if you want to break strings apart that come in via the UART. The bottom line is text parsing isn't as easy as you would think and when you have to write editors that run over terminal programs there are a lot of issues you need to resolve.

Summary

Hopefully, you have a good command of the serial driver now and you can use it to communicate with the PC or any other serial based device. There are all kinds of interesting things you can do with a serial port and the Chameleon AVR, for example you could get an external serial modem and write a simple BBS (bulletin board system) on the Chameleon AVR. How cool would that be? For example, the **"TRENDnet 56k (V.92) High Speed Voice/Fax Modem"** has a standard RS-232 DB9 connector on it, so you can connect quite easily to the Chameleon AVR and send standard HAYES AT commands at it. Can you imagine running a BBS on your little Chameleon and then dialing in from a friends house? Here's a link to the product:

<http://www.tigerdirect.com/applications/SearchTools/item-details.asp?EdpNo=671884&CatId=564>

31.1.3 FLASH Memory Demo (with X-Modem Protocol Bonus Example)

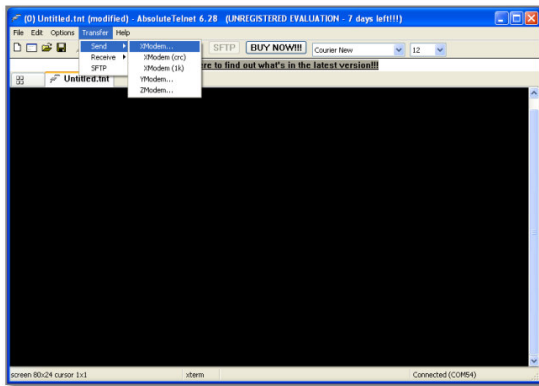
This demo illustrates how to use the onboard 1MB SPI FLASH memory (Atmel part #T26DF081A). This is not to be confused with the Propeller chip's 64K EEPROM. The EEPROM is for the Propeller's boot image and only directly accessible by the Propeller chip. The 1 MB FLASH is accessible by the AVR chip via the SPI mux interface. With that in mind, you might want review the Chameleon API as well as the data sheet for the FLASH memory located on the DVD here:

DVD-ROM:\ CHAM_AVR \ DOCS \ DATASHEETS \ Atmel_AT26DF081A_FLASH_doc3600.pdf

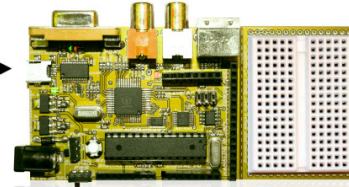
To use the FLASH API is relatively easy, you open the FLASH memory up, then read or write, then close it. Basically, four API calls to do everything. However, that's pretty boring, so we decided to do something a little more interesting. We were concerned that many newbies would have a hard time putting the FLASH to use since there isn't a direct way to access it from the PC. For example, if you could read and write files to it over the USB port that would be ideal. Moreover, if the Chameleon could act like a FLASH FAT drive that would be great. But, no one has written software to do that, and that's a good 1-2 weeks work, so if you're up to it please do – However, as a conciliation prize, we developed a little **"X-Modem"** protocol driver that allows you to download files (of any type) from the PC to the Chameleon into a RAM buffer. Then you can do what you want with the RAM buffer like write it to the FLASH memory! Figure 31.3 shows a flow diagram of what's going on.

Figure 31.3 – The various elements in an X-Modem file transfer.

Absolute Telnet (any X-Modem protocol program).

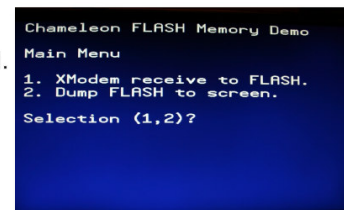


Chameleon AVR/PIC
(listening to USB serial port).



X-Modem protocol file transfer.
Data is sent a block at a time.
Each block consists of header
information along with the 128
byte data. Serial settings for
communications are N81, 2400 baud.

Receiver (Chameleon) controls the
flow of data from the sender (PC).



FLASH demo running on Chameleon
(Simple X-Modem downloader).

X-Modem Protocol

If you have never heard about X-modem protocol, it's probably the most widely used modem to modem download protocol on the planet and still widely used. There are a number of variants that have been developed over the years such as X-modem 1K, X-Modem CRC, Y-Modem, Z-Modem and so forth, but fundamentally they all work the same way; the **"uploader"** sends packets (usually 128-256 bytes) to the **"downloader"** until the entire file has been sent. The uploader is typically the PC and the downloader is typically another PC that needs the file or a stand alone device.

X-Modem protocol is rather simple, but a bit tricky to implement robustly since there are lots of potential forks in the communications logic that have to do with packets being lost, errors, resending, timeouts, etc. If you have ever written and communications protocols you really understand how complex getting them to work robustly it. You can read about X-Modem protocol by Googling for it, but here are a couple good references online:

<http://en.wikipedia.org/wiki/XMODEM>

<http://www.techheap.com/communication/modems/xmodem.html>

So, what we did was take the protocol and make a simple downloader / receive function that initiates an X-Modem transfer download over the serial port. The function assumes that the uploader is already running and waiting for the downloader (the Chameleon) to initiate the transfer. This is a very important part of X-Modem transfer protocol; the uploader is started and it simply waits for the downloader to start requesting packets. There is actually a timeout that it will give up (in the minutes), but usually the idea is you **start** the uploader and leave it **THEN** go run the downloader and it controls the flow of information or the rate of data reception. Additionally, the downloader checks each block of information against a sent checksum (all of these is covered in the links above). But, briefly here's the packet format (or block as they like to call it):

ASCII Control Codes

```

<soh> 01H
<eot> 04H
<ack> 06H
<nak> 15H
<can> 18H

```

Each block of the transfer looks like:

```

<SOH><blk #><255-blk #><--128 data bytes--><cksum>
  in which:

```

```

<SOH>      = 01 hex
<blk #>    = binary number, starts at 01 increments by 1, and
              wraps 0FFH to 00H (not to 01)
<255-blk #> = blk # after going thru 8080 "CMA" instr.
              Formally, this is the "ones complement".
<cksum>    = the sum of the data bytes only. Toss any carry.

```

Considering the block format outlined above, the downloader needs to check the checksum against the received data and can request a packet be resent if its not correct. Also, if packets are late, etc. the downloader can time out, and the downloader can simply stop the transfer if it wishes – thus, the protocol is **receiver driven**. Please review the web links above to get a better handle on this protocol.

Now, the implementation we made was quick and dirty, no checksum checking, no timeouts, etc. it simply hopes that nothing goes wrong. On the other hand, its very easy to add things to the function to make it more robust, we purposely wrote it this way, so that you could understand the function and add to it if you wanted. Implementing the complete X-Modem protocol with all error handling cases, makes the code look like spaghetti! With that in mind, here's the function itself that does the downloading:

```

int xModem_Receive( char *file_buffer, int buffer_size )
{
// receives file from an xmodem sender, standard "XMODEM" protocol as shown here:
// http://en.wikipedia.org/wiki/XMODEM
// http://www.techheap.com/communication/modems/xmodem.html
//
// function takes a pointer to the storage area of file along with the maximum size of the buffer
// if the file exceeds the buffer, the function WILL read the file and compute its length, but only the
// first buffer_size number of bytes will be written to the file_buffer.
//
// function returns the length of the file in bytes
//
// This is a "receiver" oriented protocol, so when you want to receive a file from the PC, start the
// transmission on the PC end FIRST, THEN run your xmodem download on this side. The PC will wait a min or 2
// before timing out, but this function will immediately start sending NAKs and expect the sender to respond
// I leave it to you to make this function more "robust", I wrote it in about an hour :)
//
// note: debug information only prints to NTSC currently
//
// xmodem protocol defines
//
// state machine
#define XSTATE_INIT          0
#define XSTATE_SEND_NAK     1
#define XSTATE_RECEIVE_PACKET 2
#define XSTATE_SEND_ACK     3
#define XSTATE_EOT          4
//
// control characters used in xmodem protocol
#define CHR_SOH              1
#define CHR_EOT              4
#define CHR_ACK              6
#define CHR_NAK              21
#define CHR_CAN              24
#define CHR_CTRLZ            26
//
// control debug output to NTSC screen, this is handy if you try to add more features to the function
#define DEBUG_XMODEM        1 // 0 - no output at all, 1 - display file data and file size, 2 - display
// everything
//
int xstate, xstate_running, xbytes_read = 0, ch, index;
unsigned char xchecksum, xchecksum_computed, xpacketnum, xpacketnumn;
//
// start off in initialization state
xstate = XSTATE_INIT;
xstate_running = 1;
//
// main loop
while(xstate_running)

```

```

{
// begin x-mode state machine
switch( xstate )
{
case XSTATE_INIT:      // initialize all the variables and data structures for the xmodem transmission
{
#if DEBUG_XMODEM >= 2
    NTSC_Term_Print("XSTATE_INIT");
    NTSC_Term_Char( 0x0D );
#endif

    // reset bytes read counter, etc.
    xbytes_read = 0;

    // transition to NAK state to initiate xmode transfer
    xstate = XSTATE_SEND_NAK;

    } break;

case XSTATE_SEND_NAK:  // start the transmission, or resend current packet
{
#if DEBUG_XMODEM >= 2
    NTSC_Term_Print("XSTATE_SEND_NAK");
    NTSC_Term_Char( 0x0D );

    NTSC_Term_Print("S:NAK");
    NTSC_Term_Char( 0x0D );
#endif

    // send NAK and wait
    UART_putc( CHR_NAK );

    // wait for a packet..
    xstate = XSTATE_RECEIVE_PACKET;

    } break;

case XSTATE_RECEIVE_PACKET: // receives the packet
{
#if DEBUG_XMODEM >= 2
    NTSC_Term_Print("XSTATE_RECEIVE_PACKET");
    NTSC_Term_Char( 0x0D );
#endif

    // wait for result
    while (1) { ch = UART_getc(); if (ch == CHR_SOH || ch == CHR_EOT) break; } // end while

#if DEBUG_XMODEM >= 2
    sprintf(sbuffer,"Looking for SOT/EOT: %d %c",ch, 0x0D );
    NTSC_Term_Print( sbuffer );
#endif

    // we have the first byte, test for SOH or EOT
    if (ch == CHR_SOH)
    {
#if DEBUG_XMODEM >= 2
        NTSC_Term_Print("R:SOH");
        NTSC_Term_Char( 0x0D );
#endif
    } // end if SOH
    else if (ch == CHR_EOT)
    {
#if DEBUG_XMODEM >= 2
        NTSC_Term_Print("R:EOT");
        NTSC_Term_Char( 0x0D );
#endif
    }

    xstate = XSTATE_EOT;
    break;

    } // end if

    // get packet number
    while (1) { ch = UART_getc(); if (ch != -1 || ch == CHR_CTRLZ) break; } // end while

    xpacketnum = ch;

#if DEBUG_XMODEM >= 2
    sprintf(sbuffer,"Packet # %d\r",xpacketnum );
    NTSC_Term_Print( sbuffer );
#endif

    // get packet number complement
    while (1) { ch = UART_getc(); if (ch != -1 || ch == CHR_CTRLZ) break; } // end while
    xpacketnumn = ch;

#if DEBUG_XMODEM >= 2
    sprintf(sbuffer,"Packetn (255-packet)# %d\r",xpacketnumn );
    NTSC_Term_Print( sbuffer );
#endif

    // now get the packet

```

```

        xchecksum_computed = 0;

#if DEBUG_XMODEM >= 2
    NTSC_Term_Print("DATA:");
    NTSC_Term_Char( 0x0D );
#endif
    for (index = 0; index < 128; index++)
    {
        // wait for next byte
        while (1) { ch = UART_getc(); if (ch != -1 || ch == CHR_CTRLZ) break; } // end while

        // insert into buffer (if still room?)
        if ( (index + (xpacketnum-1)*128) < buffer_size )
            file_buffer[ index + (xpacketnum-1)*128 ] = ch;

        // update checksum
        xchecksum_computed += ch;

#if DEBUG_XMODEM >= 1
        // test for printable
        if ( (ch >= 32 && ch <= 127) || ch==0x0D)
        {
            NTSC_Term_Char( ch );
            VGA_Term_Char( ch );
        } // end if printable
        else // non-printable character
        {
            // print something for place holder?
            //NTSC_Term_Char( '.' );
            //VGA_Term_Char( '.' );
        } // end else non-printable
#endif

        } // end for index

    // add bytes read
    xbytes_read += 128;

    // read the checksum
    while (1) { ch = UART_getc(); if (ch != -1 || ch == CHR_CTRLZ) break; } // end while
    xchecksum = ch;

#if DEBUG_XMODEM >= 2
    NTSC_Term_Char( 0x0D );
    sprintf(sbuffer,"Checksum: Sent=%d, Calc=%d\r",xchecksum, xchecksum_computed );
    NTSC_Term_Print( sbuffer );
    NTSC_Term_Char( 0x0D );
#endif

    // send the ACK (test if packet was good later)
    xstate = XSTATE_SEND_ACK;

    } break;

    case XSTATE_SEND_ACK:          // packet was good, please send another?
    {
#if DEBUG_XMODEM >= 2
        NTSC_Term_Print("XSTATE_SEND_ACK");
        NTSC_Term_Char( 0x0D );

        NTSC_Term_Print("S:ACK");
        NTSC_Term_Char( 0x0D );
#endif
        _delay_ms ( 100 );

        // send ACK and wait
        UART_putc( CHR_ACK );

        _delay_ms ( 100 );

        // wait for a packet..
        xstate = XSTATE_RECEIVE_PACKET;

    } break;

    case XSTATE_EOT:              // possible end of transmission
    {
#if DEBUG_XMODEM >= 2
        NTSC_Term_Print("XSTATE_EOT");
        NTSC_Term_Char( 0x0D );

        NTSC_Term_Print("S:NAK");
        NTSC_Term_Char( 0x0D );
#endif
        // send NAK, wait for another EOT
        UART_putc( CHR_NAK );

        // wait for result
        while (1) { ch = UART_getc(); if (ch != -1 || ch == CHR_CTRLZ) break; } // end while

```

```

        // test for final EOT
        if (ch == CHR_EOT )
        {
            // final EOT, send ACK, finish
            UART_putc ( CHR_ACK );
        }
    #if DEBUG_XMODEM >= 2
        NTSC_Term_Print("s:ACK");
        NTSC_Term_Char( 0x0D );
    #endif
        // exit state machine
        xstate_running = 0;
    } // end if

    } break;

    default: break;

    } // end switch

} // end while

#if DEBUG_XMODEM >= 1
sprintf(sbuffer, "\rDownload Complete.\r%d Blocks.\rFile Size = %d.\r", xbytes_read/128, xbytes_read );
NTSC_Term_Print( sbuffer );
VGA_Term_Print( sbuffer );
#endif

// return bytes read
return ( xbytes_read );

} // end XModem_Receive

```

All that conditionally compilation code is to help you modify the function to add features to it since this is really the only way to get "data" from the PC into the FLASH memory at the moment (unless you make something yourself).

Alright, now that you have an idea about X-Modem protocol, let's take a step back and see how we are going to send X-Modem files from the PC?

Sending X-Modem Files from the PC

Sending X-Modem files from the PC is almost the same as setting it up for serial communications, that is, you need a program capable of X-Modem file transfer. Luckily, there many of them available, but once again I suggest that you use two of my favorites Absolute Telnet and ZOC which you can find online here:

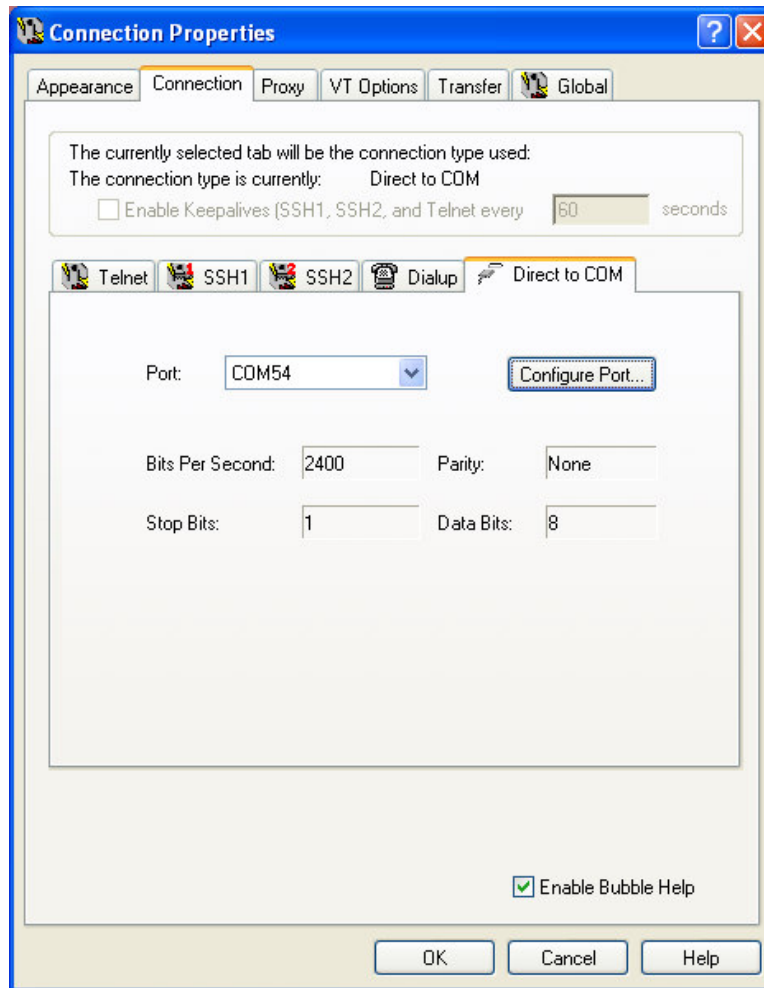
Zoc	http://www.emtec.com/zoc/
Absolute Telnet	http://www.celestialsoftware.net/telnet/

Copies of them are also on the DVD-ROM here :

DVD-ROM:\ CHAM_AVR \ TOOLS \ COMMUNICATIONS \ *.*

For this example, let's use Absolute Telnet. Similarly, you need to install the program, then create a communications profile for your USB serial COM port, but this time there is a difference, instead of 115,200 baud that we have been communicating at, we are going to slow things down to 2400 baud. This is to reduce the potential error rate to near zero.

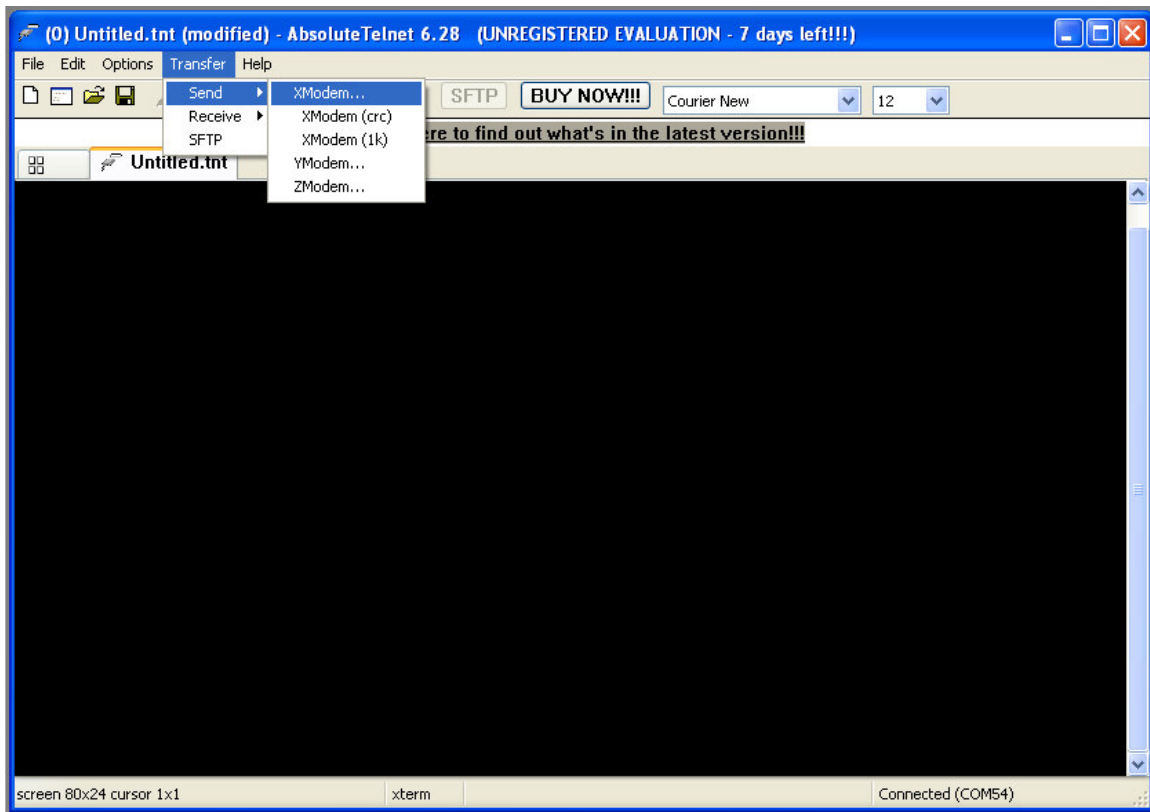
So your final setup should look something like the dialog shown in Figure 31.4 below.

Figure 31.4 – Setting up Absolute Telnet for X-Modem transfers.

The settings should be:

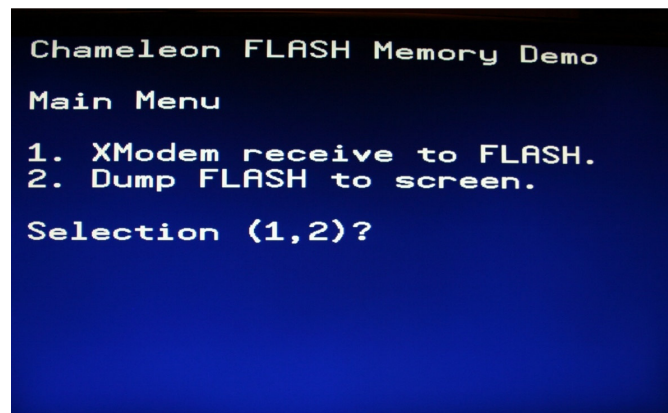
- Data Format: "N81", no parity, 8 data bits, 1 stop bit.
- Baud Rate: 2400.
- Handshaking: None.
- Terminal Mode: VT100 (doesn't really matter though since we aren't using the terminal).
- Screen Size: 80x24 or larger, 120x50 is a good choice.
- Local Echo: Off.
- Make sure that you have the serial selection switch in the **DOWN** position on the Chameleon (AVR mode).

Notice, the terminal mode VT100, etc. doesn't really matter, nor does echo really, since we aren't going to type anything. The PC is going to do the transfer itself in an automated fashion. Moving on, once you have Absolute Telnet setup, then you are going to make the connection to the Chameleon (once we have the FLASH demo loaded). To do this you simply click the lightning bolt or connection icon on the Absolute Telnet interface. Then once we want to initiate the X-Modem transfer you are going to access the main menu and access the **<Transfer → Send → XModem>** menu item (as shown in Figure 31.5 below), select a file and download to the Chameleon. But, we are getting ahead of ourselves right now. Just remember this process, and keep it in the back of your mind, lets switch gears to setting up the demo on the Chameleon and getting that ready to receive files.

Figure 31.5 – Sending an XModem file with Absolute Telnet.

Compiling and Running the Demo(s)

There are two demos in this group; the AVRStudio version and the Arduino version. They are nearly identical except that for the AVRStudio version has been ported to work on the Arduino with the usually porting tactics discussed previously in the manual. The main file that we introduce to access the FLASH API is **CHAM_AVR_FLASH_DRV_V010.C|H**. Once you compile and FLASH the Chameleon with the demo, it displays a menu on both the NTSC and VGA displays, this is shown in Figure 31.6 below.

Figure 31.6 – The FLASH demo main menu.

Referring to Figure 31.6 , there are two menu options:

1. Xmodem receive to FLASH.
2. Dump FLASH to screen.

You control the menu with the local keyboard plugged into the Chameleon's PS/2 port.

Option 1, starts a 3 second countdown and then starts the X-Modem download. Thus, the PC side better already have the XModem file transfer program ready and waiting. Moreover, the programs dumps the data to the screen in ASCII format, so I suggest that what your transfer its human readable. Also, make it short, maybe a few hundred bytes to a couple K. The AVR only has enough room for a 1K receive buffer in the X-Modem function, thus it only buffer 1K of the data and writes it to the FLASH memory everything else is shown on the screen as its received, but lost. Option 1 is shown in Figure 31.7 below.

Figure 31.7 – The FLASH demo receiving a file.

```
Download Complete.
11 Blocks.
File Size = 1408.

Opening FLASH memory...
Erasing first 4K of FLASH...
Writing 1408 bytes to FLASH...
Closing FLASH memory...

Main Menu

1. XModem receive to FLASH.
2. Dump FLASH to screen.

Selection (1,2)?
```

Of course, you can alter the XModem receive function to write files directly to the FLASH as they are received block by block, and you probably will need to.

Option 2, lists out the first 1K of the FLASH memory, so you can see what's in it. Point, being you want to boot the program, see what's in the FLASH it, then download something, see if it changes, reset, re-boot, try again. And example of Option 2 running is shown in Figure 31.8 below.

Figure 31.8 – The FLASH demo displaying a file.

```
1. XModem receive to FLASH.
2. Dump FLASH to screen.

Selection (1,2)?
Opening FLASH memory...
Reading data from FLASH...
The Road Not Taken by Robert Fro
st
.
.Two roads diverged in a yellow
wood,
.And sorry I could not travel bo
th
.And be one travel
```

Thus, the usual order of operations will be to boot the Chameleon with the FLASH demo, then you might hit **Option 2** to see what's in the FLASH's first 1K of memory. Then you might decide to overwrite it with something, so you go to your PC, set up the X-Modem download, start the file transfer and the PC program will wait...

Then you go back to the Chameleon's local keyboard and press **Option 1**, then you will see a 3 second countdown and the program will start downloading – as it downloads you will see the test stream across the screen.

Finally, I have created a few files for you to download with X-Modem that have short famous poems all located in the Source directory here:

**DVD-ROM:\ CHAM_AVR \ SOURCE **

"The Road Not Taken" by Robert Frost	- xmodem_test_file_road_not_taken.txt
"O Captain! My Captain!" by Walt Whitman	- xmodem_test_file_o_captain.txt
"All the World's a Stage" by William Shakespeare	- xmodem_test_file_all_the_worlds_stage.txt

They are better than using random .TXT files or data files and if you are over 21 years old and haven't read these at least once in your life, its worth doing so, they are very powerful and very true.

Compiling and Running the AVRStudio Version

Demo Version Description: FLASH memory demo that receives files via X-Modem protocol.

Main Source File: CHAM_AVR_FLASH_DEMO_01.c

Additional Files Required:

- CHAM_AVR_SYSTEM_V010.c|h
- CHAM_AVR_TWI_SPI_DRV_V010.c|h
- CHAM_AVR_UART_DRV_V010.c|h
- CHAM_AVR_NTSC_DRV_V010.c|h
- CHAM_AVR_VGA_DRV_V010.c|h
- CHAM_AVR_KEYBOARD_DRV_V010.c|h
- CHAM_AVR_FLASH_DRV_V010.c|h

General Requirements: NTSC or VGA display connected, local PS/2 keyboard connected, X-Modem download program running on PC, N81, 2400 baud.

Controls Required: Local PS/2 keyboard, simply use menu, also PC running X-Modem uploader.

Compiling and Running the Arduino Version

Demo Version Description: FLASH memory demo that receives files via X-Modem protocol.

Main Sketch Source File: CHAM_AVR_FLASH_DEMO

Additional Library Files Required (imported automatically by sketch)

- CHAM_AVR_SYSTEM_V010
- CHAM_AVR_TWI_SPI_DRV_V010
- CHAM_AVR_UART_DRV_V010
- CHAM_AVR_NTSC_DRV_V010
- CHAM_AVR_VGA_DRV_V010
- CHAM_AVR_KEYBOARD_DRV_V010
- CHAM_AVR_FLASH_DRV_V010

General Requirements: NTSC or VGA display connected, local PS/2 keyboard connected, X-Modem download program running on PC, N81, 2400 baud.

Controls Required: Local PS/2 keyboard, simply use menu, also PC running X-Modem uploader.

Technical Overview

The program is probably the most complex of all since it has to do so much. We need a NTSC/VGA menuing system that allows the user to enter menu options, then we needed the X-Modem function to receive files which a project in itself, and finally, the FLASH memory call to display and save the FLASH data. So a lot of parts, but with a little imagination, you can use this program as the basis of a MP3 or video player. You can download files into the FLASH with the X-Modem function, then write code that decompresses video or audio and presto!

Anyway, we have already seen the X-Modem receive file listing earlier in the overview and the complete program is rather long, too long to list. Therefore, I am just going to verbalize the FLASH aspect of it the program. Forgetting about the X-Modem support, accessing the FLASH memory is trivial with the API, all you need to do is call **Flash_Open()**, then you can read data with the **Flash_Read(...)** function. When you're done, call **Flash_Close()** and that's it! If you want to write to the flash memory, then in addition to the **Flash_Open()** call you must make sure that the blocks or sectors you want to write to have been erased first, once they are erased you can write to them. The erase function is **Flash_Erase(...)** and

as long as you have done that once before writing (or know that the FLASH is already erased) then you call **Flash_Write(...)** and write one or more bytes anywhere in the FLASH, once again, call **Flash_Close()** when you're done.

TIP

There is also a "chip erase" function supported by the FLASH memory that erases the whole chip. We removed this from the API since we were having trouble with it, the datasheets are always a little buggy, and might have something incorrect, so check it out – It's a simple command, but first you need to unprotect the chip and of course write enable then erase. It worked on some other versions of the FLASH chip, but these version it doesn't seem reliable. It WORKS – trust me, but there are some missing details in the data sheet possibly, a step that we are omitting with these new chips, so we deleted the function this time around.

Summary

In this chapter, you not only learned about the cool 1 MB Flash memory the Chameleon has, but got a taste of file transfer protocols. Hopefully, you can take our 30 minute X-Modem file transfer function and spend a good day on it and really make it robust!

32.1 Native Mode / Bootloader mode

Currently, there are two ways to use the Chameleon AVR:

- **Bootloader Mode:** with the Arduino bootloader pre-loaded into FLASH memory.
- **Native Mode:** with AVRStudio generating a complete FLASH image.

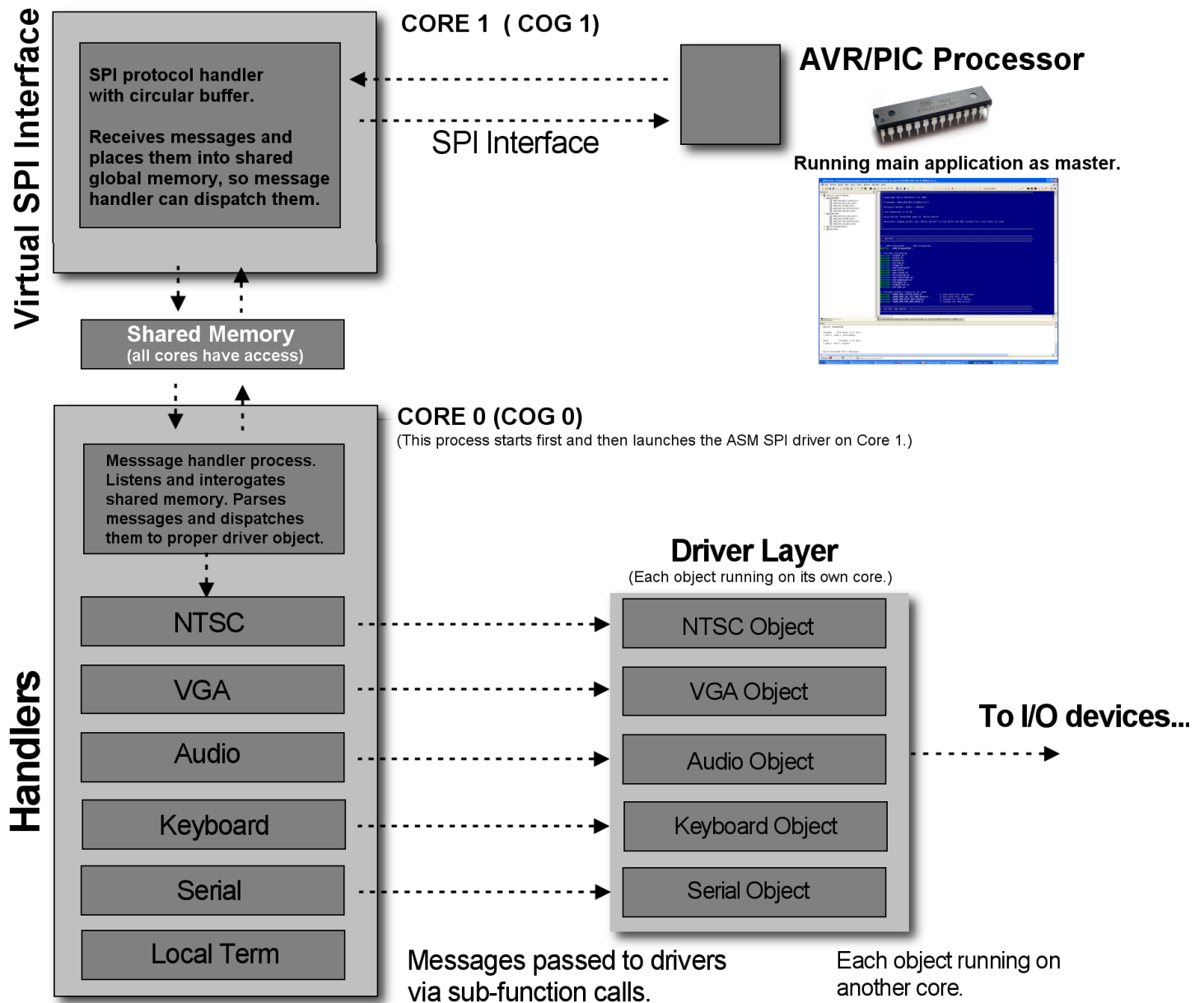
The bootloader mode is convenient since the Arduino tool can download to the Chameleon (or Arduino) with nothing more than a serial port. The native mode is nice since you have full control and you can write large complex programs that take up all the space of the AVR, download them into the AVR and not worry about the bootloader section getting corrupted, etc.

However, you do not have to use the Arduino bootloader alone... There are a number of bootloaders, in fact, the Arduino bootloader is nothing more than the Atmel standard application note bootloader slightly modified. Thus, you can make your own bootloader, use Atmel's, and so forth along with your own copy of the GNU GCC AVR tools and then program on Linux, Windows, Mac, or any other operating system the GNU GCC tools are available on. The bottom line is that all the bootloader does for us, is allow a "**serial protocol**" to communicate to the AVR chip and FLASH its memory with a program. This saves you the \$29-39 of a AVR MKII ISP programmer, that's about it.

33.1 Developing Your Own Propeller Drivers

Well, the last burning question in your mind should be, "how do I add more functions to the Propeller driver?". This is a multistep process, but very easy. As an example, we are going to add a couple messages that allow the client/master application running on the AVR to control the status LED on the Chameleon (next to the Propeller Local 8-bit port). Before, we begin, let's take a quick look at the flow of messages from the AVR to the Propeller once again to review all the components, take a look at Figure 33.1 below.

Figure 33.1 – The flow of messages from the AVR (or PIC) to the Propeller.



Referring to the figure, the client/master processor issues commands over the SPI interface which the master control program running on the Propeller chip, processes in a large message dispatching loop to the proper driver object. Thus, to add messages or functionality we have to consider the following things:

1. Will the Propeller driver need another object or driver? If so, we need to add this and then add code to the main message dispatcher loop that passes messages from the client to the new Propeller object.
2. If we add messages to the system, we need to make sure they are unique and don't overlap any other messages. Its good practice to not use message ids we have already used since you want to be as compatible as possible. Both the Propeller driver `cham_default2_drv_112.spin` and `cham_twi_spi_drv_v010.h` contain the master message id lists (copies) and the various library API files have functions that wrap the use of the messages in higher level functions.
3. There are three classes of messages we can add;
 - **Class 1:** (Same object more functionality). There is already a driver for the messages, the driver supports some extra functionality that we haven't exposed yet and wish to. The sound driver for example falls under this category. I barely exposed its true power, so you might want to add more messages. In this case, you have to add the messages to the .H file as well as the SPIN file. In addition to this you need to add more cases to process and dispatch the messages in the SPIN driver and then on the AVR C/C++ side you might want to add more API functions.
 - **Class 2:** (New object new functionality). The second case is when you want to add a completely new driver object and run it on another processor on the Propeller. In this case, you will load the object in at the top of the main driver, start it up, and then add messages along with message handlers to the main message dispatcher loop. This is the most complex case.
 - **Class 3:** (Simple messages that can be processed in place). Finally, the last case is the easiest and the one we will illustrate. This is the case, where we want to add some functionality to the main Propeller driver, but we can run that functionality right on the main processor that is running the message loop. Examples of this are the Propeller local port driver. Messages run right on the main SPIN interpreter, there is no need to add another processor.

33.1.1 Adding SPIN Driver Support for the Status LED

As we designed the Chameleon we added a status LED to it. The idea of this LED was for the Propeller to use it to indicate status, state, etc. For example, when the Propeller boots our driver it blinks 3 times to indicate that everything is good to go, driver is ready to receive messages from the client/master. However, after the we were all done designing everything and the drivers were written. We left out messages to control the LED from the client/master, but I thought, hmmm, I will hold off on this and later us it as the perfect example of adding some messages to do something useful.

Thus, this example was born of that motivation. With that in mind, this is a good exercise since at first you will want to make small changes to the master driver, but not re-write it. Anyway, so the first thing you need to do is decide what new functionality you need. In this case, all I needed was to control the LED, turn it on and off. So, the first thing you need to do is copy the Propeller driver and re-name it something else. In this case, I made a copy of:

`cham_default2_drv_112.spin`

and renamed it to:

`cham_default2_drv_112_modified.spin`

This way if I mess something up, I can go back to my original. Then I reviewed the messages at the top of the code and found where new messages can go and added two new commands 100 and 101, here's a snippet of what the constants looked like before and after:

Before

```
.
.
.
REG_CMD_WRITE      = 56 ' performs a 32-bit write to the addressed register [0..F] from the output register buffer
REG_CMD_READ       = 57 ' performs a 32-bit read from the addressed register [0..F] and stores in the input register buffer
REG_CMD_WRITE_BYTE = 58 ' write byte 0..3 of output register g_reg_out_buffer.byte[ 0..3 ]
```

```

REG_CMD_READ_BYTE      = 59 ' read byte 0..3 of input register g_reg_in_buffer.byte[ 0..3 ]

' system commands
SYS_RESET              = 64 ' resets the prop

' //////////////////////////////////////
' this range of commands for future expansion...
' //////////////////////////////////////

' advanced GFX commands for GFX tile engine
GPU_GFX_BASE_ID        = 192 ' starting id for GFX commands to keep them away from normal command set
GPU_GFX_NUM_COMMANDS   = 37  ' number of GFX commands

GPU_GFX_SUBFUNC_STATUS_R = (0+GPU_GFX_BASE_ID) ' Reads the status of the GPU, writes the GPU Sub-Function register and issues
' a high level command like copy, fill, etc.
GPU_GFX_SUBFUNC_STATUS_W = (1+GPU_GFX_BASE_ID) ' Writes status of the GPU, writes the GPU Sub-Function register and issues
' a high level command like copy, fill, etc.
.
.
.

```

After Adding New Messages

```

.
.
.
REG_CMD_WRITE          = 56 ' performs a 32-bit write to the addressed register [0..F] from the output register buffer
REG_CMD_READ           = 57 ' performs a 32-bit read from the addressed register [0..F] and stores in the input register buffer

REG_CMD_WRITE_BYTE     = 58 ' write byte 0..3 of output register g_reg_out_buffer.byte[ 0..3 ]
REG_CMD_READ_BYTE      = 59 ' read byte 0..3 of input register g_reg_in_buffer.byte[ 0..3 ]

' system commands
SYS_RESET              = 64 ' resets the prop

' //////////////////////////////////////
' this range of commands for future expansion...
' //////////////////////////////////////

' added status LED commands added for example in manual "Developing Your Own Propeller Drivers".
STATUS_LED_ON          = 100
STATUS_LED_OFF         = 101

' advanced GFX commands for GFX tile engine
GPU_GFX_BASE_ID        = 192 ' starting id for GFX commands to keep them away from normal command set
GPU_GFX_NUM_COMMANDS   = 37  ' number of GFX commands

GPU_GFX_SUBFUNC_STATUS_R = (0+GPU_GFX_BASE_ID) ' Reads the status of the GPU, writes the GPU Sub-Function register and issues
' a high level command like copy, fill, etc.
GPU_GFX_SUBFUNC_STATUS_W = (1+GPU_GFX_BASE_ID) ' Writes status of the GPU, writes the GPU Sub-Function register and issues
' a high level command like copy, fill, etc.
.
.
.

```

I have highlighted the two new message id's. That's all there is to it. The selection of 100,101 are a bit random, I could have used 65,66, or I could have used a single id and then used one of the data bytes to control on/off. But, I thought this would be a little more illustrative.

Now that we have the message ids, we need to process the messages and do something. A bit of investigation of the schematic and the boot code shows that I/O P25 is connected to the status LED. In fact, the boot code that blinks the LED 3 times during boot looks like this:

```

' blink the status LED 3 times to show board is "alive"
DIRA[25] := 1 ' set to output
OUTA[25] := 0

repeat 6
  OUTA[25] := !OUTA[25]
  repeat 25_000

```

Therefore, this code already sets up the I/O direction of the port bit, all we need to do is add the two messages that set the port bit LOW and HIGH. Reviewing the main message dispatcher loop, we simply find the last message processed and add a couple more messages. Here's that section before and after again:

Before

```

.
.
.
REG_CMD_WRITE_BYTE: ' write byte 0..3 of output register g_reg_out_buffer.byte[ 0..3 ]
' byte address to write 0..3 is in g_data while data to write is in g_data2
g_reg_out_buffer.byte[ g_data ] := g_data2

' read command from byte registers of 32-bit buffer register

REG_CMD_READ_BYTE: ' read byte 0..3 of input register g_reg_in_buffer.byte[ 0..3 ]
' this data is then placed into spi buffer for transport back to client
' byte address to read 0..3 is in g_data

```

```
g_spi_result := g_reg_in_buffer.byte[ g_data ]
```

```

SYS_RESET:      ' resets the prop
reboot

' end case commands

' set result and set dispatcher to idle
spi.finishcmd(g_spi_result)

' // end main loop
' ///////////////////////////////////////////////////////////////////
.
.
.
```

After Adding New Message Handlers

```

.
.
.

REG_CMD_WRITE_BYTE: ' write byte 0..3 of output register g_reg_out_buffer.byte[ 0..3 ]
' byte address to write 0..3 is in g_data while data to write is in g_data2
g_reg_out_buffer.byte[ g_data ] := g_data2

' read command from byte registers of 32-bit buffer register

REG_CMD_READ_BYTE: ' read byte 0..3 of input register g_reg_in_buffer.byte[ 0..3 ]
' this data is then placed into spi buffer for transport back to client
' byte address to read 0..3 is in g_data
g_spi_result := g_reg_in_buffer.byte[ g_data ]
```

```

SYS_RESET:      ' resets the prop
reboot
```

```
' // Propeller status LED commands book example on how to modify the driver
```

```
STATUS_LED_ON:
OUTA[25] := 1
```

```
STATUS_LED_OFF:
OUTA[25] := 0
```

```
' // end added code for status LED control
```

```

' end case commands

' set result and set dispatcher to idle
spi.finishcmd(g_spi_result)

' // end main loop
' ///////////////////////////////////////////////////////////////////
.
.
.
```

That's it! Save the file as **cham_default2_drv_112_modified.spin** and we are done with this portion. In fact, you can flash it down to the Propeller and every single program will still work the same. We have added messages and features, but we have **NOT** made anything incompatible or re-used message ids for something else. This is very important to keep in mind, so you do not damage a driver and make it incompatible with other software.

31.1.2 Adding AVR Support at the Client/Master Side

At this point, we have new messages added to the Propeller driver along with the message handler code. If you haven't done so, compile and download the new driver to the Propeller chip on the Chameleon, whatever application you have running on the Chameleon should work exactly the same.

Now, what we need is to add support on the client/master side or write a program that sends the messages to the Propeller. Rather than go into the SPI driver code header, I decided to just write a stand alone program that blinks the LED. I started with the port blinking demo and then modified it to directly send the new messages. Here's the entire program (AVRStudio straight C version):

```

/////////////////////////////////////////////////////////////////
// INCLUDES ///////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////

// __AVR_ATmega328P__ , __AVR_ATmega168__
```

```

#define __AVR_ATmega328P__

// include everything
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <avr/eeprom.h>
#include <avr/io.h>
#include <avr/sleep.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>
#include <inttypes.h>
#include <compat/twi.h>
#include <avr/wdt.h>

// include headers required for demo
#include "CHAM_AVR_SYSTEM_V010.h" // you need this one always
#include "CHAM_AVR_TWI_SPI_DRV_V010.h" // you need this always
#include "CHAM_AVR_NTSC_DRV_V010.h" // needed for NTSC driver

//////////////////////////////////////
// DEFINES AND MACROS ////////////////////////////////////////
//////////////////////////////////////

// define CPU frequency in MHZ here if not defined in Makefile or other includes, compiler will throw a warning,
// ignore
#ifndef F_CPU
#define F_CPU 16000000UL // 28636360UL, 14318180UL, 21477270 UL
#endif

// new commands for status LED control
#define STATUS_LED_ON 100
#define STATUS_LED_OFF 101

//////////////////////////////////////
// MAIN ////////////////////////////////////////
//////////////////////////////////////

int main(void)
{
    // initialize and set SPI rate
    SPI_Init( SPI_DEFAULT_RATE );

    // give Prop a moment before sending it commands, the boot process is about a second
    // if you like you can speed it up, by removing the Prop driver's "LED blink" sequence in the driver
    // or speed it up, but commands that are sent before the Prop driver is done booting will be ignored.
    // therefore, if you have a lot of set up work to do, then you don't require a delay, but if you jump right
    // into commands, then you need a good 1.5 - 2 second delay
    _delay_ms( 2500 );

    // clear screens
    NTSC_ClearScreen();
    NTSC_Color(0);

    // enter infinite loop...
    while(1)
    {
        // print on NTSC terminal screen
        NTSC_Term_Print(" Status LED ON");
        NTSC_Term_Char( 0x0D );

        // send command to turn on LED
        SPI_Prop_Send_Cmd(STATUS_LED_ON, 0,0 );

        // slow things down a bit, so we can read the text!
        _delay_ms(250);

        NTSC_Term_Print(" Status LED OFF");
        NTSC_Term_Char( 0x0D );

        // send command to turn off LED
        SPI_Prop_Send_Cmd(STATUS_LED_OFF, 0,0 );

        // slow things down a bit, so we can read the text!
        _delay_ms(250);

    } // end while
} // end main

```

The Arduino version simply has **loop()** instead of **main()** and the extra setup function, as well as the different header file syntax. The name of the AVRStudio version is **CHAM_AVR_STATUS_LED_DEMO_01.c** and is located on the DVD here:

DVD-ROM:\ CHAM_AVR \ SOURCE \ CHAM_AVR_STATUS_LED_DEMO_01.c

There is also the Arduino version "sketch" which is located on the DVD here:

DVD-ROM:\ CHAM_AVR \ TOOLS \ ARDUINO \ SKETCHES \ CHAM_AVR_STATUS_LED_DEMO

Which you should have copied all these files to your hard drive already. Compiling and building these programs should be old hat to you now, so I won't waste time with that. Let's just look at the code. Referring back to the source listing above, there are 2 things that make this program really

First, the addition of the message ids, show here:

```
// new commands for status LED control
#define STATUS_LED_ON 100
#define STATUS_LED_OFF 101
```

And secondly, the actual messages to the Propeller to turn the LED on and off (the on message for example):

```
// send command to turn on LED
SPI_Prop_Send_Cmd(STATUS_LED_ON, 0,0 );
```

And that's it! It just works – If you compile and run this program you will see the status LED blink on and off as well as the NTSC monitor display messages.

As you can see, its very simple to add new functionality to the driver and if you are careful and make sure not to re-use message ids then you can deploy your new driver for other Chameleon users (both AVR and PIC) and they can use the driver without change in their Chameleons and all their old programs **will** still work, but you **new** functionality will be at their finger tips.

34.1 Advanced Concepts and Ideas

There is so much you can do with the Chameleon, I don't even know where to begin. But, some of the things you might want to try are:

- Using the Propeller as the Master and writing a driver on the AVR chip, so the Propeller can use its resources, peripherals, etc.
- The Propeller has a local 8-bit port, you can use this for anything. One idea that is really easy to implement is adding a micro SD card adapter to it. Sparkfun sells a little adapter that is mounted on a right angle header, that will plug right into the 8-bit Propeller port. You can then get a SD card object, add some messages to the master control program driver, and presto you have a FLASH hard drive you can access from the AVR chip.
- Networking the Chameleons together. The Chameleons are so small and powerful, a cool idea would be to network 4-8 of them together each generating video, VGA, but being controlled by a "**super-master**" unit to do something in unison.
- Creating a Arduino Shield adapter board. The Chameleon (both AVR and PIC) have similar I/O headers to the Arduino, but due to physical constraints they are different. A cool piece of add on hardware would be a daughter board that mounts on top of the Chameleon to make it 100% I/O compatible with Arduino shields.

35.1 Demo Coder Applications, Games, and Languages

When developing any product we always like to give a select few master programmers the hardware early to see what they come up with. Not only is this a good way to find potential bugs in the hardware, but its interesting to see what some of the best programmers in the world can do with it. For this release of the Chameleons we have a couple applications that will be very useful to you.

35.1.1 Chameleon BASIC by David Betz

Figure 35.1 – A BASIC “Life” program running on the Chameleon.



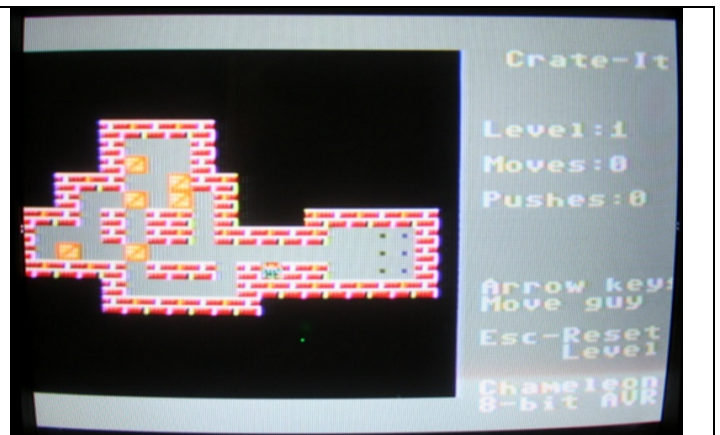
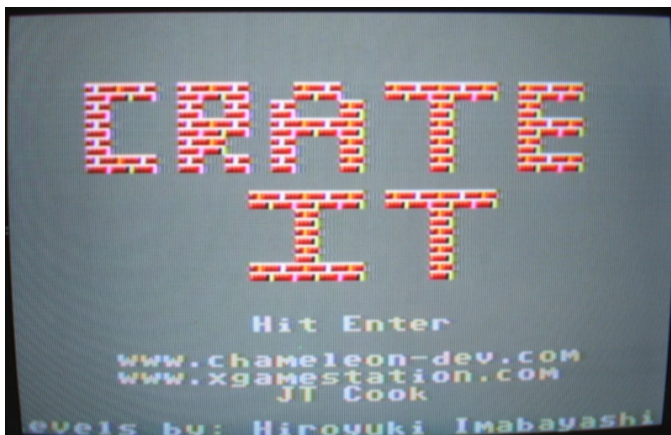
The Chameleon BASIC is a complete compiled BASIC that runs on both the Chameleon AVR and PIC platforms (Figure 35.1 shows an example of a “Life” cellular automata running written in BASIC. You write your programs on the PC with an editor then download to the Chameleon. The Chameleon runs a VM (virtual machine) that executes the compiled byte-codes. You can find the latest copy of BASIC on the DVD here:

DVD-ROM:\ CHAM_AVR \ SOURCE \ BASIC

Simply locate the newest directory version and everything you need to get BASIC up and running is in there including documentation, the compiler, examples, etc.

35.1.2 Crate-It! by JT Cook

Figure 35.2 – Crate-it! running on the Chameleon AVR.



Crate-It! (shown in Figure 35.2) is used for the Chameleons as the Quick Start demo that is pre-loaded on each unit. To compile it, you follow the same process as you would for all the other examples. There are two versions; one for the native AVRStudio mode and a Arduino based Sketch. You can find the files for the AVRStudio version on the DVD here:

DVD-ROM:\ CHAM_AVR \ SOURCE \ DEMO_CODERS \ CRATE_IT \ *.*

The Arduino sketch version is located on the DVD here:

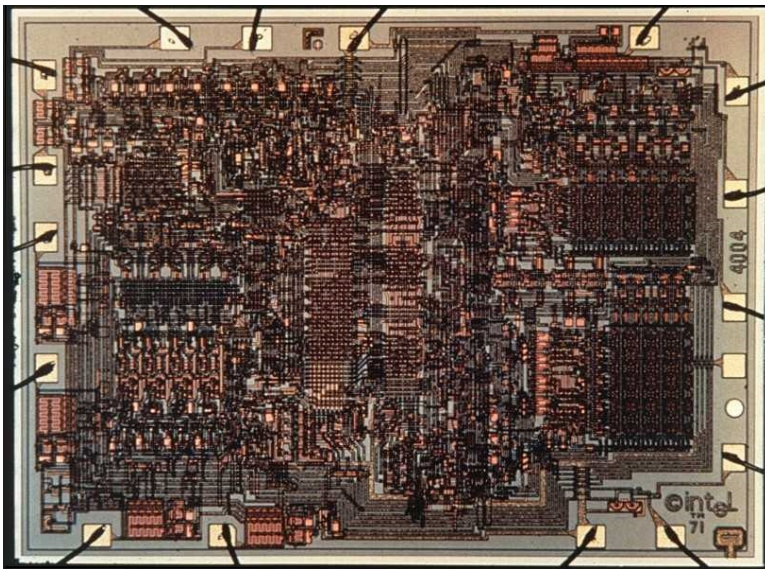
DVD-ROM:\ CHAM_AVR \ TOOLS \ ARDUINO \ CHAM_AVR_CRATE_IT

You should have already copied this on your hard drive during the installation process earlier in the manual.

Any other newer demos will be located in the \DEMO_CODER directory, so take a look there in case we don't update this manual to match. Similarly, always check the product web pages on **www.xgamestation.com** as well as **www.chameleon-dev.com**.

Epilog - From Intel 4004 to the Multiprocessing/Multicore Chameleon

Well, you made it! If you got this far then you should be a quite comfortable with the Chameleon AVR 8-Bit and see the true power of multiprocessing and the synergy between the Atmel AVR and the Parallax Propeller chip. Hopefully, you can use the Chameleons in many of your projects now and in the future. I am really excited about these little machines and I haven't had this much fun programming something in a long time – they just work and allow me to solve problems very quickly and get software done for real world applications.



It always amazes me to think about how far we have come in such a short time. The Chameleons push 180-200MIPs depending on the version; AVR/PIC. Literally, 100-200x more powerful than the 8-bit computers of the 1980's, but yet are the size of credit cards.

The first "official" processor was the 4-Bit Intel 4004 (shown to the left) created in 1971. It contained about 2300 transistors and executed instructions at a rate of 92,000 per second. Comparing this to our simple Chameleon AVR that runs about 180 million instructions per second, that's about 2000 times more powerful. But, if we look at cutting edge GPU cores that perform computation in the trillions per second then that's 10 million times more powerful! Then go head and parallel up 10, 100, 1000 of those processing cores and the amount of computation is astonishing!!!

Bottom line, our little brains are easily going to be matched and surpassed by computers **VERY** soon. So, cherish these last days while humans are still the dominant species on the planet! Good luck and please help us build the Chameleon community, you can visit our forums at:

<http://www.xgamestation.com/phpbb/index.php>

and

<http://www.chameleon-dev.com>

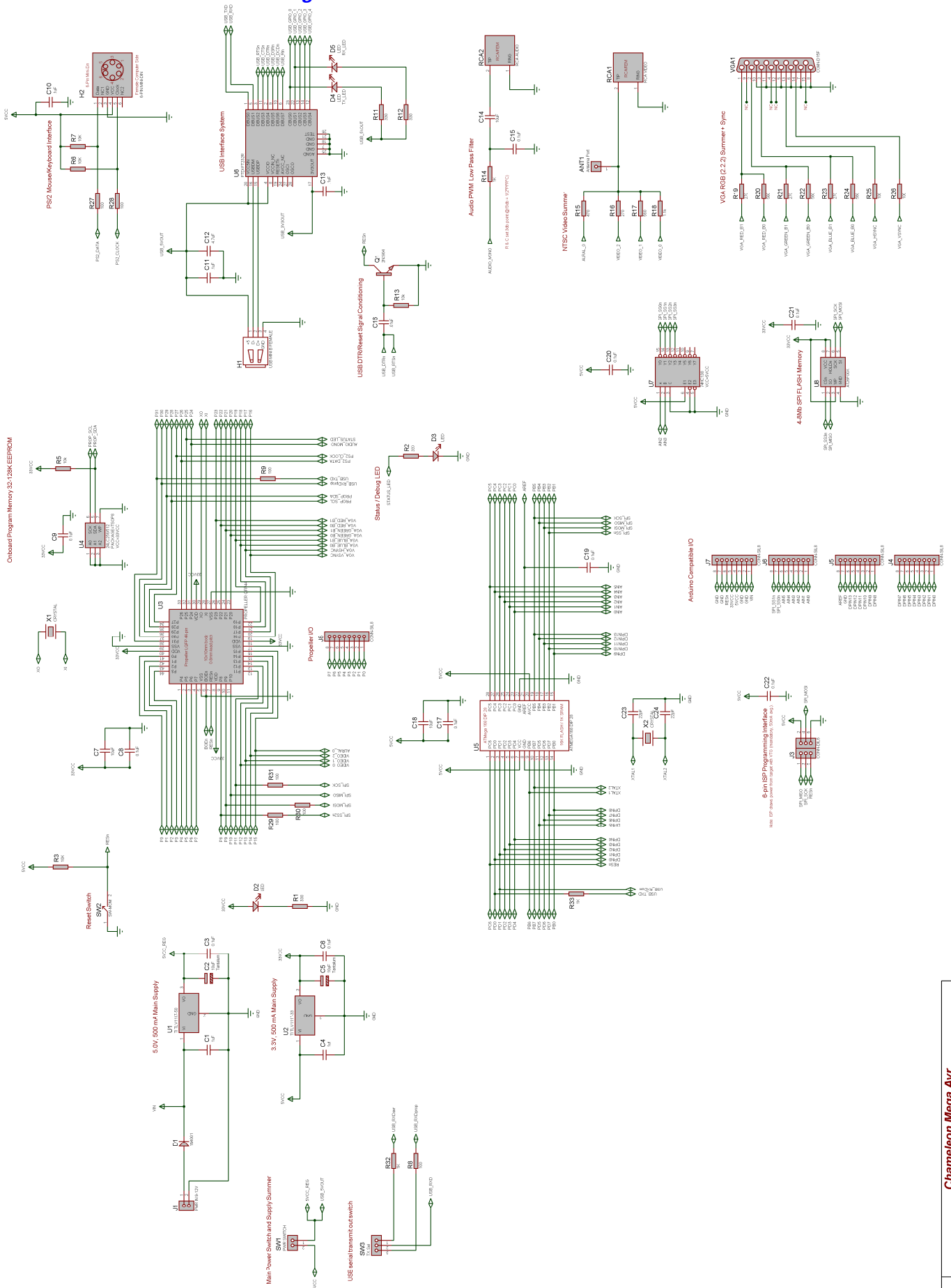
Discuss and show off your demos, applications, drivers, and games with other Chameleon programmers. Also, if you have any questions, comments, or remarks please email us at **support@nurve.net**.

Part III – Appendices

Appendix A: Schematics

This appendix contains the master schematic of the Chameleon AVR 8-Bit and all its sub-systems.

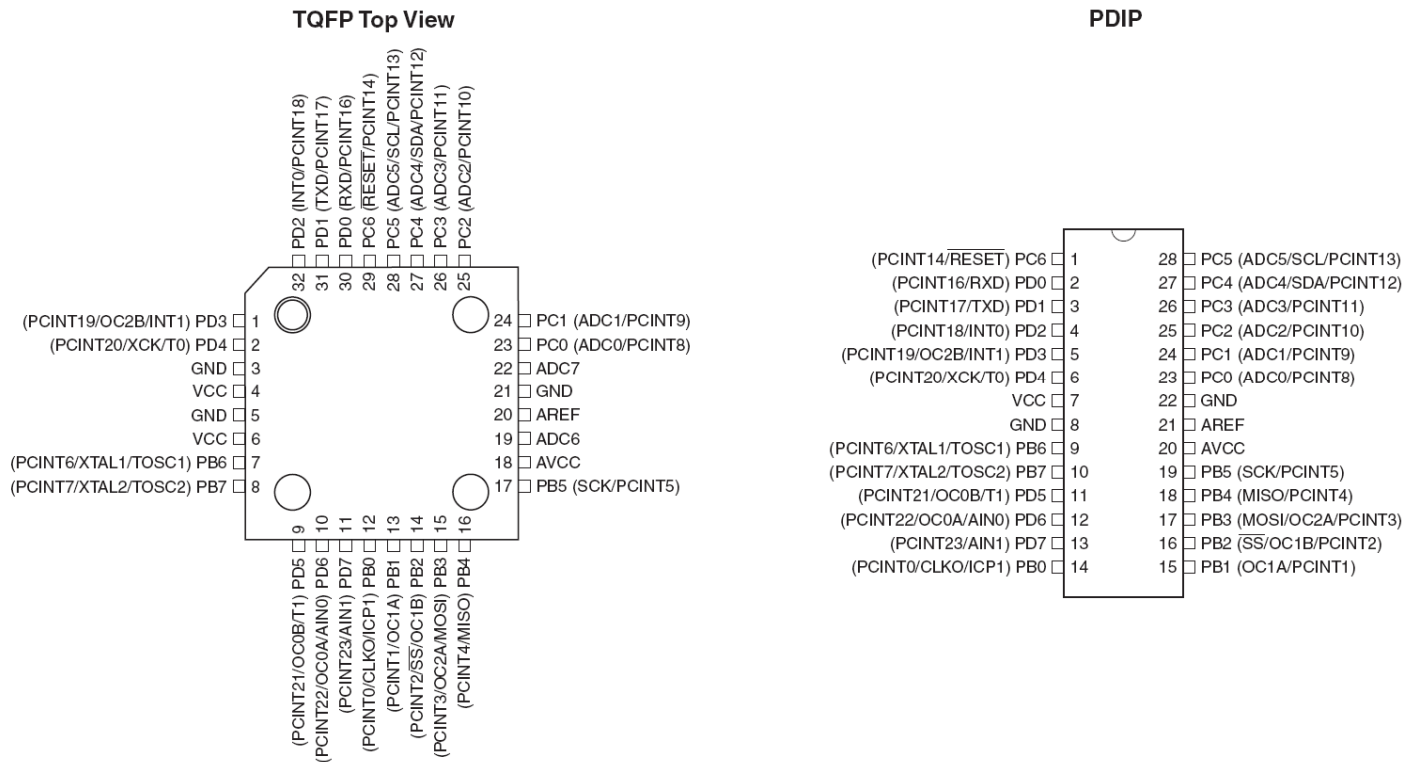
Figure A.1 –Chameleon AVR 8-Bit schematic.



Appendix B: Atmel AVR 328P Pinout

These images show the Atmel AVR644P DIP and QFP packages.

Figure B.1 – Atmel AVR 328P pinouts.



Appendix D - Using the AVR in "Stand-Alone" Mode.

The nice thing about the Chameleon is that its two systems in one. You are free to use the AVR as you wish and not worry about the Propeller chip. However, whatever driver that is on the Propeller will run and function independently even though, you aren't sending it commands over the SPI channel. Thus, if you truly want to turn off the Propeller, you might want to load a NULL program that does absolutely nothing such as:

```
CON
    ' These settings are for 5 MHZ XTALS
    _clkmode = xtal1 + pll16x      ' enable external clock and pll times 16
    _xinfreq = 5_000_000          ' set frequency to 5 MHZ

PUB Main
    repeat ' forever loop
```

This will start up a single processor, and just sit and do nothing. Of course, without the Propeller, you have no access to the physical peripheral headers such as the A/V headers, VGA, or the PS/2 header. But, you still have access to the USB serial port as well as the FLASH memory. And of course, all the headers connected to the AVR around the board you still have access to. Just make sure to keep the **serial select** switch in the **DOWN** position, so the AVR has access to the USB serial port.

Appendix E - Using the Propeller in "Stand-Alone" Mode.

The Propeller in stand alone mode is much more interesting than the AVR. The Propeller in stand alone mode allows you to more or less run anything you would run on a HYDRA or Propeller Development kit still. You have the VGA connector, PS/2, A/V connectors as well as an 8-bit port for expansion. You might have to do some "porting" of apps since the HYDRA and Propeller development kits use slightly different pin I/O connections for various devices, but those are literally 1-2 line changes in your code.

The only thing you lose in Propeller stand alone mode is the use of the onboard FLASH since the Propeller can't get to it, but the EEPROM for the Propeller is 64K, 2x the amount needed for the boot image, so that's a consolation prize in the design, if you need more assets. Just make sure to keep the **serial select** switch in the **UP** position, so the Propeller has access to the USB serial port.

Appendix F – Porting HYDRA and Parallax Development Board Applications to the Chameleon

There really isn't much to say here other than porting from the HYDRA or Propeller Development board is usually a few lines of code that needs to be changed. The things to watch out for are:

- **Make sure that you are using the same clocking and XTAL** – If the HYDRA/Dev Board application you are trying to port uses a 10MHz XTAL and the 8x mode, then make sure you change it to 5MHz and 16X mode in the source code.
- **I/O port assignments** – The Chameleon's Propeller interfaces are almost the same as the HYDRA and Propeller Development boards, but there are differences for example in what pin bank is used for the NTSC video or what pin for audio, etc. so when porting something, make sure you look at the pinouts of the source, and change them to match the Chameleon's.
- **Peripheral devices** – The Chameleon does not have a NES game port for example, so any game or program on the HYDRA that you want to port, you will need to delete that peripheral code, or swap it for something else. Of course, you can always hack a connector to the Propeller Local port and connect the controller there and switch the pins in the source code. Also, the Chameleon's uses a 2 line version of the keyboard and mouse drivers, the HYDRA and other Propeller development boards might use a 4 signal version, so watch out for that. And lastly, the Chameleon has a single sound PWM sound pin that routes to the RCA header, some Propeller dev boards use stereo sound, so in those cases you will want to route one of those pins to the Chameleon pin and leave the other dead or maybe out to the local 8-bit port.

Other than those 3 areas, I find that it takes a few mins to port anything from the HYDRA or a Propeller Development board to the Chameleon and use it in Stand-Alone mode, so pretty cool!

Appendix G - Running on the Mac and Linux.

AVRStudio is currently only available for the Windows operating system. Although, many people love their Macs and Linux, 90% of engineering development is still on Windows machines and will likely stay there for a time to come. However, that said there are 3rd party tools that run on Mac and Linux for the AVR processor. Companies such as IAR, Green Hills, and others have complete tool chains that run on Mac and Linux.

That said, the funny thing is that the AVR compiler used by AVRStudio is a Linux based GNU GCC compiler! But, Atmel hasn't had a reason to port it to Mac and Linux. However, you can develop 100% on a Mac or Linux machine and then use programs like AVRdude to download to the AVR using ISP.

But, the Arduino tool does all this for you, thus for Mac and Linux enthusiasts I suggest simply using the Chameleon in Arduino mode, installing the Mac/Linux version of the tool on your PC and then copying the Chameleon Libraries and Sketches just as you did (would have) for the Windows version. On the DVD in this location:

**DVD-ROM : \ CHAM_AVR \ TOOLS \ ARDUINO **

Are all the files you need to get started; there is a directory for the **Library** files that you need to add to the Mac/Linux installation of the Arduino tool as well as the **Sketches** for all the demos. Additionally, I have downloaded the Linux and Mac versions of the Arduino tool and put them in the respective directories **\Mac** and **\Linux** to save you downloading time.

The cool thing about using the Arduino tool is that the exact same Chameleon software for Arduino mode works on all three operating systems without change. You simply install the Arduino tool, make sure you have the bootloader on the AVR chip, and then use the tool, from your perspective the experience on Windows, Linux, and Mac should be identical.

Appendix H – Overclocking the AVR and Propeller

As with any TTL or CMOS device you can always overclock them. Typically, 10% overclocking will work 99% of the time, anything higher than 10% you have to take a few things into consideration such as power dissipation, memory response times, and overall timing of the chips. However, both the AVR and Propeller can be over clocked if you want to get more performance and are willing to potentially damage the chips.

H.1 Overclocking the Propeller Microcontroller

The Propeller uses an external clock or xtal that is then spun up by an internal PLL. The PLL has a number of power of 2 clock multiplier rates, but typically a 5-10MHz clock (xtal) is used and then the clock multiplier is set at 8x or 16. For example, the Default2 driver clock directives look like this:

```
CON
' These settings are for 5 MHZ XTALS
_clkmode = xtal1 + pll16x      ' enable external clock and pll times 16
_xinfreq = 5_000_000           ' set frequency to 5 MHZ
```

This instructs the compiler to generate an object with settings that assume a 5MHz input clock and then the processor scales this by 16X resulting in the nominal 80MHz that the Propeller chip needs to operate. The 80 MHz clock drives each processor clock and each processor executes one instruction per (4) clocks, thus 20 MIPS on average.

Now, if we want to go a little faster, we could use a 6MHz XTAL and then use the **pll16x** directive once again, this results in a clock rate of 96 MHz and 24 MIPS per processor. Depending on your Propeller chip, this might work as is; however, the transitions are so fast that the noise margins, etc. of the signals become small, and the capacitive loading too much for the 3.3V supply, this you might need to increase the power supply voltage to 3.6+ volts to get this speed to work. But, point is if you need a little bit more, you can increase you oscillator (xtal) a bit and then see if it works.

Also, the Propeller will get hotter as you overclock it, thus you might have to heat sink it.

H.2 Overclocking the AVR328P Microcontroller

The AVR is a little bit different than the Propeller chip, a little more forgiving, plus its already running at 5V. The AVR is spec'ed to max out at 20 MHz, but we are running a 16MHz xtal for Arduino compatibility. However, if you do not want to use Arduino mode, or the bootloader, then go ahead and pop in a 20 MHz parallel resonant mode Xtal and you get 4 MIPS for the taking. However, if you want more than you can do it as well. The max overclock of the AVRs is about 35 MHz before they simply fall apart and stop working. I find that 8xNTSC or 28.636636MHz is a nice compromise. I tested over 1000 units and they all can sustain this speed. The timings on some of the peripherals become tight for example, the serial I/O and SPI might error more often and the rate of bit errors with the FLASH memory increases, but 28.636 MHz is a solid overclocking rate.

The chip will get a little hotter, but not much, the interesting thing though is the chip doesn't stop malfunctioning due to overheating, but simply that certain parts of the chip can't go any faster. For example, the internal SRAM/FLASH has more limits than say the glue logic inside. So even though most of the chip might run at 40 MHz, it doesn't matter since the SRAM and FLASH won't read correctly.

In conclusion, the AVR will go 28.636 MHz no problem without any changes to the Vcc voltage, or heatsinking needed.

Appendix I: ASCII / Binary / Hex / Octal Universal Lookup Tables

ASCII – Hex Table

ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol
0	0	NUL	16	10	DLE	32	20	(space)	48	30	0
1	1	SOH	17	11	DC1	33	21	!	49	31	1
2	2	STX	18	12	DC2	34	22	"	50	32	2
3	3	ETX	19	13	DC3	35	23	#	51	33	3
4	4	EOT	20	14	DC4	36	24	\$	52	34	4
5	5	ENQ	21	15	NAK	37	25	%	53	35	5
6	6	ACK	22	16	SYN	38	26	&	54	36	6
7	7	BEL	23	17	ETB	39	27	'	55	37	7
8	8	BS	24	18	CAN	40	28	(	56	38	8
9	9	TAB	25	19	EM	41	29	)	57	39	9
10	A	LF	26	1A	SUB	42	2A	*	58	3A	:
11	B	VT	27	1B	ESC	43	2B	+	59	3B	;
12	C	FF	28	1C	FS	44	2C	,	60	3C	<
13	D	CR	29	1D	GS	45	2D	-	61	3D	=
14	E	SO	30	1E	RS	46	2E	.	62	3E	>
15	F	SI	31	1F	US	47	2F	/	63	3F	?

ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol
64	40	@	80	50	P	96	60	`	112	70	p
65	41	A	81	51	Q	97	61	a	113	71	q
66	42	B	82	52	R	98	62	b	114	72	r
67	43	C	83	53	S	99	63	c	115	73	s
68	44	D	84	54	T	100	64	d	116	74	t
69	45	E	85	55	U	101	65	e	117	75	u
70	46	F	86	56	V	102	66	f	118	76	v
71	47	G	87	57	W	103	67	g	119	77	w
72	48	H	88	58	X	104	68	h	120	78	x
73	49	I	89	59	Y	105	69	i	121	79	y
74	4A	J	90	5A	Z	106	6A	j	122	7A	z
75	4B	K	91	5B	[	107	6B	k	123	7B	{
76	4C	L	92	5C	\	108	6C	l	124	7C	
77	4D	M	93	5D	]	109	6D	m	125	7D	}
78	4E	N	94	5E	^	110	6E	n	126	7E	~
79	4F	O	95	5F	_	111	6F	o	127	7F	

Decimal – Hex – Octal – Binary Table

Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin
0	0	000	00000000	16	10	020	00010000	32	20	040	00100000	48	30	060	00110000
1	1	001	00000001	17	11	021	00010001	33	21	041	00100001	49	31	061	00110001
2	2	002	00000010	18	12	022	00010010	34	22	042	00100010	50	32	062	00110010
3	3	003	00000011	19	13	023	00010011	35	23	043	00100011	51	33	063	00110011
4	4	004	00000100	20	14	024	00010100	36	24	044	00100100	52	34	064	00110100
5	5	005	00000101	21	15	025	00010101	37	25	045	00100101	53	35	065	00110101
6	6	006	00000110	22	16	026	00010110	38	26	046	00100110	54	36	066	00110110
7	7	007	00000111	23	17	027	00010111	39	27	047	00100111	55	37	067	00110111
8	8	010	00001000	24	18	030	00011000	40	28	050	00101000	56	38	070	00111000
9	9	011	00001001	25	19	031	00011001	41	29	051	00101001	57	39	071	00111001
10	A	012	00001010	26	1A	032	00011010	42	2A	052	00101010	58	3A	072	00111010
11	B	013	00001011	27	1B	033	00011011	43	2B	053	00101011	59	3B	073	00111011
12	C	014	00001100	28	1C	034	00011100	44	2C	054	00101100	60	3C	074	00111100
13	D	015	00001101	29	1D	035	00011101	45	2D	055	00101101	61	3D	075	00111101
14	E	016	00001110	30	1E	036	00011110	46	2E	056	00101110	62	3E	076	00111110
15	F	017	00001111	31	1F	037	00011111	47	2F	057	00101111	63	3F	077	00111111

Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin
64	40	100	01000000	80	50	120	01010000	96	60	140	01100000	112	70	160	01110000
65	41	101	01000001	81	51	121	01010001	97	61	141	01100001	113	71	161	01110001
66	42	102	01000010	82	52	122	01010010	98	62	142	01100010	114	72	162	01110010
67	43	103	01000011	83	53	123	01010011	99	63	143	01100011	115	73	163	01110011
68	44	104	01000100	84	54	124	01010100	100	64	144	01100100	116	74	164	01110100
69	45	105	01000101	85	55	125	01010101	101	65	145	01100101	117	75	165	01110101
70	46	106	01000110	86	56	126	01010110	102	66	146	01100110	118	76	166	01110110
71	47	107	01000111	87	57	127	01010111	103	67	147	01100111	119	77	167	01110111

72	48	110	01001000	88	58	130	01011000	104	68	150	01101000	120	78	170	01111000
73	49	111	01001001	89	59	131	01011001	105	69	151	01101001	121	79	171	01111001
74	4A	112	01001010	90	5A	132	01011010	106	6A	152	01101010	122	7A	172	01111010
75	4B	113	01001011	91	5B	133	01011011	107	6B	153	01101011	123	7B	173	01111011
76	4C	114	01001100	92	5C	134	01011100	108	6C	154	01101100	124	7C	174	01111100
77	4D	115	01001101	93	5D	135	01011101	109	6D	155	01101101	125	7D	175	01111101
78	4E	116	01001110	94	5E	136	01011110	110	6E	156	01101110	126	7E	176	01111110
79	4F	117	01001111	95	5F	137	01011111	111	6F	157	01101111	127	7F	177	01111111

Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin
128	80	200	10000000	144	90	220	10010000	160	A0	240	10100000	176	B0	260	10110000
129	81	201	10000001	145	91	221	10010001	161	A1	241	10100001	177	B1	261	10110001
130	82	202	10000010	146	92	222	10010010	162	A2	242	10100010	178	B2	262	10110010
131	83	203	10000011	147	93	223	10010011	163	A3	243	10100011	179	B3	263	10110011
132	84	204	10000100	148	94	224	10010100	164	A4	244	10100100	180	B4	264	10110100
133	85	205	10000101	149	95	225	10010101	165	A5	245	10100101	181	B5	265	10110101
134	86	206	10000110	150	96	226	10010110	166	A6	246	10100110	182	B6	266	10110110
135	87	207	10000111	151	97	227	10010111	167	A7	247	10100111	183	B7	267	10110111
136	88	210	10001000	152	98	230	10011000	168	A8	250	10101000	184	B8	270	10111000
137	89	211	10001001	153	99	231	10011001	169	A9	251	10101001	185	B9	271	10111001
138	8A	212	10001010	154	9A	232	10011010	170	AA	252	10101010	186	BA	272	10111010
139	8B	213	10001011	155	9B	233	10011011	171	AB	253	10101011	187	BB	273	10111011
140	8C	214	10001100	156	9C	234	10011100	172	AC	254	10101100	188	BC	274	10111100
141	8D	215	10001101	157	9D	235	10011101	173	AD	255	10101101	189	BD	275	10111101
142	8E	216	10001110	158	9E	236	10011110	174	AE	256	10101110	190	BE	276	10111110
143	8F	217	10001111	159	9F	237	10011111	175	AF	257	10101111	191	BF	277	10111111

Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin
192	C0	300	11000000	208	D0	320	11010000	224	E0	340	11100000	240	F0	360	11110000
193	C1	301	11000001	209	D1	321	11010001	225	E1	341	11100001	241	F1	361	11110001
194	C2	302	11000010	210	D2	322	11010010	226	E2	342	11100010	242	F2	362	11110010
195	C3	303	11000011	211	D3	323	11010011	227	E3	343	11100011	243	F3	363	11110011
196	C4	304	11000100	212	D4	324	11010100	228	E4	344	11100100	244	F4	364	11110011
197	C5	305	11000101	213	D5	325	11010101	229	E5	345	11100101	245	F5	365	11110100
198	C6	306	11000110	214	D6	326	11010110	230	E6	346	11100110	246	F6	366	11110101
199	C7	307	11000111	215	D7	327	11010111	231	E7	347	11100111	247	F7	367	11110110
200	C8	310	11001000	216	D8	330	11011000	232	E8	350	11101000	248	F8	370	11110111
201	C9	311	11001001	217	D9	331	11011001	233	E9	351	11101001	249	F9	371	11111000
202	CA	312	11001010	218	DA	332	11011010	234	EA	352	11101010	250	FA	372	11111001
203	CB	313	11001011	219	DB	333	11011011	235	EB	353	11101011	251	FB	373	11111010
204	CC	314	11001100	220	DC	334	11011100	236	EC	354	11101100	252	FC	374	11111011
205	CD	315	11001101	221	DD	335	11011101	237	ED	355	11101101	253	FD	375	11111100
206	CE	316	11001110	222	DE	336	11011110	238	EE	356	11101110	254	FE	376	11111101
207	CF	317	11001111	223	DF	337	11011111	239	EF	357	11101111	255	FF	377	11111110
															11111111

Appendix J: ANSI Terminal Codes

This appendix lists some of the more popular **ANSI X3.64** terminal codes used to control graphically terminals and make really cool graphics on 8-bit computers! The majority of ANSI codes start off with the ANSI escape sequence which is the characters ESC (ASCII decimal 27 / hex 0x1B) and [(left bracket), in other words:

"ESC ["

Where "ESC" is actually the decimal value 27. This sequence is called **CSI** for **C**ontrol **S**equences **I**ntroducer (or Control Sequence Initiator). There is a single-character CSI value = 155 or 0x9B hexl. The "ESC [" two-character sequence is more often used than the single-character alternatives. Devices supporting only ASCII (7-bits), or which implement 8-bit code pages which use the 0x80–0x9F control character range for other purposes will recognize only the two-character sequence. Though some encodings use multiple bytes per character, in this topic all characters are single-byte. You can run a terminal program on the PC in ANSI mode and then send these commands to it from the Chameleon.

ANSI Codes (Most Popular)		
Code	Name	Effect
CSI n A	CUU	Moves the cursor n (default 1) cells in the given direction. If the cursor is already at the edge of the screen, this has no effect. CUU: Up; CUD: Down; CUF: Forward; CUB: Back;
CSI n B	CUD	
CSI n C	CUF	
CSI n D	CUB	
CSI n E	CNL	Moves cursor to beginning of the line n (default 1) lines down (next line).
CSI n F	CPL	Moves cursor to beginning of the line n (default 1) lines up (previous line).
CSI n G	CHA	Moves the cursor to column n.
CSI n ; m H	CUP	Moves the cursor to row n, column m. The values are 1-based, and default to 1 (top left corner) if omitted. A sequence such as CSI ;5H is a synonym for CSI 1;5H as well as CSI 17;H is the same as CSI 17H and CSI 17;1H
CSI n J	ED	Clears part of the screen. If n is zero (or missing), clear from cursor to end of screen. If n is one, clear from cursor to beginning of the screen. If n is two, clear entire screen (and moves cursor to upper left on MS-DOS ANSI.SYS).
CSI n K	EL	Erases part of the line. If n is zero (or missing), clear from cursor to the end of the line. If n is one, clear from cursor to beginning of the line. If n is two, clear entire line. Cursor position does not change.
CSI n S	SU	Scroll whole page up by n (default 1) lines. New lines are added at the bottom. (not ANSI.SYS)
CSI n T	SD	Scroll whole page down by n (default 1) lines. New lines are added at the top. (not ANSI.SYS)
CSI n ; m f	HVP	Moves the cursor to row n, column m. Both default to 1 if omitted. Same as CUP
CSI n [;k m	SGR	Sets SGR (Select Graphic Rendition) parameters. After CSI can be zero or more parameters separated with ;. With no parameters, CSI m is treated as CSI 0 m (reset / normal), which is typical of most of the ANSI codes.
CSI 6 n	DSR	Reports the cursor position to the application as (as though typed at the keyboard) ESC[n;mR, where n is the row and m is the column. (May not work on MS-DOS.)
CSI s	SCP	Saves the cursor position.
CSI u	RCP	Restores the cursor position.
CSI ?25l	DECTC EM	Hides the cursor.
CSI ?25h	DECTC EM	Shows the cursor.

Color Table for Select Graphics Rendition "SGR" Commands

Intensity	0	1	2	3	4	5	6	7	9
Normal	Black	Red	Green	Yellow[5]	Blue	Magenta	Cyan	White	reset
Bright	Black	Red	Green	Yellow	Blue	Magenta	Cyan	White	

SGR (Select Graphic Rendition) Parameters		
Code	Effect	Note
0	Reset / Normal	all attributes off
1	Intensity: Bold	
2	Intensity: Faint	not widely supported
3	Italic: on	not widely supported. Sometimes treated as inverse.
4	Underline: Single	not widely supported
5	Blink: Slow	less than 150 per minute
6	Blink: Rapid	MS-DOS ANSI.SYS; 150 per minute or more
7	Image: Negative	inverse or reverse; swap foreground and background
8	Conceal	not widely supported
21	Underline: Double	
22	Intensity: Normal	not bold and not faint
24	Underline: None	
25	Blink: off	
27	Image: Positive	
28	Reveal	conceal off
30–39	Set foreground color, normal intensity	3x, where x is from the color table above
40–49	Set background color, normal intensity	4x, where x is from the color table above

Examples

CSI 2 J This clears the screen and sets the cursor to (0,0) or (1,1) (upper left corner).
 CSI 32 m This makes text green.

NOTES

Mouser Electronics

Authorized Distributor

Click to View Pricing, Inventory, Delivery & Lifecycle Information:

[Nurve Networks:](#)

[Chameleon-AVR](#)