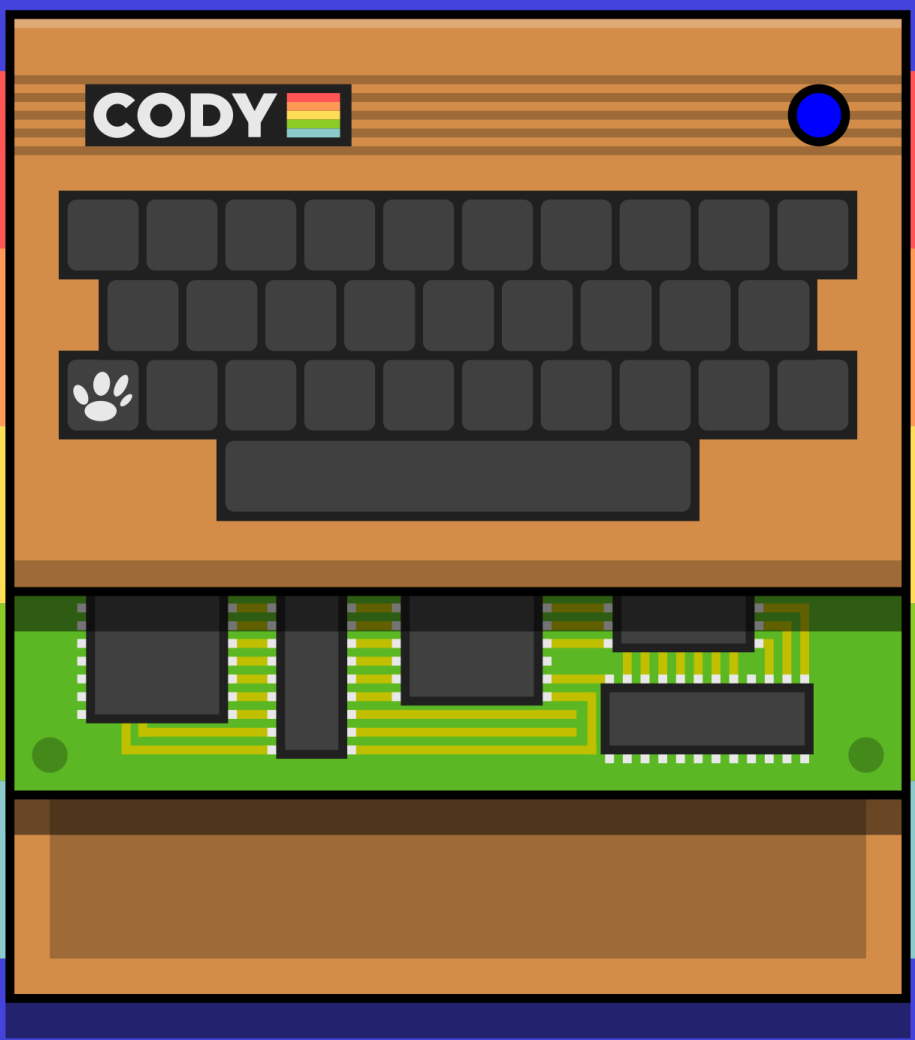
THE CODY COMPUTER BOOK



The Cody Computer Book

DRAFT

The Cody Computer Book

Frederick John Milens III

Photography by Frederick John Milens III and Ashanna Biliter.

Written in HTML5 and converted to PDF using Weasyprint. Typefaces used are Orkney, Anka Coder Narrow, and CMU Typewriter.

Artwork created with Inkscape and KiCad. Photographs for the book content were taken on a Nikon 1 S1.

For more information and sources/designs please visit www.codycomputer.org.

THIS IS A DRAFT COPY.

To Cody Biliter-Milens

2006-2020

Table of Contents

1

Introduction

INTRODUCTION

Welcome to *The Cody Computer Book*, a guide to building and programming your own 8-bit computer. The computer you'll build is inspired by the popular home computers of the 1980s—particularly the Commodore series—though it is not a direct clone of or compatible with any of them. Rather, it tries to be a somewhat-faithful modern take on a computer from that era, with many of the same limitations that inspired ingenuity and creativity in an earlier time. Some aspects have been updated and others simplified for ease of use, but in all cases we've tried to preserve the aesthetic of the era. Most of all, we've tried to make it approachable and fun.

If you follow the book, you'll build a computer with a period-appropriate 65C02 processor running at 1 megahertz and accessing 64 kilobytes of memory. You'll get an analog NTSC video output with blocky character graphics and sprites, synthesized audio, and serial ports for loading and saving programs—all through a Parallax Propeller microcontroller that replaces the features of half a dozen legacy chips. You'll even build a fully-mechanical keyboard and a toylike 3D-printed case inspired by the keyboard wedges of the 1980s, complete with joystick ports for games and an expansion port for your own peripherals or cartridges. Once it's up and running, you'll start to program in Cody BASIC and move on to 65C02 assembly.

While the computer itself belongs in the 1980s, the spirit is that of the 1970s—open hardware and open

software that is readily accessible to the end user. Unlike most modern reinventions of the classic home computer, the entire design is intended to be constructable by a single person, at home, using techniques and tools available to today's maker community. All the parts are hobbyist-friendly, and even the more obscure ones are currently in production from historically reliable companies. All the design files, including its own custom BASIC dialect, are released under copyleft licenses. And should the worst ever come to pass, synthesizable implementations of all the core components already exist in the wild.

Building the Cody Computer isn't an incredibly difficult project, but you'll need some basic skills and access to a few things. You'll need to solder a couple of circuit boards, one for the computer and one for the keyboard, and you'll also need to be able to assemble them into a 3D-printed case. All the design files you'll need are provided so that you can order your own boards or make your own tweaks when 3D printing. A large section of this book is devoted to build instructions to help you, but it assumes that you already know the basics.

We've tried to make it easy to source the parts without a lot of hassle. The electronics should all be available through a single order from Mouser, including the keyboard switches, but you may find it more cost-effective to order cheaper keyswitches through another reseller instead. If you've built any projects like this at home, you'll know that sometimes it helps to shop around. We're also assuming that you have access to items such as PLA filament through the

same means you'll use to print the case. The remainder of the items you'll need are things that can be sourced wherever you can find a hardware or craft store.

You'll have to install some software to finish programming the Cody Computer once it's built. One of the key components in the project, the Parallax Propeller, has software that you'll need to use when programming the Propeller's firmware. You'll also need to install a terminal program so that the Cody Computer can exchange data with another device. Lastly, if you want to get into assembly language programming, you'll need to have a 65C02 assembler that you're familiar with. The Cody Computer standardizes on the *64tass* cross-assembler which is also used to assemble the built-in Cody BASIC.

For the best chance of success you should already have some significant experience with electronics, programming, soldering, and 3D printing, or have people around who can help you with the topics you don't know. You'll especially need that knowledge when something doesn't go well and you need to solve a problem. If you've done any programming of any kind, built an intermediate electronics kit, downloaded software to an Arduino, or set up some command-line programs on your computer, you'll already have a lot of the technical background you'll need. If you've screwed up all of those but were able to fix it yourself, you're ready.

In terms of tooling, a good workspace, a good soldering iron, and a reliable if standard fused-filament 3D printer are the most important items to have around. You'll also need to have a means of

obtaining some double-sided circuit boards from the design files, one for the keyboard and one for the main board. You may have to order them from an offshore supply house and expect to have some spares, or perhaps go in with a friend who also wants to build a copy.

Here's an anecdote to give you an idea of what to expect: All the 3D printing was done on a more-or-less stock Creality Ender 3 Pro, mostly with Hatchbox or Inland PLA filaments, and we went through a lot as we tried different designs. For electronics, a standard multimeter was used for most measurements, with a Siglent SDS1104X-E oscilloscope only being used a few times to diagnose problems during prototyping. We ordered our boards from Aisler throughout the project because of their out-of-the-box support for KiCad, but they should be manufacturable by other board houses.

> We didn't need anything especially fancy to build the Cody Computer, nor did we get paid to write any of this. When it came time to get some of the tools we didn't have on hand, we intentionally picked the options that would be most accessible to people financially. In many respects it's kind of amazing it actually works!

WHAT'S A HOME COMPUTER?

What constitutes a home computer varies a lot depending on the era. Because the Cody Computer is channeling the early 1980s, it's worth revisiting the

1970s and 1980s to discuss exactly what computers were like at the time. As with other new technologies being introduced to the marketplace for the first time, there were many new systems being released from a variety of manufacturers large and small, much of it forgotten or otherwise lost outside of collectors' circles. It wasn't just a couple of famous companies and their famous products. There were literally too many to list here.

The earliest home computers resembled a tiny version of the 1960s Batcomputer more than anything else. The Kenbak-1 of the 1970s was made without any microprocessors at all, instead built with what looked like a small city of individual logic chips and programmed via a front panel of buttons and switches. Professional computers of the era were also built from collections of chips like this, though those used more powerful chips with a higher level of integration.

Machines with microprocessors, such as the MITS Altair and the IMSAI 8080 (famously used in WarGames), became available by the mid-1970s. These also sported a blinking-lights-and-switches appearance, with programs generally loaded manually or by paper tape readers. Finding an external terminal to talk to your computer became an adventure in itself. Projects like the TV Typewriter were popular and led to experimentation with input terminals and cheap video output hardware.

A large number of the systems of that era came in kit form, often described in magazine articles that functioned as build instructions or user guides. Single board computers or modular systems became quite popular. Among those would be systems important in

the history of the 6502 microprocessor, such as the Jolt and MOS Technology's KIM-1; that latter device was in many respects the first of the Commodore computers.

Taken as a whole, however, these machines were often more like a minicomputer for the home rather than a home computer. Yet even in this era, much of the home computer culture was being established. Microsoft got its start by selling BASIC interpreters for these systems, while the People's Computer Company created the first of many versions of the open Tiny BASIC instead. Standards for saving and loading programs emerged, such as the Kansas City Standard for storing data on the audio cassettes of the era. Commercial operating systems such as CP/M became available for many systems. And users began sharing programs via magazines, mail, and computer clubs.

The concept of the home computer began to change with systems like the Sol-20 and Apple 1, including the keyboard and video output within the computer itself. By 1977, the Commodore PET, Apple II, and Tandy TRS-80 were all launched to the public as complete systems. Graphics capabilities were limited and the game-system-inspired Atari 800 wasn't released until two years later. At this point, the outlines of the stereotypical home computer became apparent: A wedge-shaped computer, a built-in keyboard, support for cartridges and cassettes for data storage, joystick or controller ports, and output to a dedicated monitor or home television.

By the 1980s, the line between home computer and game system became blurry. Existing game systems received add-on keyboards and BASIC interpreters to

resemble a home computer. The Nintendo was sold in its native Japan as the Famicom, with keyboards, BASIC cartridges, and disk drives made available. Computer manufacturers began including more advanced graphics and sound features in their products. By 1982, the color-video ZX Spectrum was released in the UK, and in the US, the Commodore 64 was released with game-like graphics and sound capabilities. Storage devices improved as floppy drives became more common than cassettes, particularly in the US market.

As the 1980s continued, more advanced computers eclipsed the earlier 8-bit systems. The Amiga, Atari ST, Macintosh, and the IBM PC represented the next generation of computer technology. Yet companies persisted in the 8-bit market. Amstrad released its CPC family with impressive bitmap graphics for its day. Handhelds like the Atari Lynx and Nintendo Game Boy utilized 8-bit 6502 and Z80 microprocessors. The 65816, a 16-bit variant of the 6502, was used in the Apple IIGS (with capabilities often surpassing the Macintosh itself) and Super Nintendo. Despite those successes, by the middle of the 1990s, the 8-bit world was all but gone, save for third-party companies and aftermarket add-ons that gave existing systems a new lease on life.

COMMODORE AS INSPIRATION

While not compatible with the Commodore series of 8-bit computers, much of the inspiration for the Cody Computer comes from that lineage. Commodore produced one of the most influential series of 8-bit computers. Many of their systems were known for

providing an exceptional feature set at a low price, while much of the company's design and marketing had been directed at producing capable systems for the general public rather than computing nerds or enthusiasts.

Along with their significance to the early history of home computing, you'll find that much of the Cody Computer's functionality was inspired by how Commodore did things. Not everyone has firsthand experience with one of these systems, so to provide some historical context, we'll briefly review some of the better-known entries in the Commodore 8-bit family.

Commodore actually began as a typewriter company, moving by necessity into the new markets of electronic adding machines and calculators in the 1960s and 1970s. Competition in the market was brutal, and Commodore began acquiring electronics companies as part of its business strategy. One of the acquisitions was MOS Technology, the company responsible for the 6502 microprocessor. As part of the purchase, Commodore also gained access to the engineering talent behind the company.

Realizing the potential in the home computer market, Commodore began manufacturing computers using its own chips starting in the late 1970s. Future designs would continue to leverage their in-house electronics expertise instead of relying on off-the-shelf components. Commodore's sales pitch marketed their systems as friendly computers that provided amazing features for the price. Despite their successes, changing markets, cutbacks on engineering, and

problematic business practices proved too much to bear; Commodore went bankrupt in 1994.

KIM-1

The KIM-1 was a single board computer produced by MOS Technology in the mid-1970s. Its primary purpose was to serve as a reference system for their 6502 processor. Out of the box it had a keypad and numeric display for interaction and programming, while mass storage was available by connecting to cassettes or paper tape. Clones were made by other companies and aftermarket enhancements included video output. Many of the starter 65C02 projects you'll find on the Internet are, in some sense, the spiritual successors of these early single board computers.

COMMODORE PET

The PET was Commodore's first real entry into the computer market. Many of the characteristics associated with Commodore's computers began with this model. Featuring a 6502 processor, a built-in keyboard, cassette, monochrome monitor, and a copy of Microsoft BASIC, the machine was intended as a more practical computer at its release in 1977. The machine also supported the IEEE-488 bus, providing use of a variety of peripherals and storage devices.

Because of the computer's text-only display, a graphical character set called PETSCII was invented to make games and entertainment applications more feasible. The characters were prominently featured on Commodore keyboards throughout the 8-bit era.

PETSCII graphics remain one of the most uniquely-identifiable aspects of a Commodore computer system, often finding their way into hobbyist graphics and compact homebrew games.

VIC-20

After other research and development attempts at a color PET successor, Commodore released the VIC-20 as a “friendly computer” that could be plugged into your television set. The computer had expansion and cartridge slots, both of which were heavily used because of the computer's minimal standard memory. Commodore replaced the PET's IEEE-488 bus with their own serial version, the IEC bus. The VIC-20 had an optional floppy drive but datasettes were most popular at this point. BASIC was still standard and a joystick was added for gaming.

The VIC-20 also set a precedent for powerful peripheral chips made custom by Commodore. The VIC-20 used the VIC chip for handling video, sound, and other system functions. It produced two-color character graphics at a moderate resolution and four-color character graphics by halving the horizontal resolution, which became the standard approach in Commodore systems. Games and images were displayed by changing the colors and characters themselves. For sound, it produced three programmable square wave channels and a single noise channel.

COMMODORE 64

The best-known of Commodore's computers, the Commodore 64 contained the famous VIC-II and SID chips that made it a compelling video game system. Expansion and user ports existed for cartridges and add-ons, and a stripped-down C64 variant was later released as a console-like game system. Early models of the C64 bore a strong resemblance to the prior VIC-20. Datasettes were still very common but floppy drives became standard for the machine in the United States.

Much of the C64's unique character came from its custom support chips. The VIC-II supported character and bitmap graphics modes at higher resolution than the VIC-20, but continued with the VIC's tradition of a low-color high-res mode and a multicolor low-res mode. It also supported up to eight sprites at a time, including extra functions like collision detection. Raster interrupts allowed programmers to change graphics content while the screen was actually being drawn.

The SID was also a breakthrough for its era, at least within the home computing market. It was a sound chip built around digital synthesis principles rather than being a mere tone generator. It supported a total of three different sound generators called voices, each of which could produce at least four different types of sounds. Based on the principles behind music synthesizers, different waveforms, envelopes, and filters were available to craft audio output.

COMMODORE PLUS/4

The Plus/4 began as a cheap computer to compete with the ZX Spectrum and similar systems. Much like the VIC-20, video, sound, and other functions were combined into the single TED chip, which could produce more colors but lacked many VIC-II and SID features. The computer also shipped with a faster 6502 processor and a more advanced version of Commodore's BASIC.

Management changes at Commodore led to the technology being repurposed into an entire suite of business computers with built-in productivity software, marketed as the successor to the Commodore 64 and priced to match. As a result of these miscalculations, the entire line failed in the American market. In recent years developers have shown the system's full potential, porting existing titles from the C64 and creating new ones—including the well-known *Pets Rescue* platformer in 2019.

THE CODY COMPUTER DESIGN

Having reviewed the systems that inspired it, it's time to learn more about the Cody Computer's own design. The Cody Computer's overall design is quite simple, based around a handful of computer chips and some discrete components. It has a built-in keyboard just like its 1980s predecessors. Instead of using FPGAs and programmable logic, the design is limited to modern equivalents of the chips that would have been available in the era. When a modern option is

unavailable, a close substitute was chosen instead. The Cody Computer was never intended as a product to be sold. It's really a DIY project that can be the jumping-off point for your own designs even if you don't build one as-is.

Like many retrocomputers, the Cody Computer is built around the 65C02 microprocessor. It's a modern variant of the traditional 6502 originally produced by MOS Technology, then Commodore, and finally the Western Design Center. It can run at speeds over 14 megahertz, but the Cody Computer runs it at a mere 1 megahertz for reasons of both simplicity and period authenticity. It shares the same 6502 instruction set as its 1970s and 1980s predecessors, but replaces many of the original 6502's illegal instructions with new ones for bit setting, bit testing, and storing registers on the stack. Some bug fixes are also present. Otherwise it shares the same simple but powerful 6502 design, with a single accumulator register, X and Y indexing registers, 64 kilobytes of addressable memory space, and a variety of powerful but easily comprehensible addressing modes.

The Cody Computer also relies on the Propeller, a very powerful and completely custom microcontroller created by Parallax, a small company with a long commitment to education, hobbyists, and bespoke engineering. It dates to the early 2000s and has a total of eight separate processors, called cogs, that can run up to 20 million instructions per second. Its hub memory region contains 32 kilobytes of RAM and 32 kilobytes of ROM, including an interpreter for Parallax's SPIN programming language. All of this is

available in a 40-pin DIP package that fits with the overall aesthetic of the Cody Computer.

The Propeller is the Cody Computer's equivalent of the VIC, TED, and other custom chips. Out of the eight cogs, we devote five to video generation, one to sound generation, one to serial communication, and one to managing the data and address bus for the 65C02. For performance reasons the Propeller is programmed directly in PASM, the Propeller's low-level RISC instruction set, rather than SPIN. From the 65C02's perspective it doesn't matter, as the Propeller presents itself as memory-mapped hardware.

MEMORY

The Cody Computer can address a total of 64 kilobytes of memory. The lower 40 kilobytes of memory are all handled by a single AS6C1008 static RAM chip. A single page of memory is mapped to a 65C22 Versatile Interface Adapter for input and output. The remaining 24 kilobytes of memory are all handled by the Propeller chip itself. 16 kilobytes are used as shared RAM for video and simulated peripherals.

Instead of a separate ROM chip, the Cody Computer's ROM is actually included inside the firmware used by the Propeller, and when memory accesses hit the appropriate region, the ROM contents are returned. The top 8 kilobytes of RAM store the Cody BASIC ROM and a copy of the character set. In reality these are kept as 8 kilobytes in the Propeller immediately after the shared RAM section.

INPUT AND OUTPUT

Most of the Cody Computer's I/O is controlled by a single 65C22 Versatile Interface Adapter (VIA). The 65C22 contains two bidirectional 8-bit I/O ports, a shift register, some additional handshaking pins, and internal timers.

One of the two I/O ports is used to scan the keyboard and joysticks, all of which are wired together into the same matrix. Three pins are used to select one of eight rows (six keyboard rows and two joysticks) with the help of a CD4051 1-of-8 switch, with the remaining five pins used to read in the keys or joystick buttons for that row.

The other I/O port and the shift register are both wired to a general-purpose expansion port where they can be used to interface with other devices. The 65C22's handshaking lines are instead used to detect whether a cartridge containing an SPI EEPROM is present.

SERIAL PORTS

The Cody Computer has two serial ports, both of which can operate at speeds of up to 19200 baud. They're actually implemented as a dual UART peripheral running in a single cog on the Propeller. Both UARTs are hardcoded to support only an 8-N-1 protocol (one start bit, eight data bits, no parity bit, and 1 stop bit). Each UART is polling-based but utilizes ring buffers to reduce the need for 65C02 intervention.

It's assumed that the serial channels being used are unlikely to be prone to errors, particularly at the relatively low rates supported by the emulated peripherals. Some checks for simple errors are performed at the UART level, and data sent using the standard serial protocol contains no checksums or similar measures.

One of the serial ports is actually the same port as the Prop Plug connection for programming the board. This is intended to connect to another system (such as a terminal application) to load and save data and programs. It would even be possible to build a Datasette-like device that could be interfaced via this connection. The other serial port is routed to the expansion slot alongside the pins connected to the 65C22 VIA.

VIDEO

Video output is handled by the Cody Video Interface Device (VID) peripheral implemented in the Propeller. It supports a character graphics mode where the screen is divided into 40 columns and 25 rows of characters. Each character has four horizontal pixels and eight vertical pixels, similar to the Commodore 64's multicolor character mode. Each pixel can be one of four colors, two of which are unique to the individual screen location and two of which are shared by the entire screen. There is also a similar multicolor mode for bitmapped graphics. High resolution character graphics and bitmapped graphics modes allow a more standard eight-by-eight pixel character region at the

expense of some of the more game-like graphics features.

In fact, the VID has many game-focused features. Up to 8 multicolor sprites can also appear on each line. Smooth scrolling is supported. Additional features allow changing some of the data dynamically to allow more colors, characters, or sprites to appear on the screen. These allow raster-interrupt-like effects through the use of built-in video chip features. However, some of these features are only available in the normal multicolor modes and not the more limited high-resolution graphics modes.

Video generation is very complex. In the Cody Computer, most of the Propeller's internal resources are devoted to the video system. One of the Propeller's cogs is devoted to generating the actual NTSC video signal while four other cogs run in the background to generate video data. These cogs take the screen memory, color memory, character memory, and sprite memory contents and generate pixel colors that are included in the NTSC signal.

SOUND

Audio is produced by the Cody Sound Interface Device (SID), a simplified version of the famous SID from the Commodore 64. This peripheral is also implemented using the Propeller and contains a rough emulation of the SID in a single cog. The peripheral supports three voices with Attack-Decay-Sustain-Release (ADSR) envelopes. The SID's sawtooth, triangle, pulse, and white noise waves are supported,

and it also has a rudimentary attempt at features such as ring modulation.

However, the Cody SID is not a full SID emulation. Decay constants are linear instead of exponential and filters are not implemented. Many other differences also exist, and it's best to view the Cody SID as a SID-like device with its own unique characteristics.

COMPARISONS AND CONTEXT

The Cody Computer is not compatible with any of the Commodore lineage (though, to be fair, they were rarely very compatible with each other). In terms of inspiration and design decisions, however, there is a significant debt. Much of the overall philosophy and even some specific details are very similar. During development I sometimes considered it a “Commodore Junior”, a simplified system that was also an homage to the Commodore 64 in particular. I also took inspiration from how much the Plus/4 engineers were able to preserve a Commodore feeling despite stripping so much of the C64 away.

For example, the Cody Computer has two video modes inspired by the C64 and Plus/4. Its character-based graphics mode is influened by those machines' multicolor character mode. Similarly, the sprite graphics are very similar to the VIC-II's multicolor sprites, even though they don't support features like collision detection and scale-doubling. Built-in support for additional sprite banks is likewise influenced by sprite multiplexing routines from the C64. Its bitmap mode is also very similar to those on the C64 and Plus/4, falling somewhere between the

VIC-II and TED in terms of its limitations. The high-resolution mode is also very similar to the Commodore line and may make it easier to port certain applications.

Audio functionality is largely copied from the Commodore SID design. The Propeller uses a port of a SID emulation library from the Arduino to mimic basic synthesis functions, providing waveforms and ADSR functionality very similar in nature to the SID chip. Many other features including combined waveforms and filters were intentionally not implemented. The SID registers are mapped to the same locations as on the C64, and there is at least a minimal level of C64 compatibility.

Two side-mounted joystick ports are available as on later Commodore machines, but they're wired into the keyboard matrix as rows. The keyboard itself is far from a standard Commodore layout and actively avoids the multi-labelled PETSCII hieroglyphics of times past. A dedicated expansion port exposes many of the 65C22 VIA's I/O pins and a second UART from the Propeller, but it does not expose the 6502 bus as on Commodore machines. No dedicated “user port” exists, but the same serial port used to exchange programs is intended for something similar.

For loading and saving files, standard serial communication is used like a very simple datasette. For the Cody Computer, a dedicated mass storage device is not only excessive but ruins the retro spirit. Instead, the intended target is a terminal or file application running on another computer or phone. However, it wouldn't be difficult to build a Datasette-

like device that could interface with the Cody Computer over this serial port.

The Cody BASIC provided with the computer is closer to a tokenized Tiny Basic from the 1970s than to a 1980s Microsoft BASIC. It supports 16-bit integer math rather than floating point, has a limited set of commands, and has a limited feature set. However, the Cody Computer's extensions, including arrays and strings, were largely inspired by Microsoft BASIC from the Commodore. Cody BASIC is also tokenized, though it stores the programs as plain ASCII to make it easier to load and save BASIC programs from modern computers. Tokenization happens when loading, requiring some input delays by the sender so that the tokenizer can keep up.

For compatibility reasons the software uses what is essentially an extended ASCII, but the PETSCII graphics characters are available. Cody BASIC does not allow directly entering the characters into the input, but the character codes can be specified in **CHR$** commands. Cody BASIC also understands a reserved set of character codes that work as control codes, including clearing the screen, changing foreground and background colors, and implementing a reverse-field effect. So in most respects, Commodore-style PETSCII graphics are still possible even in a BASIC program, just done differently.

The Plus/4 approach of packing a huge amount of functionality into the TED chip was a major inspiration for using the Parallax Propeller as a similar device. The Propeller's advanced capabilities then opened the door to creating a more C64-like set of features. The low-resolution PETSCII graphics in the Cody Computer's

font were inspired by various 40-column extensions written for the VIC-20. Having grown up with a Commodore 64, the source of the inspiration was never far away.

In fairness, many of the major decisions were taken on the basis of what elicited the best response from one small dog. I wouldn't have done it like this. My original thought was to add a microcontroller or two and create a modernized PET. Instead the real Cody preferred SID and TED music, YouTube videos and emulations of Commodore games, Propeller demos on the TV, and so many other things I attempted to find some way to work in.

In many respects, he reminded me of myself as a very young child working on computers, electronics, or rockets with my father or uncle. My brain liked what it saw and had a glimpse of the big picture, yet I found myself overwhelmed by all the strange details and held back by tiny hands. And Cody was, in so many ways, a small dog with the heart and mind of a very young boy.

In any event, thanks to my four-legged management, what you see here is what we got. Yet Cody demonstrated better acumen, wisdom, and aesthetics through his smiles, gestures, and tail wags than I ever encountered in my working career. I'll always have doubts about certain design choices or implementation details on my part, but I think Cody was right about the big picture. His apparent interest (or lack thereof) determined so much of what did and didn't make the cut. While he was there for so much of this work, he's no longer here for one last final

inspection, big smile, or wag of the tail. But I do hope he would have been proud.

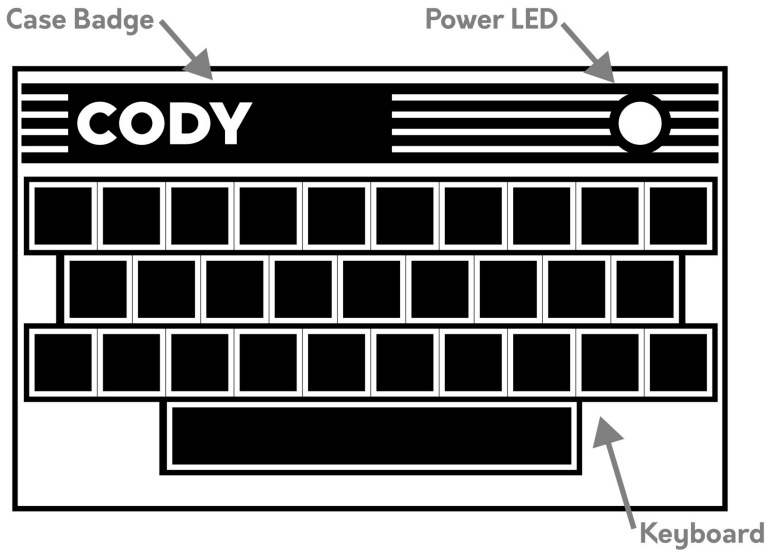
2

Hardware and Firmware Design

INTRODUCTION

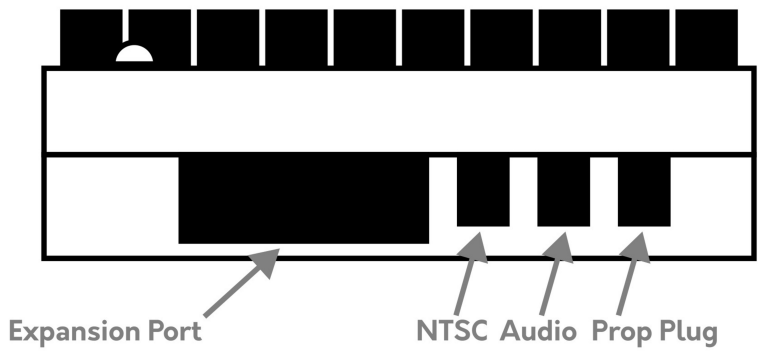
In finished form the Cody Computer is small by computer standards, fitting into a rectangle about the size of a large laptop trackpad and a couple of inches thick. Much of the industrial design is inspired by the Commodore 64 and similar 1980s computers with additional influence from the collected works of Tomy, Playskool, or Fisher-Price. The overall intent was to produce something that would be identifable as an old-school computer yet come across to a bystander as unintimidating, fun, and approachable.

From the top view you'll notice a prominent case badge (complete with an inlaid rainbow-colored badge in the finished product), a large 10mm power LED (blue according to the design, but you can replace it), and a 30-key keyboard. The keycaps are custom but compatible with Cherry MX keystems, though the Cody Computer uses a nonstandard spacing to fit everything into such a small package. Standard keycaps won't work unless you decided to saw them down.



Top of Cody Computer showing case badge, power LED, and keyboard.

While you'll spend most of your time from this position, looking down at the machine and using the keyboard, much of its most important functionality is elsewhere. In particular, a variety of ports on the back and right side of the computer are used to interface with the outside world.



Back of Cody Computer showing expansion port, video, audio, and Propeller port.

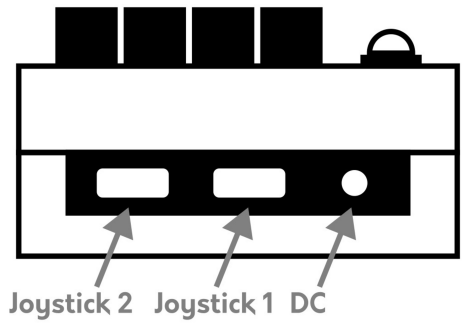
Most of the Cody Computer's ports appear on the computer's lower back panel. The largest is an expansion port that can be used to interface external devices or boot from cartridges. We'll discuss the electrical characteristics of the expansion port later. For now, it's enough to know it's here.

Next to the expansion port are RCA jacks for NTSC composite video and mono audio output. The video output can be connected to any device that supports NTSC video input (unless, in rare circumstances, the display or converter is incompatible with the software-generated video from the Cody Computer). The audio output is generally connected to a splitter and then to the left and right channels of the display.

The last connector on the back is a four-pin DuPont connector compatible with Parallax's specifications for their Prop Plug. Initially used to download the firmware to a finished Cody Computer, it later doubles as a serial communications port to other computers,

mobile phones, or compatible devices using the same mechanism.

The remaining ports are on the computer's right side (as viewed from the top).



Right side of Cody Computer showing joystick ports and DC power connector.

Two of the ports are standard Atari-style joystick ports used by many of the best 1980s computers. Purely digital, they lack support for the analog paddles of the Atari and Commodore systems, but otherwise are nearly identical. Each presents as a male DB9 connector suitable for use with any standard Atari-compatible joystick.

The other port is the DC barrel jack responsible for delivering power to the Cody Computer. Input is typically around 5 volts delivered from a wall-wart or other transformer plugged into a mains outlet. Because no switch is built into the Cody Computer, I suggest connecting an external inline switch between the DC jack and wall-wart.

MECHANICAL DESIGN

We'll explain how to build the Cody Computer in the chapter on assembly, but first it's good to have some idea of what you're actually building. Aside from a few core components, switches, and fasteners, the Cody Computer is designed to be printed on any reasonable fused-filament 3D printer.

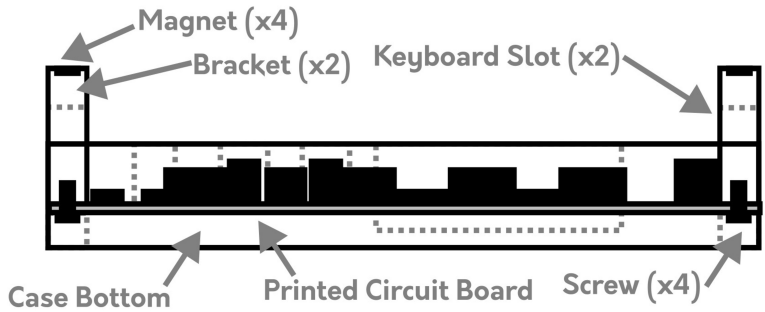
The case itself is held together with some semi-permanent screws on the lower half that also secure the main printed circuit board. The screws also hold some slotted brackets for the keyboard module, and some rare earth magnets hold a removable top section to finish the enclosure.

In addition to being easy to assemble, the Cody Computer is designed to be easy to take apart. The magnets allow the top of the case to be easily removed for a closer inspection of the keyboard and case interior. The keyboard itself can be easily slid out of its brackets to expose the main printed circuit board for the entire system. If you do this a lot, you may find yourself in need of some additional glue, but the idea is for the system to be open for inquiry in every possible way.

CASE BOTTOM

The bottom subassembly, built around the case bottom itself, is essentially a stack. The printed circuit board containing the circuitry for the computer rests on standoffs at the base of the case. Above the PCB are two brackets used to provide some support for the top

of the case, as well as a mounting location for the keyboard.



Cutaway view of the bottom section of the Cody Computer.

The entire stack is held together by four screws that are inserted from the bottom of the case through holes in the PCB and into the mounting brackets at top. Pilot holes for the screws are designed into the brackets, though they may need to be adjusted for particular printers.

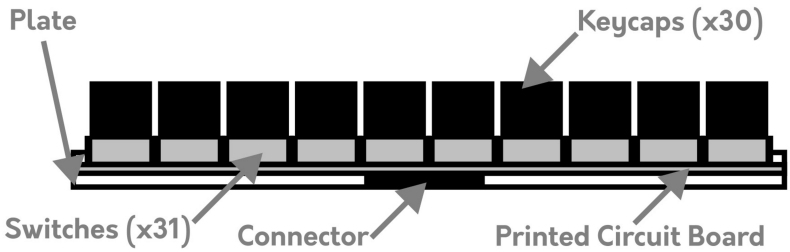
Holes in the back of the case expose the expansion port, video and audio connectors, and serial port on the back of the printed circuit board.

The mounting brackets contain slots to slide the keyboard assembly into. The right bracket also contains punchouts for the joystick ports and DC power connector. Recessed holes at the top of the brackets contain magnets that will anchor to the case top. The keyboard itself is a separate piece.

KEYBOARD MODULE

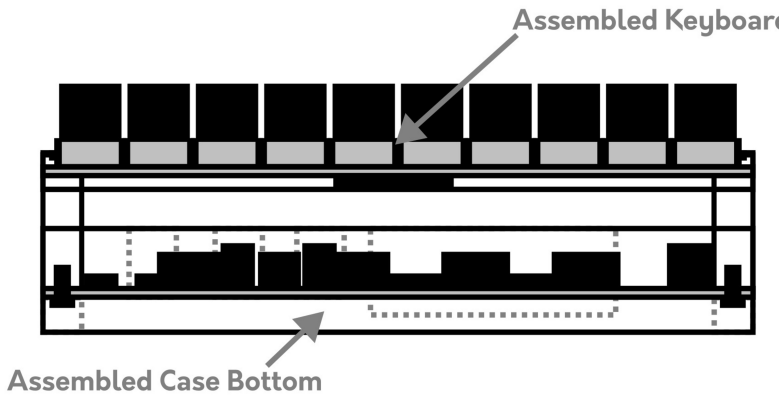
The keyboard module consists of a keyboard plate, a printed circuit board, and a set of Cherry MX

compatible mechanical keyswitches and their keycaps. The printed circuit board rests along the bottom of the keyboard plate, with the keyswitches pressed in from the top. The switches are soldered into the PCB, along with a DuPont connector, and the keycaps pressed on.



Cutaway view of the keyboard module.

The keyboard plate is sized to friction-fit into the slots on the brackets mentioned earlier. One side of the keyboard is slid into place, followed by the other. This allows the keyboard to be removed and the underlying PCB for the Cody Computer to be examined for educational purposes.

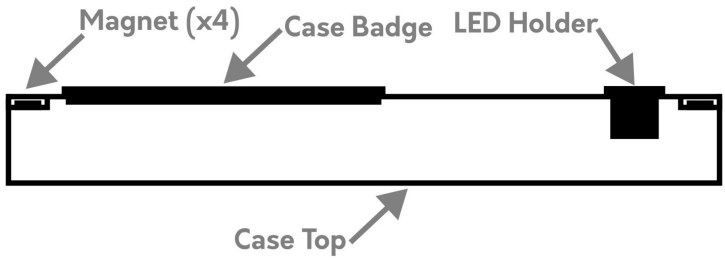


Bottom assembly with keyboard module slotted into place.

With the keyboard in place, all that remains is the top cover for the Cody Computer.

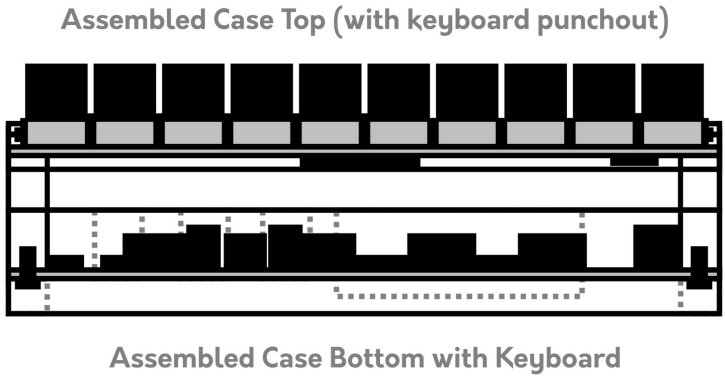
CASE TOP

Similar to the bottom cover, the top cover has holes for the keyboard, case badge, and the holder for the power LED. These parts are glued or press-fit to the top of the case. Four bosses for magnets also exist on the top of the case. In these locations magnets are glued into place, matching those inserted into the brackets attached to the lower half of the computer.



Cutaway view of the top section of the Cody Computer.

With the magnets correctly affixed to the brackets and the case top, the top cover can be easily popped on and off the remainder of the assembly.



Cutaway view of the assembled Cody Computer.

OPENSCAD FILES

All mechanical designs for the Cody Computer were created using OpenSCAD and released under an open-source license. This means that the original design files are available to review and even change if you need to. The generated STL files for each component are available and should be the primary

source for printing Cody Computer parts under normal circumstances. The OpenSCAD files were only there to produce the canonical set of STLs for the Cody Computer using a standard open source tool.

However, the OpenSCAD files are available if you need to adjust them for your own 3D printer or parts. They're direct translations from pencil-and-paper sketches so they aren't particularly pleasant to work with. The files aren't done in a parametric CAD style, magic numbers are everywhere, and changes to one measurement will often necessitate other changes. To the extent that changes are possible, it's wise to limit them to adding or subtracting fudge factors for specific 3D printer setups or part substitutions.

```
module CaseBottom() {

    difference() {

        union() {

            // bottom with cavity
            difference () {

                // main shape
                hull() {

                    translate([0, 2, 2]) rotate([0, 90, 0]) cylinder(h=165, r=2, $fn=20);

                    translate([0, 103, 2]) rotate([0, 90, 0]) cylinder(h=165, r=2, $fn=20);

                    translate([0, 0, 25]) cube([165, 105, 1]);

                }

                // interior
                translate([2, 2, 2]) cube([161, 101, 25]);

            }

            // PCB mounting standoffs
            translate([2.5 + 5, 2.5 + 5, 0]) cylinder(h=9.63, d=10, $fn=20);
            translate([2.5 + 5, 2.5 + 5 + 90, 0]) cylinder(h=9.63, d=10, $fn=20);
            translate([2.5 + 5 + 150, 2.5 + 5, 0]) cylinder(h=9.63, d=10, $fn=20);
            translate([2.5 + 5 + 150, 2.5 + 5 + 90, 0]) cylinder(h=9.63, d=10, $fn=20);
        }

        // screw heads
        translate([2.5 + 5, 2.5 + 5, 0]) cylinder(h=7.63, d=6.5, $fn=20);
        translate([2.5 + 5, 2.5 + 5 + 90, 0]) cylinder(h=7.63, d=6.5, $fn=20);
        translate([2.5 + 5 + 150, 2.5 + 5, 0]) cylinder(h=7.63, d=6.5, $fn=20);
        translate([2.5 + 5 + 150, 2.5 + 5 + 90, 0]) cylinder(h=7.63, d=6.5, $fn=20);

        // screw holes (gives a couple of layers to punch out rather than using supports)
        translate([2.5 + 5, 2.5 + 5, 7.63 + 0.20]) cylinder(h=10, d=3.1, $fn=20);
        translate([2.5 + 5, 2.5 + 5 + 90, 7.63 + 0.20]) cylinder(h=10, d=3.1, $fn=20);
        translate([2.5 + 5 + 150, 2.5 + 5, 7.63 + 0.20]) cylinder(h=10, d=3.1, $fn=20);
```

```
        translate([2.5 + 5 + 150, 2.5 + 5 + 90, 7.63 + 0.20]) cylinder(h=10, d=3.1, $fn=20);

        // vent holes
        for(count = [0 : 6]) {
            translate([15 + count * 8, 15, 0]) VentHole();
            translate([15 + count * 8, 105 - 15 - 30, 0]) VentHole();
            translate([165 - 15 - 4 - count * 8, 15, 0]) VentHole();
            translate([165 - 15 - 4 - count * 8, 105 - 15 - 30, 0]) VentHole();
        }

        // expansion port
        translate([2.5 + 34.2, 0, 4]) cube([58, 10, 17 + 10]);

        // video port
        translate([2.5 + 95.7, 0, 11.23]) cube([12, 10, 17]);

        // audio port
        translate([2.5 + 114.9, 0, 11.23]) cube([12, 10, 17]);

        // prop plug port
        translate([2.5 + 134.1, 0, 11.23]) cube([12, 10, 17]);

        // side panel
        translate([0, 10 + 2.5, 11.23]) cube([5, 80, 15]);
    }
}
```

*Example from **Case.scad** showing heavy use of magic numbers.*

The **Case.scad** file contains the designs for the case top, case bottom, LED holder, badge, and badge inlays. Each portion of the design resides in its own SCAD module (**CaseTop**, **CaseBottom**, **LEDHolder**, **LEDHolder**, **CaseBadge**, and **BadgeInlay**). In some cases these modules rely on other modules within the same file.

The **Keyboard.scad** file contains the designs for the keyboard plate (as the **KeyboardPlate** module) and keyboard brackets. The two keyboard brackets are somewhat different as one contains punchouts for the DB9 Atari joystick ports (**KeyboardBracketWithHoles**), while the other does not (**KeyboardBracket**). A helper module, **DB9Hole**, contains the shape of the hole.

The **Keycap.scad** file contains the keycap designs. The **Keycap** module has the design for a normal keycap, with the legend specified as a parameter. The designs for the keycap legends exist as SVG files in a

subdirectory, with the appropriate SVG legend being subtracted from the keycap's face based on the parameter.

The spacebar is a special keycap and has its own module, **Spacebar**. Supporting modules are **KeySlice**, which generates a two-dimensional keycap shape used for extrusion, and **KeyStem**, which creates a Cherry MX-compatible keystem. The tolerances for a suitable keystem are quite small, and if you need to modify any of the SCAD files directly, it will likely be this one.

The **Keychain.scad** file is unused for the actual Cody Computer build, but I've included it anyway. It's a design for a simple keychain based on the Cody Computer's case badge and has similar assembly requirements. During the Cody Computer's development, one of these was used to test the longevity of air-dried clay for keycap legends.

ELECTRONIC DESIGN

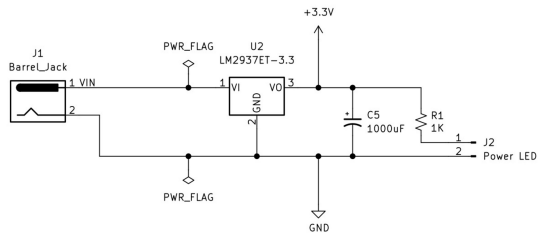
We've discussed the overall concept behind the Cody Computer and how it fits together mechanically, so now we'll talk about how the actual electronics work. In many respects this is a guided tour through the schematics, starting with the power supply and going on to the microprocessor, RAM, and other major components.

While excerpts of the schematics are available here, the full schematics are also available as original files or PDF exports. It's recommended to follow along with those if you're particularly interested in any of the electrical details. The Cody Computer was designed

using KiCad 5 and later KiCad 6, so even the software used to design it is available as free and open source software.

POWER SUPPLY

The Cody Computer's power supply circuit is simple but very important. Almost all of the glitches and transient faults encountered when developing the computer were actually the result of glitches in the power supply, either from third-party power supply boards or from loose connections in the wires supplying power to the breadboards.



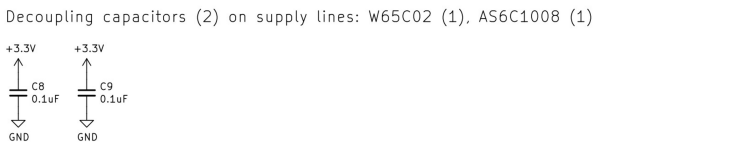
Schematic of the Cody Computer's power supply.

For the power supply circuit, a standard DC barrel jack (J1) supplies power from a wall-wart transformer or other device. The external device typically supplies power at a level around 5 or 6 volts. This is regulated by a LM2937ET-3.3 voltage regulator (U2) that produces 3.3 volts from the input. There's also a rather large capacitor (C5) to take care of any minor wobbles. A 1 kilohm resistor (R1) connects to a 2-pin plug (J2) for the power LED, so that the LED turns on whenever power is being supplied to the circuit as a whole.

The power supply circuit is a subset of the power supply circuit featured in Andy Lindsay's *Propeller Education Kit Labs: Fundamentals*. Aimed at students, that circuit was powered from a 9 volt battery and had regulators for both 5 volts and 3.3 volts. Only a subset of that circuit is needed here for the 3.3 volt supply.

Andy Lindsay's text and the associated kit were my introduction to the Propeller and were very useful in getting started. I went through a few 9 volt batteries during my own later experiments and ran into some weirdness when the batteries started to go dead. For very long-term projects use your bench power supply.

There are also individual 0.1 microfarad decoupling capacitors scattered throughout the circuit, typically one per integrated circuit and sometimes more. These are omitted from the simplified schematics in this section but appear in the full schematic. We place these capacitors very close to the positive voltage and ground pins on each integrated circuit to ensure a reliable and noise-free power supply.



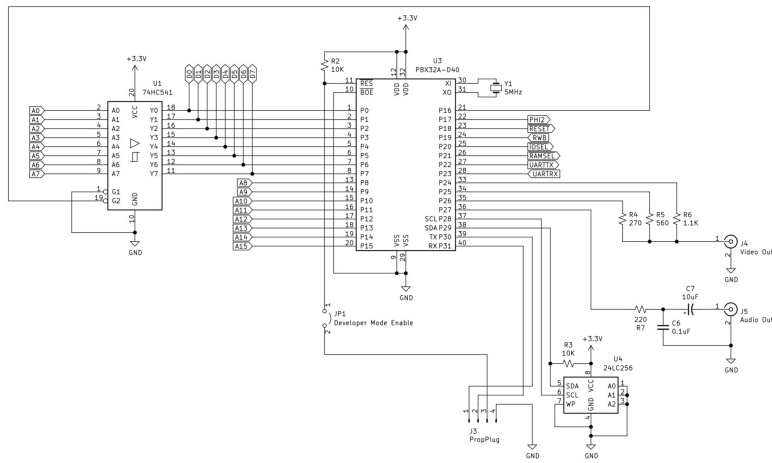
Part of a Cody Computer schematic showing some decoupling capacitors.

Note that as the Cody Computer doesn't have a built-in power switch because of space constraints, it's beneficial to get an inline switch. There are many power switches that accept a DC jack connector, and similar switches have been used on everything from the ZX81 to most of today's Raspberry Pi models. Such items are available from Amazon, Sparkfun, and a variety of other retailers, usually costing less than a few dollars.

PROPELLER

Much of the circuit is offloaded to a single microcontroller, the Parallax Propeller. It does most of the same jobs as Commodore's old VIC or TED, and sometimes a lot more. Fortunately, it's able to keep up as it's a rather unique (and open-source) device that actually contains eight lightweight processor "cogs" on a single chip. It's used to clock the 65C02 microprocessor, monitor and decode the 65C02 bus, perform serial communications, and generate video and sound. The complexity of the schematic sheet containing the Propeller gives you an idea of just how

important the chip is to the Cody Computer's functioning.



Schematic of the Propeller and closely-related circuitry.

When the circuit powers up, the Propeller (U3) wakes up using its own internal oscillator. It later switches to a 5 megahertz crystal (Y1) which internally is multiplied by 16 to give an actual clock frequency of 80 megahertz. Because each Propeller instruction takes four cycles (with some exceptions), there are 20 million instructions per second *per cog*. That's a lot of CPU cycles, especially when you take into account the Propeller's built-in support for video generation. On the other hand, it has a lot to do!

On startup, it checks to see if a program is being uploaded via the Prop Plug. If a program is being uploaded, the Prop Plug (J3) generates a reset pulse and begins sending the program. We need this feature to program the Propeller for the first time, but after

that, external devices shouldn't be able to reset the computer. To inhibit this, a small jumper (JP1) connects the Prop Plug reset pin to the Propeller's reset pin and a pull-up resistor (R2). When removed, the Prop Plug's reset pin is disconnected so the Propeller's reset pin cannot be pulled low and trigger a reset. Other features are unaffected, allowing it to work as a serial user port to communicate with other devices.

Aside from the rare circumstance when the Propeller is being programmed, it will load its firmware from a 32 kilobyte I2C EEPROM (U4), a 24LC256 or similar. The Propeller has an internal 64 kilobyte memory space of its own, half of which is RAM and half of which is ROM. The content of the 32 kilobyte I2C EEPROM is copied into the RAM portion and then run, first using the Propeller's built-in SPIN interpreter, but soon dropping directly into the Propeller's own assembly language. Contained in that EEPROM is not only the program for the Propeller but also the ROM for the 65C02.

Once the Propeller begins running its code, most of its I/O pins are used for communicating with the 65C02's system bus and other devices. Eight of the Propeller's I/O pins, P16 through P21, are used to generate the 65C02's PHI2 clock signal and reset pulse, chip select signals for other devices on the board, and monitor the read/write signal from the 65C02. An additional two pins are used for a second UART that interfaces with the Cody Computer's expansion port.

When running, one of the Propeller's many responsibilities is to decode the 65C02's address bus. Along with the mentioned read/write signal, it uses I/

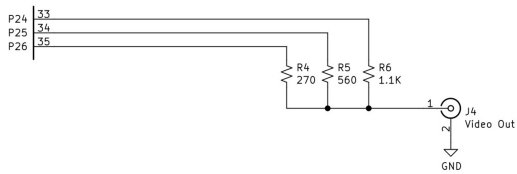
O pins P0 through P15 to interface with the 65C02's address and data buses. We're even able to share some pins and minimize part count because of a unique characteristic of the 65C02's bus. The 65C02 puts the address on the address bus throughout a clock cycle, but it only puts the data on the data bus during the latter half of the cycle when PHI2 is high. During the first part, when PHI2 is low, the data bus is essentially disconnected.

This means that we can actually share the same pins on the Propeller (P0 through P7) for both. We just need a way to control the lower eight bits of the address bus and shut them off to avoid a collision when PHI2 is high. To solve that problem, a 74HC541 buffer (U1) sends the lower eight address bits to the Propeller when enabled. When disabled, its outputs are also tristated, allowing the data lines access instead.

This technique can be used by any 6502-based system, not just a Propeller-based one. In the Propeller community it became popularized from Dennis Ferron's PROP-6502 and Jac Goudsmit's Propeddle, both of which used it to solve a similar problem of conserving I/O pins on the Propeller.

The Propeller is also responsible for generating NTSC video. The chip itself has built-in circuitry for generating NTSC or PAL video output, generating a variety of colors. However, the circuitry still needs to be programmed on the software side and interfaced on

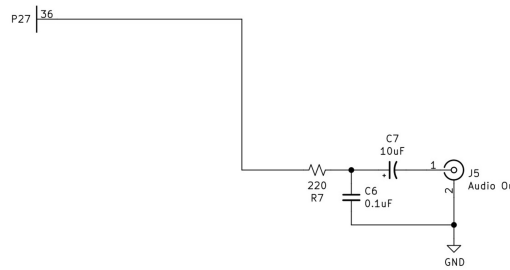
the hardware side using a digital-to-analog converter (DAC) made of resistors.



Schematic detail showing the video output pins, resistor DAC, and RCA jack.

For the Cody Computer, I/O pins P24 through P26 are used as the video output pins. These are summed into a single analog signal through a DAC made of up of 1.1 kilohm (R6), 560 ohm (R5), and 270 ohm (R4) resistors connected to an RCA composite video jack (J4). The Cody Computer uses 1% tolerance resistors for this particular part of the circuit, but the values aren't that finicky. Some resistor values in the same ballpark should suffice for our purposes. The resistor values themselves come from André Lamothe's *Unleashing the Propeller C3* about the eponymous credit card sized computer.

Audio output is handled by the Propeller as well. The Propeller's internal counters and support for pulse width modulation is used to output a pulse with a changing duty cycle. The stronger the signal, the longer the pulse stays on before turning off. This output, in turn, gets converted by support circuitry into a normal audio signal.



Schematic detail of the audio circuit.

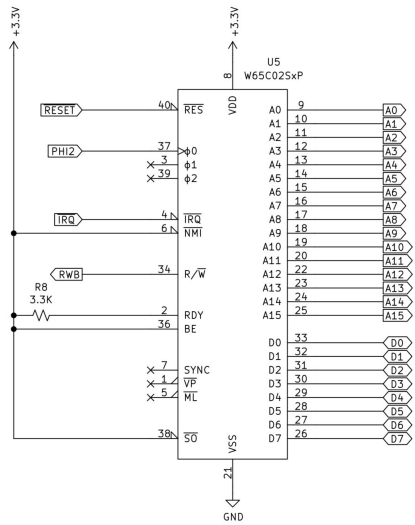
For the Cody Computer, Propeller I/O pin P27 is used for the audio output. It connects to a 220 ohm resistor (R7) which is itself connected to a 0.1 microfarad capacitor (C6). The resistor and capacitor essentially smooth out the on-or-off pulses generated by the Propeller. This output is further filtered by a larger 10 microfarad capacitor (C7) that also couples the output to the RCA output jack (J5).

The circuit itself comes from a September 2006 Propeller forum posting by Parallax engineer Paul Baker, who noted that the circuit was not necessarily “optimal” but would suffice. I've been using it since I started prototyping with the Propeller on a breadboard, and it's been a part of what became the Cody Computer ever since. You'll find many variations of the same circuit floating around with different component values for different frequency cutoffs.

65C02

The Cody Computer's brain is the 65C02 microprocessor (U5). The actual computing performed by the Cody Computer happens entirely as a result of

the 65C02's actions. It's also responsible for directing what happens in the rest of the circuit, though the Propeller assists greatly when it comes to decoding the 65C02's address bus.



Schematic detail showing the 65C02 microprocessor and its connections.

The Propeller's generated PHI2 signal is directed to the 65C02's input on pin 37; this pin has gone by various names over the years, but in modern variants, it's essentially the PHI2 clock input. A Propeller-generated reset pulse is also applied to its reset pin on startup. The 65C02's IRQ line is connected to the corresponding pin on the 65C22 I/O chip so that timers and output port events can signal the processor when needed.

The 65C02's other interrupt line, the non-maskable interrupt (NMI), isn't used in the Cody Computer and is connected to 3.3 volts. Several other 65C02 pins, such as those for setting overflow or enabling the

address bus, are also tied high. Some unused pins are left unconnected and do not pose a concern for our purposes.

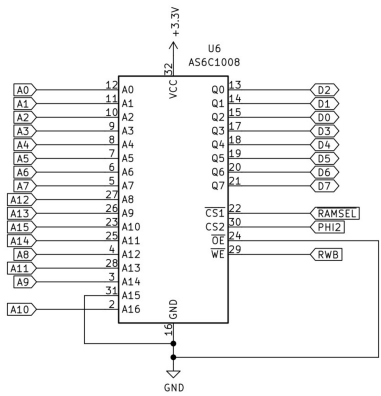
One notable pin is the RDY pin, which is connected to a 3.3 kilohm pull-up resistor (R8) rather than directly tied high to 3.3 volts. This is because on the 65C02, a **WAI** (wait for interrupt) instruction can actually make the RDY pin go low. The 65C02 has no built-in pull-up resistor to deal with this problem. Without a pullup resistor, the 65C02 would essentially be connecting the positive voltage to a logic zero when a **WAI** instruction runs. To avoid that problem, there needs to be a pull-up resistor.

The 65C02's other connections are to the system bus. The 65C02's address pins (or a subset thereof) are wired to the Propeller, SRAM, and 65C22. The data bus pins are similarly connected. Lastly, the 65C02's RWB pin, a read-write strobe indicating whether the current bus operation is a read or a write, is connected to the same devices and completes the necessary bus signals. The PHI2 clock generated by the Propeller is used throughout the entire circuit instead of the PHI2 output from the 65C02. The Propeller generates the master clock, so the 65C02's PHI2 output is left unconnected.

RAM

Most of the Cody Computer's RAM is provided by a single AS6C1008 static RAM chip (U6). The chip is actually a 128 kilobyte memory chip, but the Cody Computer uses less than half of that—40 kilobytes reside in the static RAM and the top 24 kilobytes are

inside the Propeller itself. Unfortunately, while there are 32 kilobyte static RAM chips and 128 kilobyte static RAM chips readily available, modern production of 64 kilobyte static RAM is nonexistent. As a result, designers just use the next biggest size and ignore the extra space.



Schematic detail showing static RAM connections.

The static RAM itself is rather unremarkable. The address and data pins come directly from the 65C02, as does the read/write strobe indicating the type of memory operation in progress. The PHI2 clock and chip select both come from the Propeller, which is responsible for decoding addresses and selecting the appropriate chip.

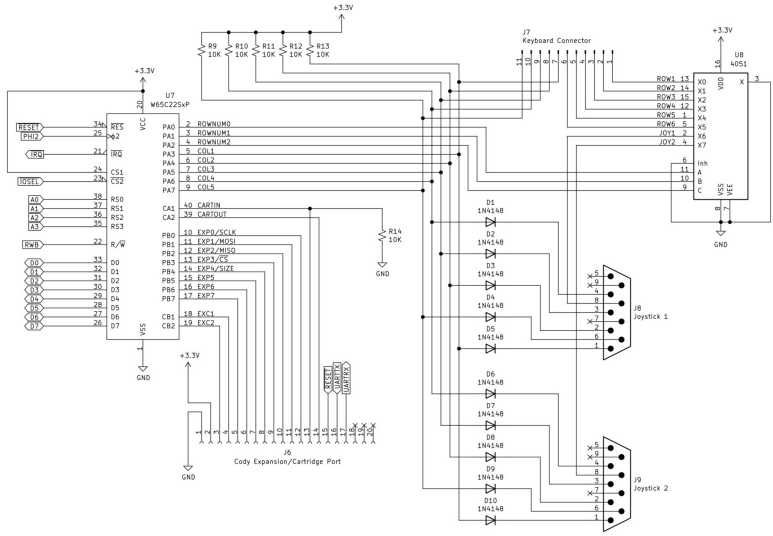
If you look closely at the address and data lines you'll realize they don't match up with the exact same line on the 65C02. For example, the 65C02's address line A12 is connected to the static RAM's address line A8. It may appear to be an error, but it's a quite intentional choice. The static RAM is really just a sequential bunch of byte-sized buckets, and it doesn't

care what 65C02 address maps to its own internal address as long as the mapping is one-to-one.

You can't use this in all cases, but for static RAM chips and similar, switching around the lines like this is a common trick when you're trying to route your printed circuit board. That's what happened to the Cody Computer; it was easier to route the connections if some of the address lines were moved around.

65C22 AND I/O

Aside from two serial ports provided by the Propeller, all input and output from the Cody Computer is handled by a single 65C22 Versatile Interface Adapter (U7). We use some additional circuitry to assist in scanning the keyboard, thus freeing up more of the 65C22's I/O pins for an expansion port. In general, the Cody Computer's I/O is there to provide mechanism, not policy. In other words, you have direct access to I/O pins which you can program however you want, whether that's to perform modern SPI or I2C communications or just turn individual lines on and off. The only exception is when a Cody Computer cartridge is inserted into the expansion port, at which point certain pins read binary code from an external SPI memory.



65C22 and associated I/O ports.

The 65C22 is connected to the system's data and address buses, with the PHI2 clock and chip selects being provided by the Propeller. The 65C22 also has an /IRQ pin that's connected to the 65C02's own interrupt pin, thereby letting the 65C22 trigger interrupts based on timers or I/O events. The remainder of the 65C02's pins are dedicated to two output ports, port A and port B, both of which are 8-bit and have some additional out-of-band pins used to handle handshaking or for general I/O.

The Cody Computer uses the 65C22's port A to scan the keyboard and joysticks. The keyboard and joystick ports are all combined into the same matrix, consisting of five columns and eight rows. The last two of the eight matrix rows are the two joystick ports, with all other rows part of the keyboard itself.

To cut down on pin counts, the CD4051 one-of-eight analog switch (U8) is used to assist in scanning rows.

Three output lines from the 65C22 are used to select one of eight outputs on the CD4051. This specific use of the CD4051 goes back to the Oric computer.

> The use of the CD4051 as a keyboard scanning aid is explained as part of Garth Wilson's *Circuit Potpourri*. His entire *Wilson Mines Company* website is a vital resource for those new to the 65C02, with his *6502 Primer* required reading for anyone embarking on their own 65C02 computer design.

Both the keyboard rows and keyboard columns are connected to the actual keyboard by the keyboard connector (J7). Each column is connected to a pull-up resistor (R9 through R13) so that, by default, a key that is *not* pressed will register as a logic 1. When a row is scanned, the selected row is pulled low by the CD4051, with all others left disconnected in a high-impedance state. In this situation, when a key is pressed, it completes the circuit to ground, resulting in a logic 0 for the pressed key.

The joystick ports, which reside on the main board, work in a similar fashion. Both joystick ports are male DB9 connectors (J8 and J9) that support a subset of the Atari joystick pinout common to the 8-bit era. Each port has the standard connections for up, down, left, right, and fire button wired as the keys for a keyboard row, while the ground pin for each port is wired as one of the rows on the CD4051's outputs. To scan a joystick, one selects the row just as for a keyboard, then reads the joystick pins.

One minor difference is that the joystick pins have diodes (D1 through D10) connected to them to avoid ghosting, a phenomenon where simultaneous keypresses can result in erroneous data. We don't worry about this for the keyboard itself, as there are a very limited number of valid multiple-key combinations and ghosting will not be a problem for those. However, for the joysticks, where vigorous action and many multiple presses can be expected, we need to directly deal with the ghosting issue.

The remainder of the 65C22's I/O pins are connected to the expansion port (J6). All eight I/O pins from 65C22 port B are routed there and can be used as general-purpose pins in most situations. The CB1 and CB2 pins can be used as handshake pins for communication with compatible devices, but also feature a shift-register mode that will likely be more useful for most applications. While not connected to the 65C22, the Propeller's second UART has its transmit and receive pins routed to the expansion port as well.

The CA1 and CA2 handshake pins, not used with port A, are used to check whether a Cody Computer cartridge has been connected to the expansion slot. CA1 is tied high via a 10 kilohm resistor (R14), but will be pulled down during the cartridge-check routine if CA1 and CA2 are actually tied together by a cartridge in the slot. In all other cases, CA1 will remain at a high logic level and not trigger anything.

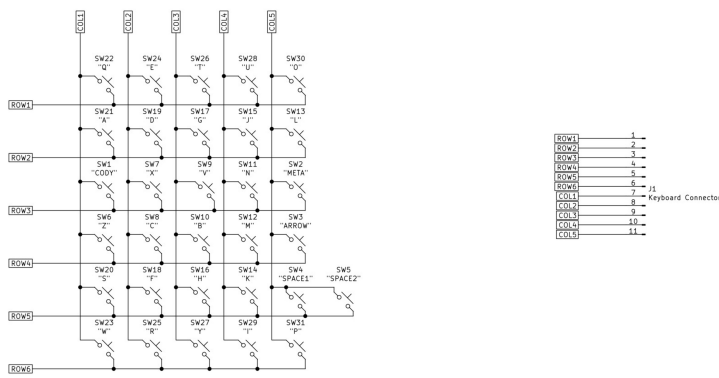
In the event a cartridge is detected, the value of PB4 is examined to determine whether the cartridge uses two-byte or three-byte addressing. Following that, PB0 through PB3 are used to read the contents of the

cartridge into memory over a lowest-common-denominator SPI protocol for memories.

KEYBOARD

The Cody Computer's keyboard exists as a separate schematic and printed circuit board. It contains 29 keys and a spacebar. The physical layout of the keys differs significantly from the electrical layout, with the keyboard itself arranged in a very compact QWERTY layout. The keyboard also uses a nonstandard spacing to keep the size down.

Three of the keys—the Cody, Meta, and Arrow keys—are special keys used to select other characters, change caps lock, and delete or enter text. Two switches are actually combined into the spacebar, one on each side of the spacebar' keycap. This solution was actually easier than designing a nonstandard spacebar stabilizer.



Schematic with keyboard matrix and connector.

The keyboard matrix consists of 31 Cherry MX or compatible switches (SW1 through SW31) arranged

into an electrical matrix of five columns and six rows. The spacebar uses two switches (SW4 and SW5) placed on either end of the spacebar; from the standpoint of the keyboard matrix they're more or less the same switch. The matrix is wired to the keyboard connector (J1) and is connected to the main board via a cable.

No diodes are added to the keyboard to prevent ghosting. Instead the Cody Computer is designed so that no more than two keys would need to be pressed simultaneously at any time, thereby avoiding ghosting issues; at least three simultaneous presses would be necessary to produce ghosting.

Note that this means the keyboard is a poor choice for arcade games or similar. In those situations the joystick ports are the more proper input device. As mentioned above, these do have diodes to prevent ghosting and allow the joysticks to be read without problems under heavy use.

PROPELLER FIRMWARE

As mentioned earlier, much of the Cody Computer's functionality comes from the Propeller chip. That functionality is specified within the Propeller's firmware. Mostly written in the Propeller's own assembly language, PASM, with minor use of SPIN, the Propeller's interpreted high-level language, it should be at least somewhat understandable to anyone with experience in low-level programming. The files are released under the GPL and are available with the rest of the Cody Computer's files.

The Propeller actually contains eight small processors, each of which can run its own small program of up to 512 instructions. While this may not sound like a lot, it suffices for most low-level programming, and larger programs can be written in SPIN or executed using various low-level workarounds.

For our purposes, we rely on the fast, deterministic execution of Propeller assembly language code, so those don't apply to us. Instead, we break up the necessary parts of the Cody Computer's emulated hardware into small programs, then start them up on individual cogs, letting them run until the computer is shut off.

The firmware is split up into five files:

- The **cody_computer.spin** file contains startup code and drives the circuit.
- The **cody_uart.spin** file contains code for two emulated serial UARTs.
- The **cody_audio.spin** file contains a rough emulation of the SID sound chip.
- The **cody_video.spin** file contains code for NTSC color video generation.
- The **cody_line.spin** file contains per-line rendering code used for video.

Each file is heavily commented but we'll do a brief review of each one here in the book. If you're new to the Propeller you may want to find a reference for PASM and SPIN from the Parallax website, especially if you're going to be following through in the original source files.

CODY_COMPUTER.SPIN

The **cody_computer.spin** file contains the main startup code for the entire Cody Computer, both Propeller and 65C02 code, and acts as the overall driver for the rest of the system. Everything else that happens in the Cody Computer directly or indirectly happens because of the contents of this file.

In its **DAT** section it declares the memory regions that will be visible to the 65C02 bus. One region is a 16-kilobyte area containing zeroes, used for the emulated 16-kilobyte RAM. Following that is an 8-kilobyte area that contains the contents of the **cody.bin** file, the 65C02 firmware that contains the Cody Computer's code and BASIC interpreter.

```
DAT

memory

    long 0[4096]    ' 16K shared RAM starting at 65C02 address $A000
    long            ' 8K ROM (BASIC, character set) starting at 65C02 address $E000
    FILE "cody.bin"
```

Declarations for shared memory mapped into the 65C02's address space.

The actual startup code is written in SPIN, the Propeller's interpreted language, and is contained in the **start** method. The Propeller contains a copy of the SPIN interpreter, and once it starts up, it calls this routine and starts interpreting the code. From there, control is passed to us. Our code starts the audio, UART, and video cogs of the code, then uses the Propeller's **coginit** function to replace the code in the current cog with the driver code under **cogmain**.

```
PUB start

    audio.start(@memory)
    uart.start(@memory)
    video.start(@memory)

    waitcnt(cnt + 10000)
    coginit(0, @cogmain, @memory)
```

The Cody Computer's startup sequence as written in SPIN.

The rest of the file is written in PASM. When control is passed to **cogmain**, the assembly language entry point, it sets up some of the Propeller's I/O pins and does some quick memory calculations to speed up the code later. After that, it emits a reset pulse to start the 65C02 by calling the **emit_reset** routine.

```
cogmain         mov     memory_ptr, PAR

                ' adjust ROM cutoff location with start address of memory
                add     BOUNDARY_ROM, memory_ptr

                ' configure the IO pins used for 6502 and bus signals
                mov     OUTA, INIT_OUTA
                mov     DIRA, INIT_DIRA

                ' run 65C02 reset sequence of 10 clocks with reset high
                call    #emit_reset

                ' dummy read to align our code with hub access windows
                ' before commencing the main loop driving the 6502
                rdbyte  data, addr
```

The entry point in Propeller PASM.

```
emit_reset
                ' begin with reset high and emit 20 clock cycles
                or      OUTA, MASK_RES
                mov     count, #20
:loop
                ' clock low
                andn    OUTA, MASK_PHI
                mov     temp, cnt
                add     temp, #40
                waitcnt temp, temp

                ' clock high
                or      OUTA, MASK_PHI
                mov     temp, cnt
                add     temp, #40
                waitcnt temp, temp

                ' bring reset low after 10 cycles
                cmp     count, #10      wz
```

```
if_z            andn    OUTA, MASK_RES

                ' next clock cycle
                djnz    count, #:loop

                ' bring reset high when done
                or      OUTA, MASK_RES

emit_reset_ret  ret
```

*The Propeller **emit_reset** routine that starts the 65C02.*

Once done, the program enters the main loop, under **cycle**, where it handles all the operations necessary to drive the circuit for a single cycle. It brings the PHI2 clock signal for the 65C02 low, reads the address on the bus to determine what device to use, selects the appropriate device, and brings the PHI2 clock signal high. Checks are also performed to determine if the Propeller itself is the device being selected, which will happen if the address is at the top 24 kilobytes of memory.

Because this main loop also produces the main clock for the rest of the circuit, it must be exact with its timing. In order to achieve that, we perform what is called a hub operation, syncing the code up with the rest of the Propeller, before entering the main loop. After that, we go through and add up the time required for each instruction, including other hub operations, to ensure that a stable 1 megahertz clock results from the code regardless of any path taken through it.

```
cycle
                ' Begin the main 6502 loop by bringing phi low to end
                ' the previous cycle, then reset the OUTA/DIRA config.
                '
                ' Once we've reset our state to begin the next cycle,
                ' read from the inputs and determine what we need to do.

                andn    OUTA, MASK_PHI          ' phi2 low at start (1)
                mov     DIRA, INIT_DIRA         ' reset IO direction (2)
                mov     OUTA, INIT_OUTA         ' reset output state (3)
                mov     addr, INA               ' read address (4)
                and     addr, MASK_WORD         ' mask address bits (5)
```

```
                cmp     addr, BOUNDARY_RAM  wc  ' test address for prop memory (6)
if_nc           jmp     #internal               ' prop internal memory path (7)
                cmp     addr, BOUNDARY_VIA  wc  ' test address for sram or io (8)
if_nc           andn    OUTA, MASK_IOSEL        ' io selected (9)
if_c            andn    OUTA, MASK_RAMSEL       ' otherwise ram selected (10)
                or      OUTA, MASK_ABE_PHI      ' address bus off, phi2 high (11)
                nop                             ' wait (12)
                nop                             ' wait (13)
                nop                             ' wait (14)
                nop                             ' wait (15)
                nop                             ' wait (16)
                nop                             ' wait (17)
                nop                             ' wait (18)
                nop                             ' wait (19)
                jmp     #cycle                  ' next loop (20)
```

The main loop that drives the rest of the circuit, including the 65C02.

In this latter case, it also has to read data from the 65C02's bus into the Propeller or write data from the Propeller onto the 65C02's bus. In these cases, control jumps to the **internal** branch, and on to the labelled **read** or **write** sections depending on the exact operation. It also performs a special check to see if the 65C02 is attempting to write to the top 8 kilobytes, and if so, ignore it. This emulates a traditional ROM at the top of the address space by making it unwritable.

```
                ' Accessing hub memory so capture the address while the
                ' address bus is enabled, then process as read or write.

internal        sub     addr, BOUNDARY_RAM      ' adjust address for prop (8)
                add     addr, memory_ptr        ' adjust with base pointer (9)
                test    MASK_RWB, INA       wz  ' read or write op? (10)
                or      OUTA, MASK_ABE_PHI      ' address bus off, phi2 high (11)
if_z            jmp     #write                  ' write operation (12)

                ' Performing a read operation from the hub memory, so we
                ' have to read from memory during the hub window and put
                ' the data on the data bus (note that the pin direction
                ' also has to be changed to actually put the data on the
                ' 6502 bus).

read            nop                             ' wait (13)
                nop                             ' wait (14)
                rdbyte  data, addr              ' read byte (15, 16)
                or      OUTA, data              ' set output data (17)
                or      DIRA, MASK_LOBYTE       ' enable outputs (18)
                nop                             ' wait (19)
                jmp     #cycle                  ' next loop (20)

                ' Performing a write operation, so we need to get the
                ' data from the 6502 data bus and write it to hub ram
                ' during our hub window.

write           mov     data, INA               ' get input data (13)
                cmp     addr, BOUNDARY_ROM  wc  ' test for non-writeable ROM area (14)
if_c            wrbyte  data, addr              ' write input data (15, 16)
                nop                             ' wait (17)
```

```
                nop                             ' wait (18)
                nop                             ' wait (19)
                jmp     #cycle                  ' next loop (20)
```

The paths taken when the Propeller's memory is accessed by the 65C02.

CODY_UART.SPIN

The Cody Computer contains two UART devices used for serial communication. However, both are implemented purely in software inside the Propeller and are exposed to the 65C02 through shared memory in the Propeller. Each UART uses ring buffers in memory for transmitted and received information, a technique very common in serial communications.

Both are defined in the same file and run in the same cog, with coroutines used to interleave the running code for both UARTs. The Propeller has a special machine language instruction, **jmpret**, that performs a jump while updating a return address, making it well-suited for implementing coroutines.

The **cody_uart.spin** file contains a **start** method that's called by the main program to launch the UART cog. Passed along as a parameter is the base of the shared memory area in the Propeller. Because the UART will talk to the rest of the circuit using addresses in shared memory it needs to know where the shared memory begins within the Propeller. From there, the **start** method, written in SPIN, eventually launches a new cog with assembly code using **cognew**.

```
PUB start(mem_ptr)
```

```
cognew(@cogmain, mem_ptr)
```

*The UART **start** entry point written in SPIN.*

The assembly code, starting under **cogmain**, begins by adjusting a variety of memory pointers with the base address of shared memory. This way the adjustment only occurs once at the start of the program rather than each time it reads or writes a value. After that, it configures the Propeller I/O pins used for serial I/O and does initial setup for the coroutines.

Two variables, **uart1_task** and **uart2_task**, store the current positions within the **uart1** and **uart2** routines (the names are just a convention and could have been anything). The UARTs are implemented within the **uart1** and **uart2** routines, which are identical except that they use different local variables and I/O pins.

```
cogmain
                ' Adjust all pointers using hub memory base address
                mov     temp, #18
:adjust         add     UART1_CONTROL, PAR
                add     :adjust, INC_DEST
                djnz    temp, #:adjust

                ' Initialize serial port pins
                or      DIRA, UART1_TX_PIN
                or      OUTA, UART1_TX_PIN

                or      DIRA, UART2_TX_PIN
                or      OUTA, UART2_TX_PIN

                ' Prepare to run as coroutines
                mov     uart2_task, #uart2
```

*The PASM **cogmain** that sets up the UARTs.*

Control initially begins with **uart1**. On each loop it begins by checking if the UART is enabled, and if so, reading the baud rate from the UART's configuration settings. Once read the baud rate is converted to a time value using the **BAUD_RATE_TABLE**. If the UART

is disabled then it does some cleanup at the end and loops until the UART is reenabled.

```
uart1
                ' Yield to other UART
                jmpret  uart1_task, uart2_task

                ' Is the UART running?
                rdbyte  temp, UART1_COMMAND
                test    temp, #$01              wz
if_z            jmp     #:disabled

                ' Mark UART1 status bit as high
                or      uart1_state, #$40
                wrbyte  uart1_state, UART1_STATUS

                 ' Get the baud rate for the UART
                rdbyte  temp, UART1_CONTROL
                and     temp, #$0F
                add     temp, #BAUD_RATE_TABLE
                movs    :baud, temp
                nop
:baud           mov     uart1_delta, 0-0
```

*The initial lines of the **UART1** routine.*

```
BAUD_RATE_TABLE long    0                               ' 0x0
                long    (80_000_000 /    50)            ' 0x1
                long    (80_000_000 /    75)            ' 0x2
                long    (80_000_000 /   110)            ' 0x3

                long    (80_000_000 /   135)            ' 0x4
                long    (80_000_000 /   150)            ' 0x5
                long    (80_000_000 /   300)            ' 0x6
                long    (80_000_000 /   600)            ' 0x7

                long    (80_000_000 /  1200)            ' 0x8
                long    (80_000_000 /  1800)            ' 0x9
                long    (80_000_000 /  2400)            ' 0xA
                long    (80_000_000 /  3600)            ' 0xB

                long    (80_000_000 /  4800)            ' 0xC
                long    (80_000_000 /  7200)            ' 0xD
                long    (80_000_000 /  9600)            ' 0xE
                long    (80_000_000 / 19200)            ' 0xF
```

***BAUD_RATE_TABLE** lookup table that maps register values to time delays.*

When the UART is running, it checks to see if any bits remain to be sent, and if so, whether enough time has elapsed since the last bit to send another one. If there are no more bits to send, it checks to see if there are more bytes to send in the transmit ring buffer and brings in the next byte. Using that byte, it constructs

the entire frame for the byte, including a start bit and a stop bit, and saves it so that the code can send it out a bit at a time.

```
                ' Yield to other UART
:transmit       jmpret  uart1_task, uart2_task

                ' Do we have bits left to send?
                cmp     uart1_tx_left, #0       wz
if_nz           jmp     #:send

                ' Get buffer head and tail positions
                rdbyte  head, UART1_TXHEAD
                and     head, #$07

                rdbyte  tail, UART1_TXTAIL
                and     tail, #$07

                ' Is the buffer empty? If so, move on
                cmp     head, tail              wz
if_z            jmp     #:receive

                ' Mark transmit bit as high
                or      uart1_state, #$10
                wrbyte  uart1_state, UART1_STATUS

                ' Read the next item from memory
                mov     temp, UART1_TXBUF
                add     temp, tail
                rdbyte  uart1_tx_bits, temp

                ' Update the tail position
                add     tail, #1
                and     tail, #$07
                wrbyte  tail, UART1_TXTAIL

                ' Construct frame for bits (start and stop bit)
                or      uart1_tx_bits, #$100
                shl     uart1_tx_bits, #2
                or      uart1_tx_bits, #1

                ' Calculate first timestamp to send a bit
                mov     uart1_tx_time, CNT
                add     uart1_tx_time, uart1_delta

                ' Loop 11 times (high, start, data, stop)
                mov     uart1_tx_left, #11

:send
                ' Yield to other UART
                jmpret  uart1_task, uart2_task

                 ' See if it's time to send data
                mov     temp, uart1_tx_time
                sub     temp, CNT
                cmps    temp, #0                wc
if_nc           jmp     #:receive

                ' Shift out the next bit
                shr     uart1_tx_bits, #1       wc
                muxc    OUTA, UART1_TX_PIN
                add     uart1_tx_time, uart1_delta

                ' Decrement bit count by one
                sub     uart1_tx_left, #1       wz

                ' Clear transmit bit when done with the byte
if_z            andn    uart1_state, #$10
```

```
if_z            wrbyte  uart1_state, UART1_STATUS
```

Code path taken when transmitting bits.

The receive process is generally the same, checking to see if a bit needs to be read, and if no receive operation is in progress, whether a start bit has been encountered. As bytes are read, they are added to the receive buffer similar to how they're consumed from the transmit buffer. Throughout the process, the code updates various local variables, status bits in shared memory, and at key points jumps back to the other UART so both run concurrently.

```
:receive
                ' Yield to other UART
                jmpret  uart1_task, uart2_task

                ' Are we already receiving a byte?
                cmp     uart1_rx_left, #0       wz
if_nz           jmp     #:recv

                ' Do we have a start bit? (start bits are 0)
                test    UART1_RX_PIN, INA       wz
if_nz           jmp     #uart1

                ' Mark receive bit as high
                or      uart1_state, #$08
                wrbyte  uart1_state, UART1_STATUS

                ' Calculate first timestamp to receive a bit
                mov     uart1_rx_time, uart1_delta
                shr     uart1_rx_time, #1
                add     uart1_rx_time, uart1_delta
                add     uart1_rx_time, CNT

                ' Clear out bits
                mov     uart1_rx_bits, #0

                ' Nine bits to receive (includes the stop bit)
                mov     uart1_rx_left, #9

:recv
                ' Yield to other UART
                jmpret  uart1_task, uart2_task

                ' See if it's time to receive data
                mov     temp, uart1_rx_time
                sub     temp, CNT
                cmps    temp, #0                wc
if_nc           jmp     #uart1

                ' Read the next bit
                test    UART1_RX_PIN, INA       wz
if_nz           or      uart1_rx_bits, BIT_9
                shr     uart1_rx_bits, #1
                add     uart1_rx_time, uart1_delta

                ' Decrement number of bits left to read
                sub     uart1_rx_left, #1       wz
if_nz           jmp     #uart1
```

```
                ' Test stop bit was set (framing error?)
                test    uart1_rx_bits, BIT_8    wz
if_z            jmp     #:frame

                ' Yield to other UART
                jmpret  uart1_task, uart2_task

                ' Get buffer head and tail positions
                rdbyte  head, UART1_RXHEAD
                and     head, #$07

                rdbyte  tail, UART1_RXTAIL
                and     tail, #$07

                ' Check for overflow (can only store 7 items)
                mov     temp, tail
                sub     temp, head
                abs     temp, temp
                cmp     temp, #7                wc
if_nc           jmp     #:overflow

                ' Calculate address for next byte in buffer
                mov     temp, UART1_RXBUF
                add     temp, head

                ' Calculate new buffer head position
                add     head, #1
                and     head, #$07

                ' Update buffer and position
                wrbyte  uart1_rx_bits, temp
                wrbyte  head, UART1_RXHEAD

                ' Clear receive bit at end of byte
                andn    uart1_state, #$08
                wrbyte  uart1_state, UART1_STATUS

                jmp     #uart1
```

Code path taken when receiving bits.

Some special paths exist for when errors are detected or the UART is disabled. During error conditions an appropriate bit is set in the status register to indicate the nature of the problem. When the UART is disabled, it is also an opportunity to reset the UART for the next time it's used. Some of the internal variables in particular need cleared out.

```
:frame
                ' Set frame bit (bit 1) on status register
                or      uart1_state, #$02
                wrbyte  uart1_state, UART1_STATUS

                jmp     #uart1

:overflow
                ' Set overflow bit (bit 2) on status register
                or      uart1_state, #$04
                wrbyte  uart1_state, UART1_STATUS

                jmp     #uart1
```

```
:disabled
                ' Clear any pending bits in the system
                mov     uart1_rx_left, #0
                mov     uart1_tx_left, #0
                mov     uart1_state, #0

                ' Clear out any registers managed by the UART
                wrbyte  ZERO, UART1_RXHEAD
                wrbyte  ZERO, UART1_TXTAIL
                wrbyte  ZERO, UART1_STATUS

                jmp     #uart1
```

Special paths used when an error is found or the UART is turned off.

The UART code, while not as complex as other portions of the firmware, still contains a variety of concepts that may be new. For a simple example of implementing a single UART on the Propeller, one might start with the *Full Duplex Serial* example by Propeller designer Chip Gracey posted on the Propeller OBEX. The code uses coroutines to toggle between the receive and transmit paths for a single software UART and lacks many of the complicating factors in the Cody Computer UART code. It is very useful as a learning aid or reference.

CODY_AUDIO.SPIN

The Cody Computer uses a simplified version of the Commodore SID chip for its sound generation. Instead of a real SID, one of the cogs in the Propeller is devoted to generating audio output, and a portion of the shared memory is set aside to mimic the SID's registers.

The Cody Computer's implementation is in most respects a port of the GPL-licensed *MOS6581 SID Emulator Arduino Library* by Christoph Haberer and Mario Patino. In addition to rewriting the library in PASM from the original code, many changes were

made to support the Propeller's similar but not identical output-pin hardware. Yet other changes had to be made to integrate it into the Cody Computer as a whole.

> *SIDcog* is a more complete emulation for the Propeller created by Johannes Ahlebrand and later enhanced by Ada Gottensträter. The emulator is excellent but some timing and space requirements on top of our already busy Propeller made it a challenge to integrate. Nonetheless, the possibility exists for an interested reader.

As with the other portions of Propeller firmware, the implementation is written using PASM. A small SPIN method, **start**, launches the cog with PASM code starting at **cogmain**, similar to the UART. The PASM code begins by adjusting some internal memory pointers relative to the shared memory region, sets up an output pin for the audio signal, and initializes some variables used for the main loop.

One important step is setting the cog's **ctra** register to enable what's known as the duty single-ended mode on the pin we've selected for audio output. Each cog has an internal counter that can be used for a variety of operations. In this case we're using the counter to quickly generate an on-or-off output with a varying duty cycle faster than we could possibly do in software alone.

The external circuitry discussed in the previous section smoothes this out into an analog waveform despite the actual output being a digital on-or-off.

Once enabled, we can put an output value into the matching **frqa** register to control the duty cycle, and by extension, control the sound that comes out of the Propeller.

```
cogmain
                ' Calculate actual position of registers
                add     REGS_BASE, PAR
                add     OSC3_PTR, PAR
                add     ENV3_PTR, PAR

                ' Configure output for sound
                mov     dira, INIT_DIRA
                mov     ctra, INIT_CTRA

                ' Configure timing
                mov     time, cnt
                add     time, WAIT_TIME

                mov     output, #0
```

*The **cogmain** entry point in PASM.*

From there the code enters **main_loop**, which begins by waiting until enough time has elapsed to run the main loop again. The Cody Computer's SID has a sample rate of 16 kilohertz, which means that we want the main loop to run 16000 times per second. The Propeller's clock ticks 80 million times per second, so after dividing the Propeller's clock by the desired sampling rate, we realize we need to run the loop once every 5000 ticks. And because each Propeller instruction takes four of its clock cycles, we calculate that our loop has to run in no more than 1250 instructions.

When the loop is ready to run again, it begins by updating the white noise generator. White noise was one of the waveform options for the real SID, so we also need a source for it here. Our implementation follows the Arduino SID emulator mentioned previously, so it uses a linear feedback shift register implemented in software.

In a linear feedback shift register, a sequence of bits is generated by storing a seed value, extracting certain bits, shifting the original value, A portion of the result can be extracted and used for other purposes (such as noise), with other portions of the result fed back in to repeat the proces on the next iteration.

Once the noise value is updated, the code runs the **:voice_loop** three times, one for each voice. Subroutines for processing the voice are called from within the loop. Once done, the voices are combined and output by calling the **make_output** routine.

```
main_loop

:loop
                ' Wait for next cycle
                waitcnt time, WAIT_TIME

                ' Update noise
                mov     temp, noise
                and     temp, #$1
                neg     temp, temp
                and     temp, NOISE_BITS
                shr     noise, #1
                xor     noise, temp
                and     noise, MASK_16

                ' Start at beginning of internal voice states on each main loop
                movs    readvar, #state1
                movd    savevar, #state1

                ' Start at beginning of registers on each main loop
                mov     register_ptr, REGS_BASE

                ' Three voices to process
                mov     voice_count, #3
:voice_loop
                call    #voice_begin
                call    #make_wave
                call    #make_envelope
                call    #make_waveform
                call    #voice_end

                djnz    voice_count, #:voice_loop

                ' Combine into a single output
                call    #make_output

                ' Repeat the main loop
                jmp     #:loop
```

The Cody SID's main loop.

The **voice_begin** routine prepares everything for generating a voice. Because the Propeller's assembly

language has very limited support for indirect addressing, the code has to copy variables for each voice to temporary variables used within the loop. When it's done processing the current voice, it copies them back at the end.

Once that initial per-voice setup is completed, the code performs special checks for the SID's sync and test bits. If the sync bit is enabled the code syncs the current voice's phase with another voice, but if the test bit is set, the code resets most of the current voice's internal state.

```
voice_begin
                ' Read the registers for a single voice into COG memory
                movd    :readreg, #voice_freq_l
                mov     count, #7
:readreg        rdbyte  0-0, register_ptr
                add     :readreg, INC_DEST
                add     register_ptr, #1
                djnz    count, #:readreg

                ' Copy the internal states for the current voice into temp vars
                movd    readvar, #state
                mov     count, #7
readvar         mov     0-0, 0-0
                add     readvar, INC_BOTH
                djnz    count, #readvar

                ' Sync voice if the other voice indicates it's time to sync,
                ' test if sync bit is on AND it's time to sync (order is)
                ' reversed because we're counting down).
                cmp     voice_count, #2                 wc,wz
if_nc           movd    :testsync, #sync3     ' Voice 1 uses voice 3
if_z            movd    :testsync, #sync1     ' Voice 2 uses voice 1
if_c            movd    :testsync, #sync2     ' Voice 3 uses voice 2
                nop
:testsync       test    0-0, voice_control              wz
if_nz           mov     phase, #0

                ' Reset voice if the test bit is on
                test    voice_control, #$08             wz
if_nz           mov     phase, #0
if_nz           mov     amplitude, #0
if_nz           mov     state, #0

voice_begin_ret ret
```

*The **voice_begin** routine called at the start of each loop.*

The next part of the loop is in **make_wave**, which generates the wave portion of the current voice. The wave, which is the raw triangle, sawtooth, pulse, or

noise signal, is shaped by an envelope in a later step. However, it comprises the base upon which the rest of the sound is built.

To begin, it takes the frequency specified for the voice, using that to update an internal phase counter. This counter is used to determine what portion of a particular wave to generate based on how much time has gone by. Different code paths, **:triangle**, **:sawtooth**, **:pulse**, and **:noise**, exist for each supported wave type.

```
make_wave
                ' Combine frequency into 16 bit number
                ' Shift by 2 because frequency * 4000 / 16 KHz sample rate
                mov     freq_coefficient, voice_freq_h
                shl     freq_coefficient, #8
                or      freq_coefficient, voice_freq_l
                shr     freq_coefficient, #2

                 ' Calculate next phase
                mov     temp_phase, phase
                add     temp_phase, freq_coefficient

                ' If we overflowed, set our internal sync bit to apply later
                testn   temp_phase, MASK_16                 wz
                muxnz   sync, #$02

                ' Limit phase calculation to 16 bits internally
                and     temp_phase, MASK_16

:triangle
                ' Triangle waveform?
                test    voice_control, #$10                 wz
if_z            jmp     #:sawtooth

                ' Time to invert? (Goes up half the time, then down half the time)
                ' Double the value to make sure it covers the full range
                mov     wave, phase
                test    wave, BIT_15                        wz
                shl     wave, #1
if_nz           xor     wave, MASK_16
                and     wave, MASK_16
                jmp     #:done

:sawtooth
                ' Sawtooth waveform?
                test    voice_control, #$20                 wz
if_z            jmp     #:pulse

                mov     wave, phase
                jmp     #:done

:pulse
                ' Pulse waveform?
                test    voice_control, #$40                 wz
if_z            jmp     #:noise

                mov     temp, voice_pulse_h
                shl     temp, #8
                or      temp, voice_pulse_l
                shl     temp, #4
                and     temp, MASK_16
```

```
                cmp     phase, temp                         wc
if_c            mov     wave, MASK_16
if_nc           mov     wave, #0

                jmp     #:done

:noise
                ' Noise waveform?
                test    voice_control, #$80                 wz
if_z            jmp     #:done

                mov     temp, phase
                xor     temp, temp_phase
                test    temp, PHASEBIT_NOISE                wz
if_nz           mov     temp, noise
if_nz           and     temp, MASK_16
if_nz           mov     wave, temp

:done
                ' Update phase for the current voice (limited to unsigned 16 bits)
                mov     phase, temp_phase

                ' Ensure wave only has 16 bits of resolution
                and     wave, MASK_16

make_wave_ret   ret
```

*The **make_wave** routine generates a voice's underlying sound.*

After generating the wave, **make_envelope** runs to generate the ADSR envelope. ADSR, short for Attack-Decay-Sustain-Release, is a key concept in synthesis, specifying the "envelope" for a sound. The attack specifies how long it takes to reach a maximum volume once a sound is started, while the decay specifies how long it takes for the sound to go back down to its sustain level after peaking. The release specifies how long the sound takes to fade out once the sound is shut off.

For the Cody Computer's SID, a voice is turned on when its gate bit is set, so the code checks it to see if the sound has started. It also refers to an internal state variable to determine where it is in the ADSR envelope. As part of the calculations, precomputed tables **ATTACK_RATES**, **DECAY_RATES**, and **SUSTAIN_LEVELS** are used to look up how much to add or subtract during the attack and decay or what

volume level to hold at during sustain. At the end of the calculation, it has generated the envelope that will be combined with the previously-generated wave.

```
make_envelope
                ' Is gate bit set? (playing a note?)
                test    voice_control, #$01                wz
if_z            jmp     #:release

:attack
                ' Gate bit set, but are we on attack or decay state?
                tjnz    state, #:decay

                ' Increment amplitude with attack value from table
                movs    :addattack, #ATTACK_RATES
                mov     temp, voice_attack_decay
                shr     temp, #4
                add     :addattack, temp
                nop
:addattack      add     amplitude, 0-0

                ' Did we reach the maximum value (end of attack portion?)
                cmp     amplitude, MAXLEVEL                wc
if_c            jmp     #:done

                ' Cap at maximum amplitude, enter decay phase
                mov     amplitude, MAXLEVEL
                mov     state, #1

                jmp     #:done

:decay
                ' Look up the matching sustain value from the table
                mov     temp, voice_sustain_release
                shr     temp, #4
                add     temp, #SUSTAIN_LEVELS
                movs    :getsustain, temp
                nop
:getsustain     mov     level_sustain, 0-0

                ' Did we reach that sustain level?
                cmp     level_sustain, amplitude           wc
if_nc           jmp     #:done

                ' Subtract the current decay value from our amplitude,
                ' but don't let our amplitude fall below zero
                mov     temp, voice_attack_decay
                and     temp, #$0F
                add     temp, #DECAY_RATES
                movs    :subdecay, temp
                nop
:subdecay       sub     amplitude, 0-0                     wc
if_c            mov     amplitude, #0

                ' Limit amplitude from falling below sustain level
                min     amplitude, level_sustain

                jmp     #:done

:release
                ' Gate bit is off so not in attack state
                mov     state, #0

                ' Have we reached zero amplitude?
                tjz     amplitude, #:done

                ' Subtract the current decay value from our amplitude,
                ' but don't let our amplitude fall below zero
                mov     temp, voice_sustain_release
                and     temp, #$0F
                add     temp, #DECAY_RATES
```

```
                movs    :subrelease, temp
                nop
:subrelease     sub     amplitude, 0-0                  wc
if_c            mov     amplitude, #0

                ' Scale envelope from 24 to 16 bits resolution
:done           mov     envelope, amplitude
                shr     envelope, #8

make_envelope_ret   ret
```

*The **make_envelope** routine generates a voice's ADSR envelope.*

The **make_waveform** combines both of these values together. It first checks if ring modulation is enabled and applies it if so. Ring modulation is a technique where one voice is combined with the output of another to generate unique sounds, and the SID chip implemented a special case of ring modulation that we attempt to mimic.

Once ring modulation has been applied, the wave value and the envelope value are multiplied together to get the final waveform value for this voice in the loop.

```
make_waveform
                ' We'll be multiplying the wave value by the envelope value
                mov     x, wave

:ring
                ' Ring modulation bit?
                test    voice_control, #$04             wz
if_z            jmp     #:done

                ' For "ring modulation" we invert the wave based on another's phase
                ' (Order is reversed because we're counting down)
                cmp     voice_count, #2                 wc,wz
if_nc           movd    :testphase, #phase3   ' Voice 1 uses voice 3
if_z            movd    :testphase, #phase1   ' Voice 2 uses voice 1
if_c            movd    :testphase, #phase2   ' Voice 3 uses voice 2
                nop
:testphase      test    0-0, BIT_15                     wz
if_nz           xor     x, MASK_16

:done
                ' Multiply the wave by the envelope
                mov     y, envelope
                call    #multiply

                ' Scale result down from 32 to 16 bits
                shr     y, #16
                mov     output, y
```

```
make_waveform_ret   ret
```

*The **make_waveform** routine that combines the wave and envelope.*

After that, there are some bookkeeping tasks to perform, such as copying the temporary variables back to their original locations. At the end of each voice loop the **voice_end** routine is called. This handles any final processing or cleanup at the end of a voice. As a practical matter, it's responsible for copying the temporary voice variables back to their permanent locations. Just as **voice_begin** copied them in at the beginning of the loop, this routine does the reverse when the voice has come to an end. Once that's done, the **voice_loop** repeats for each remaining voice.

```
voice_end
                movs    savevar, #state
                mov     count, #7
savevar         mov     0-0, 0-0
                add     savevar, INC_BOTH
                djnz    count, #savevar

voice_end_ret   ret
```

*The **voice_end** routine saves the values of temporary variables.*

Once output values for all three voices have been generated, **make_output** puts them together. All three voices are combined together (with the possible exception of voice 3, which can be shut off), multiplied by the current global volume, and scaled to the range supported by the audio output circuitry. Once the combined output value is written to the Propeller's **frqa** register, the rest is handled by hardware, and a pulse-width-modulated signal is output to the audio circuitry on the board.

A few other operations are also performed, such as updating a couple of shared memory locations with some internal values from voice 3. The SID did this and the values were often used for random numbers or special audio effects, so here we do something similar to keep the spirit alive. Other features such as filters haven't been implemented.

```
make_output
                ' Read the filter registers
                movd    :readfilt, #filter_cutoff_l
                mov     count, #4
:readfilt       rdbyte  0-0, register_ptr
                add     :readfilt, INC_DEST
                add     register_ptr, #1
                djnz    count, #:readfilt

                ' Combine outputs (voice 3 is a special case)
                mov     x, output1
                add     x, output2

                ' Voice 3 is skipped if bit is set
                test    filter_mode_volume, #$80            wz
if_z            add     x, output3

                ' Apply volume setting
                mov     y, filter_mode_volume
                and     y, #$0F
                call    #multiply
                shr     y, #4

                ' Scale output value to Propeller PWM value
                mov     output, y
                sub     output, BIT_15
                shl     output, #11
                add     output, BIT_31
                mov     frqa, output

                ' Write high byte of voice 3 oscillator waveform
                mov     temp, wave3
                shr     temp, #8
                wrbyte  temp, OSC3_PTR

                ' Write high byte of voice 3 envelope
                mov     temp, envelope3
                shr     temp, #8
                wrbyte  temp, ENV3_PTR

make_output_ret ret
```

*The **make_output** routine merges all three voices into one output.*

Note that because the Propeller has no built-in multiplication hardware, all multiplication is done in software. While this sounds somewhat primitive, it also helps keep the Propeller the simple and

deterministic system it is from a hardware standpoint. We have a routine, **multiply**, that was taken from Appendix B of the Propeller's reference manual and multiplies two 16-bit numbers together. This suffices for our purposes and doesn't take that many cycles.

```
multiply
                shl     x, #16      ' Get multiplicand into x high bits
                mov     t, #16      ' Ready for 16 multiplier bits
                shr     y,  #1  wc  ' Get initial multiplier bit into c
:loop

if_c            add     y, x    wc  ' If carry set, add multiplicand into product
                rcr     y, #1   wc  ' Get next multiplier bit into c, shift product

                ' Loop until done
                djnz    t,  #:loop

multiply_ret    ret
```

The software multiply routine.

CODY_VIDEO.SPIN

A significant portion of the Propeller's capabilities are used to implement the Cody Computer's Video Interface Device (VID). Five of the chip's eight cogs are devoted to some aspect of video generation, and the chip's custom video generation hardware is utilized to generate an NTSC-compatible analog video signal. The Propeller contains circuitry that can generate all the relevant portions of a video signal, including blanking and color sync pulses.

Using the circuitry involves configuring a counter to the appropriate output rate for the video signal, then using the **waitvid** instruction to pass color and pixel data to it. As a special case, we can actually call **waitvid** with four colors and four pixels, making it possible to use any of the Propeller's colors anywhere on the screen.

Software-based NTSC video generation from first principles isn't something that can be easily summed up in a few paragraphs. One level of detail would be to discuss the characteristics of the signal itself, while another would be to discuss in depth the Propeller's unique capacities for analog video output. In this book it's assumed that all of that just works, instead focusing on how these capabilities are used at a high level to implement the Cody Computer's video interface device.

For a more in-depth discussion of video generation without all the extra complications caused by the Cody Computer, one might start with Eric Ball's *NTSC and PAL Driver Templates* available on the Propeller OBEX. Portions of that code were foundational to the Cody Computer's own video code, and it's an excellent walkthrough of analog video generation in the context of the Propeller. I'd also recommend reading any of the relevant Propeller forum postings.

Video generation on the Cody Computer begins in the **cody_video.spin** file. Memory is reserved for four scanline "mailboxes" in the **scanlines** variable, which will later be used to communicate with the cogs responsible for rendering the video lines. A lookup table, **COLOR_TABLE**, is also defined to map Cody Computer color codes to their Propeller equivalents. On startup, the **start** SPIN method sets up the scanline mailboxes, then launches the video signal generation cog with PASM code starting at **cogmain**.

```
PUB start(mem_ptr) | index

    ' Start up the scanline renderer cogs
    repeat index from 0 to 3

        ' Set up each mailbox
        mailboxes[index * 100 + 0] := index
```

```
        mailboxes[index * 100 + 1] := mem_ptr
        mailboxes[index * 100 + 2] := @COLOR_TABLE
        mailboxes[index * 100 + 3] := 0

        ' Launch the corresponding cog
        line_renderer.start(@mailboxes + index * 400)

    ' Launch the video cog itself once the scanline cogs are running
    launch_cog(mem_ptr, @COLOR_TABLE, @mailboxes+0, @mailboxes+400, @mailboxes+800, @mailboxes

PRI launch_cog(mem_ptr, ctable_ptr, scan1_ptr, scan2_ptr, scan3_ptr, scan4_ptr)

    cognew(@cogmain, @mem_ptr)
```

SPIN portion of the video startup code.

The **cogmain** code first calls the **load_params** routine to read in the locations of shared memory and the four mailboxes for the scanline cogs. It also uses the shared memory base address to calculate the positions of some of the video registers used by the video signal generator.

```
cogmain
                call    #load_params
                call    #init_video
:loop
                call    #frame
                jmp     #:loop
```

The main loop for the NTSC video generation code.

After that, **cogmain** calls the **init_video** routine to set up **vcfg** for the video mode and what bank of output pins to use, **ctra** for the counter mode, and **frqa** for the video frequency. The video output pins are also set as outputs in **dira**, as without doing so, the video will not actually be emitted on the pins selected in **vcfg**. (For more detail on these Propeller registers, refer to the Propeller reference manual in particular.)

```
init_video
                ' Sets up the parameters for video generation
                mov     vcfg, ivcfg

                ' Internal PLL mode, PLLA = 16 * colorburst frequency
                mov     ctra, ictra

                ' 2 * colorburst frequency
                mov     frqa, ifrqa
```

```
                ' Configure selected video pins as outputs
                or      dira, idira

init_video_ret  ret
```

Initialization of the Propeller's video registers and output pins.

After that the **load_params** routine is responsible for retrieving the parameters passed from SPIN. The previously-mentioned **launch_cog** routine in SPIN used the SPIN interpreter's stack to hold multiple parameters, passing the address of the first one to the newly-created cog running the code. The PASM code sequentially reads parameters from the SPIN stack beginning at that starting address. It also adjusts a few addresses along the way.

```
load_params
                mov     params_ptr, PAR

                rdlong  memory_ptr, params_ptr
                add     params_ptr, #4

                rdlong  lookup_ptr, params_ptr
                add     params_ptr, #4

                rdlong  temp, params_ptr
                add     toggle1_ptr, temp
                add     buffer1_ptr, temp
                add     buffer5_ptr, temp
                add     params_ptr, #4

                rdlong  temp, params_ptr
                add     toggle2_ptr, temp
                add     buffer2_ptr, temp
                add     buffer6_ptr, temp
                add     params_ptr, #4

                rdlong  temp, params_ptr
                add     toggle3_ptr, temp
                add     buffer3_ptr, temp
                add     buffer7_ptr, temp
                add     params_ptr, #4

                rdlong  temp, params_ptr
                add     toggle4_ptr, temp
                add     buffer4_ptr, temp
                add     buffer8_ptr, temp
                add     params_ptr, #4

                mov     vblreg_ptr, memory_ptr
                add     vblreg_ptr, VBLANK_REG_OFFSET

                mov     ctlreg_ptr, memory_ptr
                add     ctlreg_ptr, CONTROL_REG_OFFSET

                mov     colreg_ptr, memory_ptr
                add     colreg_ptr, COLOR_REG_OFFSET
```

```
load_params_ret ret
```

PASM for loading parameters from the SPIN ***launch_cog*** *routine.*

From this point the video generator code enters an infinite loop, outputting video signals for NTSC frames one after the other. The scanline generators are set to the start of a new frame, a vertical sync pulse is generated by calling **vertical_sync**, the video control and border color registers are read, blank lines are generated by calling **ntsc_blank_lines**, and at last the scanline generators are turned on.

The top border is generated via **top_border**, the drawable screen area via **screen_area**, and the bottom border via a call to **bottom_border**. The vertical blanking register is also updated during this process to indicate when the 65C02 can generally update video memory or registers without fear of collision.

```
frame
                ' Generate NTSC vertical sync
                call    #vertical_sync

                ' Generate NTSC blank lines after vertical sync
                call    #ntsc_blank_lines

                ' Set vertical blanking indicator to zero (not safe to update)
                wrbyte  ZERO, vblreg_ptr

                ' Read current video control register from memory
                rdbyte  control, ctlreg_ptr

                ' Read current border color and convert to Propeller color
                rdbyte  border, colreg_ptr
                shl     border, #1
                add     border, lookup_ptr
                rdword  border, border

                ' Reset scanline generators back to beginning
                wrlong  TOGGLE_FRAME, toggle1_ptr
                wrlong  TOGGLE_FRAME, toggle2_ptr
                wrlong  TOGGLE_FRAME, toggle3_ptr
                wrlong  TOGGLE_FRAME, toggle4_ptr

                ' Draw part of the screen top border
                call    #top_border

                ' Turn scanline generators on
                wrlong  TOGGLE_LINE1, toggle1_ptr
                wrlong  TOGGLE_LINE1, toggle2_ptr
                wrlong  TOGGLE_LINE1, toggle3_ptr
                wrlong  TOGGLE_LINE1, toggle4_ptr
```

```
                ' Draw the rest of the screen top border
                call    #top_border

                ' Draw the screen (and horizontal borders)
                call    #screen_area

                ' Set vertical blanking indicator to 1 (safe to update)
                wrbyte  ONE, vblreg_ptr

                ' Draw screen bottom border
                call    #bottom_border

frame_ret       ret
```

*The **frame** routine generates a single TV frame.*

Most of the work occurs in the **screen_area** routine where the actual screen is drawn. A quick check is performed to see if vertical scrolling is enabled, and if so, reduce the size of the vertical area by one row. After that, it loops for each row on the screen, toggling the scanline renderers and generating a video signal for each rendered scanline by calling the **scanline** routine.

The scanline renderers are called in order, giving each renderer the equivalent of four scanlines to render the next line. To make this possible, each scanline renderer has two buffers so that it can be rendering a new line while the previous line is being sent out.

```
screen_area
                ' Generate additional top border lines if vertical scroll enabled
                test    control, #%00000010 wz
if_nz           call    #scroll_border

                ' 25 groups of lines to generate (assuming no vertical scrolling)
                mov     numline, #25

                ' Adjust number of lines if vertical scrolling enabled
                test    control, #%00000010 wz
if_nz           sub     numline, #1

                ' Render scanlines behind the scenes as we generate NTSC signals
:loop           wrlong  TOGGLE_LINE2, toggle1_ptr
                mov     source, buffer1_ptr
                call    #scanline

                wrlong  TOGGLE_LINE2, toggle2_ptr
                mov     source, buffer2_ptr
                call    #scanline

                wrlong  TOGGLE_LINE2, toggle3_ptr
                mov     source, buffer3_ptr
                call    #scanline
```

```
                wrlong  TOGGLE_LINE2, toggle4_ptr
                mov     source, buffer4_ptr
                call    #scanline

                wrlong  TOGGLE_LINE1, toggle1_ptr
                mov     source, buffer5_ptr
                call    #scanline

                wrlong  TOGGLE_LINE1, toggle2_ptr
                mov     source, buffer6_ptr
                call    #scanline

                wrlong  TOGGLE_LINE1, toggle3_ptr
                mov     source, buffer7_ptr
                call    #scanline

                wrlong  TOGGLE_LINE1, toggle4_ptr
                mov     source, buffer8_ptr
                call    #scanline

                ' Continue on to next group of 8 lines
                djnz    numline, #:loop

                ' Generate additional bottom border lines if vertical scroll enabled
                test    control, #%00000010 wz
if_nz           call    #scroll_border

screen_area_ret ret
```

PASM routine for generating the drawable screen area.

The **scanline** routine actually generates the video signal for a single line in the drawable screen area. It generates the horizontal sync at the start of the line, followed by the NTSC signal's back porch. Following that, a total of 40 **waitvid**s are performed in a loop.

The exact output and timing depends on the current video mode. For the lower-resolution multicolor mode, each **waitvid** consists of four pixels read from a scanline renderer's inactive buffer. In the high-resolution mode, eight pixels are read and the data format in the scanline buffer is somewhat different. While the output timing differs (each high-resolution pixel takes half the time of a lower-resolution pixel) most of the same Propeller configuration values are used for both outputs.

Once all the pixels have been output, the NTSC signal's front porch is generated to end the line. The **horizontal_sync**, **front_porch**, and **back_porch**

routines are used to help with some of the above. When drawing the line, some checks are also made for situations where the display is disabled or horizontal scrolling is enabled. If these conditions exist, adjustments are made to the output.

```
scanline        ' Start signal with horizontal sync and NTSC back porch
                call    #horizontal_sync
                call    #back_porch

                ' Test hires or lowres mode based on bit in control register
                test    control, #%00100000 wz
if_nz           jmp     #:hires

:lores          ' By default we have 40 waitvids (160 pixels / 4 pixels per waitvid)
                mov     count, #40
                mov     VSCL, vsclactv

                ' If horizontal scrolling, draw fewer pixels and a bigger border
                test    control, #%00000100 wz
if_nz           waitvid border, #0
if_nz           sub     count, #2

                ' Adjust pointer for offscreen scratch area in scanline buffer
                add     source, #12

:lores_loop     ' Read the next four pixels from the scanline buffer
                rdlong  colors, source

                ' If the display is enabled, draw the pixels from the buffer
                ' If the display is shut off, draw the border color instead
                test    control, #%00000001 wz
if_z            waitvid colors, lores_pixels
if_nz           waitvid border, #0

                ' Go on to the next four pixels
                add     source, #4
                djnz    count, #:lores_loop

                ' If horizontal scrolling, draw a bigger border
                test    control, #%00000100 wz
if_nz           waitvid border, #0

                ' Done generating NTSC video for the multicolor mode
                jmp     #:done

:hires         ' We always have 40 waitvids (320 pixels / 8 pixels per waitvid)
                mov     count, #40
                mov     VSCL, vsclactvhi

:hires_loop     ' Read the next eight pixels from the scanline buffer
                rdword  hires_pixels, source
                add     source, #2

                ' Read the colors for the 8x8 tile from the scanline buffer
                rdword  colors, source
                add     source, #2

                ' If the display is enabled, draw the pixels from the buffer
                ' If the display is shut off, draw the border color instead
                test    control, #%00000001 wz
if_z            waitvid colors, hires_pixels
if_nz           waitvid border, #0

                ' Go on to the next eight pixels
                djnz    count, #:hires_loop

                ' Generate the NTSC front porch before completing
:done           call    #front_porch
```

```
scanline_ret    ret
```

PASM routine for generating a single NTSC scanline.

CODY_LINE.SPIN

The last component of the Cody Computer's video firmware are the scanline renderers. Rendering the contents of a single 160 pixel line, both background tiles and sprites, takes quite a bit of time (from the standpoint of a video signal). In fact, it takes longer than a single scanline just to generate its contents. The 320 pixel lines in the high-resolution mode don't have the overhead of rendering sprites, but they do have more pixels they have to generate because of the higher resolution.

To work around this problem we set up other cogs as renderers that store pixels to a buffer in memory. When it's time to generate the signal containing the line, the video cog reads the pre-rendered pixels and generates the corresponding signal.

The video generator cog launches a total of four scanline renderer cogs, each running the code from **cody_line.spin**. The video generator calls a short SPIN method, **start**, passing the pointer to the start of the mailbox used to communicate with the renderer. The renderer, in turn, starts running PASM code starting at **cogmain**. Some initial setup code runs to get data from the mailbox and calculate some pointer addresses.

```
cogmain
                ' Load parameters and calculate pointers from the scanline structure
                ' using the calculated offsets within the mailbox memory area
                add     renderer_index, PAR
                add     memory_ptr, PAR
                add     lookup_ptr, PAR
                add     toggle_ptr, PAR
                add     buffer1_ptr, PAR
```

```
                add     buffer2_ptr, PAR

                rdlong  renderer_index, renderer_index
                rdlong  memory_ptr, memory_ptr
                rdlong  lookup_ptr, lookup_ptr

                ' Adjust our offsets into shared memory now that we know where it is
                add     VIDCTL_REGS_OFFSET, memory_ptr
                add     SPRITE_REGS_OFFSET, memory_ptr

                add     ROWEFF_CNTL_OFFSET, memory_ptr
                add     ROWEFF_DATA_OFFSET, memory_ptr
```

*The **cogmain** PASM code called when starting a scanline renderer.*

From there the scanline renderer enters the **:frame_loop** for the start of a new frame. It waits until the mailbox shows a new frame has started (because the video cog has toggled it), then does some initial setup for the new frame. The video registers are read from shared memory.

The code then waits for another toggle to render a line, running the **:line_loop** for a total of 50 times. Because the drawable screen has 200 lines and there are four cogs rendering the screen contents, each cog is responsible for 50 lines.

For each line, any row effects are applied first via **apply_row_effects**, followed by decoding the video register values in **decode_registers**. Finally the scanline's contents are rendered by calling other routines. In the low-resolution multicolor mode the **render_chars_lo** and **render_sprites** routines are responsible for rendering the scanline. In the high-resolution mode the **render_chars_hi** mode is called instead. The **:line_loop** repeats until no more lines remain on the current frame, each time waiting for a toggle from the main video cog.

```
:frame_loop
                ' Wait for the TOGGLE_FRAME value to begin the next frame
                rdlong  toggle, toggle_ptr
                cmp     toggle, TOGGLE_FRAME    wz
```

```
if_nz           jmp     #:frame_loop
                wrlong  TOGGLE_EMPTY, toggle_ptr

                ' Read in the video registers at the start of a new frame
                mov     video_register_ptr, VIDCTL_REGS_OFFSET

                rdbyte  blankreg, video_register_ptr
                add     video_register_ptr, #1

                rdbyte  controlreg, video_register_ptr
                add     video_register_ptr, #1

                rdbyte  colorreg, video_register_ptr
                add     video_register_ptr, #1

                rdbyte  basereg, video_register_ptr
                add     video_register_ptr, #1

                rdbyte  scrollreg, video_register_ptr
                add     video_register_ptr, #1

                rdbyte  screenreg, video_register_ptr
                add     video_register_ptr, #1

                rdbyte  spritereg, video_register_ptr
                add     video_register_ptr, #1

                ' Reset row effects at the beginning of each frame
                mov     roweff_remaining, #32
                mov     roweff_cntl_ptr, ROWEFF_CNTL_OFFSET
                mov     roweff_data_ptr, ROWEFF_DATA_OFFSET

                ' Render each line
                mov     lines_remaining, #50
                mov     curr_scanline, renderer_index

:line_loop
                ' Wait for a TOGGLE_LINE1 or TOGGLE_LINE2 value to begin the next line
                rdlong  toggle, toggle_ptr

                cmp     toggle, TOGGLE_EMPTY    wz
if_z            jmp     #:line_loop

                cmp     toggle, TOGGLE_FRAME    wz
if_z            jmp     #:frame_loop

                ' Clear toggle value once we begin a new line
                wrlong  TOGGLE_EMPTY, toggle_ptr

                ' Select the destination buffer for this scanline
                cmp     toggle, TOGGLE_LINE1    wz
if_z            mov     buffer_ptr, buffer1_ptr

                cmp     toggle, TOGGLE_LINE2    wz
if_z            mov     buffer_ptr, buffer2_ptr

                ' Read any row effects that may be pending for this scanline
                call    #apply_row_effects

                ' Decode the video registers (including any raster changes)
                call    #decode_registers

                ' Render the scanline to the buffer
                test    controlreg, #%001000000             wz
if_z            call    #render_chars_lo

                test    controlreg, #%001000000             wz
if_nz           call    #render_chars_hi

                test    controlreg, #%001000000             wz
if_z            call    #render_sprites

                ' Go to the next line
                add     curr_scanline, #4
                djnz    lines_remaining, #:line_loop

                ' Begin a new frame
```

```
                jmp     #:frame_loop
```

Code executed in the frame and line loops.

The **render_chars_lo** and **render_chars_hi** routines are responsible for rendering the characters on the screen. These two routines have some similarities but also have differences resulting from the different behavior in the multicolor and high-resolution graphics modes. The multicolor mode is intended for general purpose programming including games, so additional features like scrolling are supported. The high-resolution mode is intended for more serious uses and focuses on rendering a larger number of pixels without other complications.

First let's discuss the more common (and default) low-resolution multicolor mode that begins with the **render_chars_lo** routine. It makes some adjustments for vertical and horizontal scrolling, if enabled, and then proceeds to render the current scanline. Calculations use the **SCREEN_OFFSET_TABLE** to determine the screen and color memory locations corresponding to the current scanline.

Looping over each of the 40 columns in the scanline in the **:char_loop**, the screen and color information are read from shared memory. The colors for that screen location are converted from Cody Computer color codes to Propeller NTSC color codes using the previously-mentioned **COLOR_TABLE** and merged with the current global colors for the screen. If in character graphics mode, the matching character line for the character in screen memory is also read and the byte pattern returned. In bitmap graphics mode, the corresponding four-pixel byte within screen

memory is returned instead, but the operation is very similar otherwise. From there the **:pixel_loop** renders the actual pixels into the scanline buffer before continuing on to the next character.

```
render_chars_lo
                ' Set up the output pointer taking into account the left "margin" for sprites
                mov     dest_ptr, buffer_ptr
                add     dest_ptr, #12

                ' Update the output start position to account for horizontal scrolling
                test    controlreg, #%00000100  wz
if_nz           sub     dest_ptr, scrollh

                ' Update the source line position to account for vertical scrolling
                mov     adjustv, #0
                test    controlreg, #%00000010  wz
if_nz           mov     adjustv, scrollv

                ' Precalculate the current offset for each character based on the scanline
                mov     char_offset_y, curr_scanline
                add     char_offset_y, adjustv
                and     char_offset_y, #%0111

                ' Determine offset in the screen and color memory based on the current row
                mov     screen_memory_offset, curr_scanline
                add     screen_memory_offset, adjustv
                shr     screen_memory_offset, #3
                add     screen_memory_offset, #SCREEN_OFFSET_TABLE
                movs    :load_offset, screen_memory_offset
                nop

:load_offset    mov     screen_memory_offset, 0_0

                ' Calculate the locations in color and screen memory using the offset above
                mov     curr_colors_ptr, colmem_ptr
                add     curr_colors_ptr, screen_memory_offset

                test    controlreg, #%00010000 wz
if_z            mov     curr_screen_adv, #1
if_nz           mov     curr_screen_adv, #8
if_nz           shl     screen_memory_offset, #3

                mov     curr_screen_ptr, scrmem_ptr
                add     curr_screen_ptr, screen_memory_offset

                mov     chars_remaining, #40

:char_loop      rdbyte  color_data, curr_colors_ptr

                shl     color_data, #1
                add     color_data, lookup_ptr

                rdword  color_data, color_data
                or      color_data, common_screen_colors

                add     curr_colors_ptr, #1

                test    controlreg, #%00010000              wz
if_nz           mov     source_ptr, curr_screen_ptr
if_z            rdbyte  source_ptr, curr_screen_ptr
if_z            shl     source_ptr, #3
if_z            add     source_ptr, chrset_ptr
                add     source_ptr, char_offset_y
                add     dest_ptr, #3
                rdbyte  pixel_data, source_ptr

                mov     pixels_remaining, #4

:pixel_loop     mov     temp, pixel_data
```

```
                and     temp, #%11

                shl     temp, #3
                ror     color_data, temp

                wrbyte  color_data, dest_ptr

                sub     dest_ptr, #1
                rol     color_data, temp

                shr     pixel_data, #2
                djnz    pixels_remaining, #:pixel_loop

                add     dest_ptr, #5
                add     curr_screen_ptr, curr_screen_adv

                djnz    chars_remaining, #:char_loop

render_chars_lo_ret    ret
```

*The **render_chars_lo** routine renders a line's background characters in the low-resolution multicolor mode.*

The **render_sprites** routine is largely the same, except that it renders the sprites over the now-drawn background characters. It begins by determining the sprite register bank to read from based on the current value in a shared memory register, positioning a pointer at the start of the appropriate bank. The sprite bank registers have the needed coordinates, color, and sprite pointer information, so it's important to start in the right place.

Once prepared, it loops over each of the eight possible sprites in the **:sprite_loop**, verifying that they're actually on screen and adjusting for scrolling if necessary. It also looks up the sprite's unique colors and finds their Propeller equivalents in the same way used for the character colors. When it's ready to draw the sprite, it goes into the **:byte_loop** to draw each of the sprite's three data bytes, with the individual pixels being drawn in the **:pixel_loop**.

Some key differences exist between these loops and the corresponding loops for drawing character pixels,

with one of the main differences being that sprites can have transparent pixels.

```
render_sprites

                ' Start sprite pointer at the beginning of the current bank
                mov     curr_sprite_ptr, spritereg
                and     curr_sprite_ptr, #$70
                shl     curr_sprite_ptr, #1
                add     curr_sprite_ptr, SPRITE_REGS_OFFSET

                ' Draw the 8 sprites we have in this bank
                mov     sprites_remaining, #8
:sprite_loop
                ' Read in and check the sprite x coordinate is within bounds
                rdbyte  sprite_x, curr_sprite_ptr
                add     curr_sprite_ptr, #1

                cmp     sprite_x, #0        wz
if_z            jmp     #:next_sprite

                cmp     sprite_x, #172      wc
if_nc           jmp     #:next_sprite

                ' Read in and check the sprite y coordinate is within bounds
                rdbyte  sprite_y, curr_sprite_ptr
                add     curr_sprite_ptr, #1

                ' Adjust sprite y position by subtracting top margin amount
                sub     sprite_y, #21
                sub     sprite_y, curr_scanline
                neg     sprite_y, sprite_y

                cmp     sprite_y, #0        wc
if_c            jmp     #:next_sprite

                cmp     sprite_y, #21       wc
if_nc           jmp     #:next_sprite

                ' Read in the sprite colors and combine them with the common sprite color
                rdbyte  sprite_colors, curr_sprite_ptr
                shl     sprite_colors, #1
                add     sprite_colors, lookup_ptr
                rdword  sprite_colors, sprite_colors
                shl     sprite_colors, #8
                or      sprite_colors, common_sprite_colors
                add     curr_sprite_ptr, #1

                ' Read in the sprite pointer and adjust for the current scanline
                rdbyte  sprite_ptr, curr_sprite_ptr
                add     sprite_y, #SPRITE_OFFSET_TABLE
                movs    :load_offset, sprite_y
                shl     sprite_ptr, #6
:load_offset    add     sprite_ptr, 0_0
                add     sprite_ptr, memory_ptr

                ' Set up our destination buffer
                mov     dest_ptr, buffer_ptr
                add     dest_ptr, sprite_x

                ' Draw each byte remaining in this scanline
                mov     chars_remaining, #3
:byte_loop
                ' Read in the sprite data
                rdbyte  pixel_data, sprite_ptr
                add     sprite_ptr, #1

                ' Draw each pixel in this byte (in reverse order)
                add     dest_ptr, #3
                mov     pixels_remaining, #4
:pixel_loop
                ' Move the current color into position for drawing
                mov     temp, pixel_data
```

```
                and     temp, #%11
                shl     temp, #3
                ror     sprite_colors, temp

                ' Draw the pixel if non-transparent
                cmp     temp, #0                wz
if_nz           wrbyte  sprite_colors, dest_ptr
                sub     dest_ptr, #1

                ' Prepare for the next pixel
                rol     sprite_colors, temp
                shr     pixel_data, #2

                djnz    pixels_remaining, #:pixel_loop

                add     dest_ptr, #5
                djnz    chars_remaining, #:byte_loop

:next_sprite
                ' Increment the sprite register pointer to the start of the next sprite
                andn    curr_sprite_ptr, #3
                add     curr_sprite_ptr, #4

                ' Loop if we have more sprites remaining
                djnz    sprites_remaining, #:sprite_loop

render_sprites_ret      ret
```

*The **render_sprites** routine handles eight sprites per line.*

When the high-resolution mode is enabled, the **render_chars_hi** routine is called instead of **render_chars_lo** and **render_sprites**. This routine is similar but not exactly the same as **render_chars_lo**. It calculates certain offsets and locations in memory, but doesn't support scrolling so those additional calculations are not performed. Additionally, it also switches between reading the pixel data from characters or more sequentally for a bitmap.

However, the actual data rendered into the scanline buffer is quite different. Instead of rendering four different Propeller color values, this mode renders eight pixels and their colors. Because the Propeller will be running in its four-color output mode, we also expand the single-bit pixel values into two-bit values for the Propeller hardware. Because we have eight pixels and each pixel is expanded to two bits, this means a total of two bytes is required. This pixel data

is followed by two bytes containing the Propeller colors for that group of eight pixels.

```
render_chars_hi
                ' Set up the output pointer
                mov     dest_ptr, buffer_ptr

                ' Precalculate the current offset for each character based on the scanline
                mov     char_offset_y, curr_scanline
                add     char_offset_y, adjustv
                and     char_offset_y, #%%0111

                ' Determine offset in the screen and color memory based on the current row
                mov     screen_memory_offset, curr_scanline
                shr     screen_memory_offset, #3
                add     screen_memory_offset, #SCREEN_OFFSET_TABLE
                movs    :load_offset, screen_memory_offset
                nop

:load_offset    mov     screen_memory_offset, 0_0

                ' Calculate the locations in color and screen memory using the offset above
                mov     curr_colors_ptr, colmem_ptr
                add     curr_colors_ptr, screen_memory_offset

                test    controlreg, #%00010000 wz
if_z            mov     curr_screen_adv, #1
if_nz           mov     curr_screen_adv, #8
if_nz           shl     screen_memory_offset, #3

                mov     curr_screen_ptr, scrmem_ptr
                add     curr_screen_ptr, screen_memory_offset

                mov     chars_remaining, #40

                ' Read the per-character color information and look up the Propeller colors
:char_loop      rdbyte  color_data, curr_colors_ptr

                shl     color_data, #1
                add     color_data, lookup_ptr

                rdword  color_data, color_data
                add     curr_colors_ptr, #1

                ' Fetch the character bits
                test    controlreg, #%00010000              wz
if_nz           mov     source_ptr, curr_screen_ptr
if_z            rdbyte  source_ptr, curr_screen_ptr
if_z            shl     source_ptr, #3
if_z            add     source_ptr, chrset_ptr
                add     source_ptr, char_offset_y
                rdbyte  temp, source_ptr

                mov     pixel_data, #0
                mov     pixels_remaining, #8

:pixel_loop     ' Shift out the character data bit at a time
                shr     temp, #1                            wc
if_c            or      pixel_data, #%01

                ' Next output bit
                shl     pixel_data, #2
                djnz    pixels_remaining, #:pixel_loop

                ' Compensate for last shift right inside the loop
                shr     pixel_data, #2

                ' Write the pixel and color information to the buffer
                wrword  pixel_data, dest_ptr
                add     dest_ptr, #2

                wrword  color_data, dest_ptr
                add     dest_ptr, #2
```

```
                ' Move to the next character
                add     curr_screen_ptr, curr_screen_adv
                djnz    chars_remaining, #:char_loop

render_chars_hi_ret    ret
```

*The **render_chars_hi** routine renders a high-resolution scanline.*

The **decode_registers** routine is a helper called during the main loop to decode the video register values from local variables. These contain some information, including Cody Computer color codes, that need translated to their Propeller NTSC equivalents. Others contain data that's packed into a single register, such as nibble values that map to memory locations within the shared memory. This routine helps with unpacking and keeps the related logic in one place.

```
decode_registers

                ' Calculate color memory position
                mov     colmem_ptr, colorreg
                shr     colmem_ptr, #4
                shl     colmem_ptr, #10
                add     colmem_ptr, memory_ptr

                ' Calculate screen memory position
                mov     scrmem_ptr, basereg
                shr     scrmem_ptr, #4
                shl     scrmem_ptr, #10
                add     scrmem_ptr, memory_ptr

                ' Calculate character set position
                mov     chrset_ptr, basereg
                and     chrset_ptr, #$7
                shl     chrset_ptr, #11
                add     chrset_ptr, memory_ptr

                ' Calculate scroll values
                mov     scrollv, scrollreg
                and     scrollv, #%00000111

                mov     scrollh, scrollreg
                shr     scrollh, #4
                and     scrollh, #%00000011

                ' Calculate shared screen colors
                mov     common_screen_colors, screenreg
                shl     common_screen_colors, #1
                add     common_screen_colors, lookup_ptr
                rdword  common_screen_colors, common_screen_colors
                shl     common_screen_colors, #16

                ' Calculate shared sprite colors
                mov     common_sprite_colors, spritereg
                shl     common_sprite_colors, #1
```

```
                add     common_sprite_colors, lookup_ptr
                rdword  common_sprite_colors, common_sprite_colors
                shl     common_sprite_colors, #24

decode_registers_ret  ret
```

*The **decode_registers** routine that unpacks register values.*

The **apply_row_effects** routine is related. On old computers, it was common to use special tricks, such as switching out video data, on certain lines to extend the hardware's graphics abilities. The Cody Computer has a similar feature where data can be overridden on each of the 25 rows on the screen. Rather than setting interrupts and changing register data, additional registers let you specify override values and where to apply them.

This routine handles those situations by checking to see if the row effects are enabled, and if so, whether they need to be applied based on the current scanline. The scanline is divided by 8 to determine what row on the screen is being drawn, and then any of the video data that has been overridden is updated in the local variables. By doing this in the main loop prior to decoding the registers, any overridden values are automatically used when rendering the scanline. The code also remembers the last row effect processed and begins from there on the next scanline. This optimization exists to reserve more cycles for actually rendering the scanline's contents.

```
apply_row_effects

                ' Quick check to ensure that row effects are enabled
                test    controlreg, #%00001000          wz
if_z            jmp     #apply_row_effects_ret

                ' Calculate what row we're currently on for row effects
                mov     roweff_row, curr_scanline
                shr     roweff_row, #3

                ' Check if we have more row effects to look at
```

```
:loop           cmp     roweff_remaining, #0            wz
if_z            jmp     #apply_row_effects_ret

                ' Read the control/data bytes and extract the row number
:cont           rdbyte  roweff_cntl_byte, roweff_cntl_ptr

                mov     temp, roweff_cntl_byte
                and     temp, #%000011111

                rdbyte  roweff_data_byte, roweff_data_ptr

                ' Test that this line is applicable for this row
                cmp     temp, roweff_row                wz, wc
if_nz_and_nc    jmp     #apply_row_effects_ret

                ' Apply the replacement for the selected register
                mov     temp, roweff_cntl_byte
                and     temp, #%111000000

                cmp     temp, #%100000000               wz
if_z            mov     basereg, roweff_data_byte

                cmp     temp, #%101000000               wz
if_z            mov     scrollreg, roweff_data_byte

                cmp     temp, #%110000000               wz
if_z            mov     screenreg, roweff_data_byte

                cmp     temp, #%111000000               wz
if_z            mov     spritereg, roweff_data_byte

                ' Move on to the next entry
                add     roweff_cntl_ptr, #1
                add     roweff_data_ptr, #1
                sub     roweff_remaining, #1

                ' Next row effect
                jmp     #:loop

apply_row_effects_ret   ret
```

*The **apply_row_effects** routine replaces old-school raster interrupts.*

3

Software Design

INTRODUCTION

On startup, the Cody Computer boots into Cody BASIC, a BASIC interpreter written from scratch just for the Cody Computer. It allows you to write moderately-complex programs and perform file operations from the BASIC prompt. The BASIC dialect is inspired by Tiny BASIC, a small open-source BASIC dating to the 1970s.

While largely a dialect of Tiny BASIC, Cody BASIC has some additional features typically not present in most Tiny BASIC environments. These include (limited) arrays, strings, and DATA statements. Cody BASIC also uses messages and commands inspired by Commodore BASIC instead of the Tiny BASIC equivalents. Also unlike many Tiny BASIC dialects but similar to the Commodore, the program is not directly interpreted. Rather, the BASIC program is tokenized into small pieces that are executed more quickly at runtime.

We'll cover how to program in Cody BASIC later in the book, but here we'll talk a bit about how it's implemented in 65C02 assembly. The code itself is open source and heavily commented, so we won't go over every single line here. We're more focused on a high-level view of the code, with some detailed analysis of particular subroutines.

Keep in mind that while the actual source file is somewhat long, it produces a mere 6 kilobytes of machine code for the 65C02 (an additional 2 kilobytes contain the character set). The Cody BASIC ROM itself is embedded as data within the Propeller program

mentioned in the previous section, mapped to the very top of the 65C02's memory area.

STARTUP AND INITIALIZATION

When the 65C02 starts, it loads a two-byte address from memory location **$FFFC**, lowest byte first (this is always the case for the 65C02, as it's a little-endian processor). Here we put the address for our **MAIN** routine, responsible for the initial startup. It sets the boundary page of BASIC program memory into **PROGEND** so that it can be overridden by any memory-resident programs later on. After that, it calls **INIT** to initialize most of the hardware and software from the 65C02's side. Finally it performs a check to see if a cartridge is inserted in the expansion slot. If so, it boots from the cartridge. If not, it drops through to the **BASIC** routine we'll discuss in a moment.

```
MAIN      LDA #>PROGMAX         ; Set the top of program memory to the default page
          STA PROGEND

          JSR INIT              ; Run initialization on startup

          JSR CARTCHECK         ; Check for cartridge plugged in
          BEQ BASIC

          STZ IOMODE            ; Cartridge found, load and run binary instead of BASIC
          STZ IOBAUD
          JMP LOADBIN
```

*The Cody BASIC interpreter's **MAIN** routine.*

The **MAIN** routine calls **INIT** to initialize most of the hardware and software from the 65C02's side, including copying the character set into video memory, setting up video registers, and preparing a timer interrupt for timekeeping and keyboard scanning. It also sets up a simple error handling system that allows BASIC interpreter routines to easily signal an

error. Because **INIT** is also used to partially reset the interpreter after running a binary program, some things aren't reset by this routine. In particular, the **PROGEND** zero-page variable is untouched by this routine so that memory-resident programs can adjust it.

```
        STZ VID_SCRL            ; Clear out scroll registers

        STZ VID_CNTL            ; Clear out control register

        LDA #$E7                ; Point the video hardware to default color memory, border co
        STA VID_COLR

        LDA #$95                ; Point the video hardware to the default screen and character
        STA VID_BPTR

        STZ KEYLAST             ; Clear out the major keyboard-related zero page variables
        STZ KEYLOCK
        STZ KEYMODS
        STZ KEYCODE
```

*A small excerpt from the **INIT** routine.*

Different parts of the initialization process run depending on whether a cartridge is connected to the computer or not. If a cartridge is present, most of the initialization process is skipped or not enabled, instead loading and running a binary program from the cartridge. In other situations the Cody BASIC interpreter is launched.

TIMER INTERRUPT

Cody BASIC relies on a timer interrupt to handle keyboard scanning, simple timekeeping, and other periodic tasks. This timer interrupt is generated by the 65C22 VIA chip that also handles most of the computer's I/O operations. The interrupt is configured to run 60 times per second. Most of the setup occurs in the **MAIN** routine, but the interrupt isn't actually started until the BASIC interpreter itself takes control.

```
        LDA #<TIMERISR        ; Set up ISR routine address
        STA ISRPTR+0
        LDA #>TIMERISR
        STA ISRPTR+1

        LDA #<JIF_T1C         ; Set up VIA timer 1 to emit ticks (60 per second)
        STA VIA_T1CL
        LDA #>JIF_T1C
        STA VIA_T1CH

        LDA #$40              ; Set up VIA timer 1 continuous interrupts, no outputs
        STA VIA_ACR

        LDA #$C0              ; Set up VIA timer 1 interrupt
        STA VIA_IER
```

*Setting up the timer interrupt in **MAIN**.*

One level of indirection exists for the timer interrupt's handler. Because the 65C02's interrupt handler is fixed at address **$FFFE** in memory, code in ROM would make it impossible for other programs (such as those written in assembly language) to change the interrupt handler to something different.

To avoid that problem, we put a simple stub, **ISRSTUB**, at the 65C02's interrupt handler address. This jumps to a different address, **ISRPTR**, stored in the zero page and pointing to the actual location of the interrupt service routine. If other code wants to change the interrupt behavior, it needs only change the value of **ISRPTR** to point to its own routine.

```
ISRSTUB   JMP (ISRPTR)
```

*The **ISRSTUB** that jumps to the actual interrupt handler.*

Cody BASIC's interrupt handler or service routine, **TIMERISR**, is responsible for several important functions. First it calls **KEYSCAN** to scan the keyboard matrix. Next it updates the jiffies count stored in **JIFFIES**, a two-byte variable. A jiffy is the time for a

single timer tick, and we keep a count to provide a simple mechanism for determining elapsed time without a full real time clock (this technique was very common in the 8-bit era).

The interrupt handler also provides an important safety function for BASIC programs. When a BASIC program is running, it checks to see if the Cody and Arrow keys are both held down on the keyboard. If both are pressed, the keypresses are interpreted as a break request by the user. Without this functionality, it would be possible to get into a nonterminating BASIC program and be unable to exit without turning the Cody Computer on and off.

```
TIMERISR  PHA               ; Preserve accumulator

          BIT VIA_T1CL      ; Read the 6522 to clear the interrupt

          JSR KEYSCAN       ; Scan keyboard

          INC JIFFIES       ; Increment jiffy count lower byte (after scanning!)
          BNE _TEST

          INC JIFFIES+1     ; Increment jiffy count upper byte on overflow

_TEST     LDA RUNMODE       ; Only allow breaks if we're running a program
          BEQ _DONE

          LDA KEYROW2       ; Check for Cody key on row 2 (and ONLY the Cody key)
          CMP #$1E
          BNE _DONE

          LDA KEYROW3       ; Check for arrow key on row 3 (and ONLY the arrow key)
          CMP #$0F
          BNE _DONE

          JMP RAISE_BRK     ; Break

_DONE     PLA               ; Restore accumulator

          RTI               ; Return from interrupt routine
```

*The **TIMERISR** routine runs for each interrupt.*

KEYBOARD SCANNING

The Cody Computer has a 30-key keyboard set up in a matrix of five columns and six rows. In addition,

two Atari-style joystick ports with five buttons each are mapped as keyboard rows. Cody BASIC scans the keyboard as part of the timer interrupt routine, updating eight bytes in zero page memory (**KEYROW0** through **KEYROW7**) with the current values of the keyboard rows. These values are subsequently used by other routines to handle keyboard or joystick input.

Scanning is handled by the **KEYSCAN** routine. It uses port A on the 65C22 VIA to iterate over the keyboard matrix, with a one-of-eight analog switch used to convert a three-bit number into the current keyboard row to scan. Once a row is selected, the remainder of port A is read, containing the five bits for the columns, and stored in the appropriate **KEYROW** variable. The timer interrupt calls this routine on a regular basis to update the data.

```
KEYSCAN   PHA                   ; Preserve registers
          PHX

          STZ VIA_IORA          ; Start at the first row and first key of the keyboard
          LDX #0

_LOOP     LDA VIA_IORA          ; Get the keys for the current row from the VIA port
          LSR A
          LSR A
          LSR A
          STA KEYROW0,X

          INC VIA_IORA          ; Move on to the next keyboard row
          INX

          CPX #8                ; Do we have any rows remaining to scan?
          BNE _LOOP

          PLX                   ; Restore registers
          PLA

          RTS
```

*The **KEYSCAN** routine that scans the keyboard matrix.*

Converting the raw bits from the matrix into a keyboard code is the responsibility of the **KEYDECODE** routine. There the **KEYROW** values are examined and

converted to a scan code and stored in **KEYCODE**. It also performs a special check to see if the Cody key is pressed, and if so, updates the state of the keyboard modifiers in **KEYMODS**.

```
KEYDECODE PHX                   ; Preserve registers
          PHY

          STZ KEYMODS           ; Reset scan codes and modifiers at start of new scan
          STZ KEYCODE

          LDX #Ø                ; Start at the first row and first key scan code
          LDY #Ø

_ROW      LDA KEYROWØ,X         ; Load the current row's column bits from zero page
          INX

          PHX                   ; Preserve row index

          LDX #5                ; Loop over current row's columns

_COL      INY                   ; Increment the current key number at the start of each new ke

          LSR A                 ; Shift to get the next column bit

          BCS _NEXT             ; If the current column wasn't pressed, just skip to the next

          CPY #KEY_META         ; Is this the META special key?
          BNE _CODY

          PHA                   ; META key is pressed, update current key modifiers
          LDA KEYMODS
          ORA #$2Ø
          STA KEYMODS
          PLA

          BRA _NEXT             ; Continue on to the next column

_CODY     CPY #KEY_CODY         ; Is this the CODY special key?
          BNE _NORM

          PHA                   ; CODY key is pressed, update current key modifiers
          LDA KEYMODS
          ORA #$4Ø
          STA KEYMODS
          PLA

          BRA _NEXT             ; Continue on to the next column

_NORM     PHA                   ; Not a special key so just store it as the current scan code
          TYA
          STA KEYCODE
          PLA

_NEXT     DEX                   ; Move on to the next keyboard column
          BNE _COL

          PLX                   ; Restore current row index

          CPX #6                ; Continue while we have more rows to process
          BNE _ROW

          LDA KEYCODE           ; Update the current key scan code with the modifiers
          ORA KEYMODS
          STA KEYCODE

          PLY                   ; Restore registers
          PLX
```

```
        RTS
```

*The **KEYDECODE** routine produces a key code from the matrix.*

Key scan codes represent an actual button on the keyboard, not a character. The Cody Computer uses CODSCII, a special character set that's just traditional ASCII with the PETSCII graphics symbols appended to it. As a result, character handling is greatly simplified compared to the actual Commodore computers. Unfortunately, we still have to convert scan codes to their ASCII (or more accurately CODSCII) values.

This is handled by the **KEYTOCHR** routine, which accepts a scan code for the keyboard and converts it to an ASCII code. The routine's implementation relies on a lookup table containing the ASCII codes for each scan code. The ASCII codes correspond to the arrangement of keys in the keyboard matrix so that once we have a scan code we can look up the appropriate value. The lookup table also takes into account whether the Cody or Meta keys have been pressed on the keyboard. (Shift status and conversion to lowercase, however, happens elsewhere.)

```
KEYTOCHR  PHX
          DEC A
          TAX
          LDA _LOOKUP,X
          PLX
          RTS

_LOOKUP

.BYTE 'Q', 'E', 'T', 'U', 'O'      ; Key scan code mappings without any modifiers
.BYTE 'A', 'D', 'G', 'J', 'L'
.BYTE $00, 'X', 'V', 'N', $00
.BYTE 'Z', 'C', 'B', 'M', $0A
.BYTE 'S', 'F', 'H', 'K', ' '
.BYTE 'W', 'R', 'Y', 'I', 'P'
.BYTE $00, $00

.BYTE '!', '#', '%', '&', '('      ; Key scan code mappings with META modifier
.BYTE '@', '_', ':', $27, ']'
.BYTE $00, '<', ',', '?', $00
.BYTE '\', '>', '.', '/', $08
.BYTE '=', '+', ';', '[', ' '
```

```
.BYTE '"', '$', '^', '*', ')'
.BYTE $00, $00

.BYTE '1', '3', '5', '7', '9'      ; Key scan code mappings with CODY modifier
.BYTE 'A', 'D', 'G', 'J', 'L'
.BYTE $00, 'X', 'V', 'N', $1B
.BYTE 'Z', 'C', 'B', 'M', $18
.BYTE 'S', 'F', 'H', 'K', ' '
.BYTE '2', '4', '6', '8', '0'
.BYTE $00, $00
```

*The **KEYTOCHR** routine and its lookup table.*

The **KEYDECODE** and **KEYTOCHR** routines are never called as part of the keyboard scanning done in the timer interrupt. Instead, they're called from the **READKBD** routine, which is completely separate. This routine is called when the Cody BASIC interpreter expects line-based input, such as during the REPL loop or in an **INPUT** statement. Each character entered is also echoed to the screen. We'll discuss those routines in detail when we talk about input and output handling.

ERROR HANDLING

As part of the initialization process a simple form of error handling is set up for the BASIC interpreter and its related code. Error handling in Cody BASIC works like a very simple exception handler. On startup the current location in the 65C02's own stack is stored in the **STACKREG** variable for later use.

At runtime, whenever the interpreter encounters an error, one of several error routines are called. The error routine then calls **ERROR** to handle the error, print an error message, and unwind the 65C02 stack. After unrolling the error, it jumps back into the BASIC interpreter's REPL loop.

```
BASIC     TSX                 ; Preserve the stack register for unwinding on error conditior
          STX STACKREG
```

Preserving the stack position to unwind in the event of an error.

Four helper routines exist to save code and provide a consistent interface to raise an error condition. The **RAISE_BRK** routine corresponds to the **ERR_BREAK** error code, **RAISE_SYN** to **ERR_SYNTAX**, **RAISE_LOG** to **ERR_LOGIC**, and **RAISE_SYS** to **ERR_SYSTEM**.

```
RAISE_BRK LDA #ERR_BREAK
          BRA ERROR

RAISE_SYN LDA #ERR_SYNTAX
          BRA ERROR

RAISE_LOG LDA #ERR_LOGIC
          BRA ERROR

RAISE_SYS LDA #ERR_SYSTEM
          BRA ERROR
```

Entry points to the error-handling system in Cody BASIC.

The first error type, **ERR_BREAK** isn't an error in the strictest sense. An error of this type only indicates that the user is attempting to break from the current program by pressing the Cody and Arrow keys simultaneously. In this situation, the error handling process is somewhat abbreviated instead of displaying a full error message.

The other error types largely match the error conditions from the original Tiny BASIC in the 1970s. **ERR_SYNTAX** indicates that a syntax error was encountered in the current program, similar to Tiny BASIC's **WHAT?**. **ERR_LOGIC** indicates that the program was running but didn't make logical sense, similar to Tiny Basic's **HOW?**. The last error,

ERR_SYSTEM, indicates a system problem such as running out of memory caused an error, similar to Tiny BASIC's **SORRY**.

Using the error routines is straightforward. When code determines that an error exists in the program, it performs an unconditional jump to the corresponding routine to raise that particular error. Detecting the error itself (for example, a missing keyword in a statement) is the responsibility of the calling routine. However, once an error routine is called, further error handling will be taken care of automatically.

```
MOD16     LDA NUMTWO          ; See if the low byte of the second argument is nonzero
          BNE _OK

          LDA NUMTWO+1        ; See if the high byte of the second argument is nonzero
          BNE _OK

          JMP RAISE_LOG       ; Raise a logic error for divide by zero
```

*Example from **MOD16** of raising an error on division by zero.*

Once another part of the program has called into the error handlers, control eventually passes to the **ERROR** routine. It unwinds the stack, restores any I/O settings to their screen and keyboard defaults, and finally prints an error message indicating the type of error that occurred. If the error occurred while the program was running, the current line number is appended as in Commodore BASIC. Once completed, the routine jumps to the **REPL** loop, allowing the user to continue to work with the computer.

```
ERROR     LDX STACKREG        ; Unwind the stack
          TXS

          JSR SERIALOFF       ; Turn off serial mode (just in case it was on)

          STZ IOMODE          ; Reset IO mode for all future output
          STZ IOBAUD
```

```
          STZ OBUFLEN         ; Reset output buffer position

          PHA                 ; Preserve the provided error code in the accumulator

          LDA #CHR_NL         ; Ensure error messages begin on a new line
          JSR PUTOUT

          PLA                 ; Restore the error code into the accumulator

          CLC                 ; Calculate the message table index for the provided error
          ADC #MSG_ERRORS

          JSR PUTMSG          ; Print the error

          CMP #MSG_ERRORS     ; "Break" errors don't have the word "error" (just BREAK IN ...)
          BEQ _BREAK

          LDA #MSG_ERROR      ; Print the word "ERROR" for all other errors
          JSR PUTMSG

_BREAK    LDA RUNMODE         ; Are we running a program right now? (otherwise hide line numbe
          CMP #RM_PROGRAM
          BNE _NOLINE

          LDA #MSG_IN         ; Append "IN" to our error message
          JSR PUTMSG

          LDY #1              ; Start at line number position in current line

          LDA (PROGPTR),Y     ; Copy line number low byte
          STA NUMONE

          INY                 ; Next byte

          LDA (PROGPTR),Y     ; Copy line number high byte
          STA NUMONE+1

          JSR TOSTRING        ; Write the line number into the buffer

_NOLINE   LDA #CHR_NL         ; New line after the error message
          JSR PUTOUT

          LDA #CHR_NL         ; Blank line
          JSR PUTOUT

          LDA #MSG_READY      ; Ready message
          JSR PUTMSG

          JSR FLUSH           ; Print the error message

          STZ RUNMODE         ; Reset run mode (REPL mode after errors or breaks)

          CLI                 ; Enable interrupts (in case we came from the interrupt routine)

          JMP REPL            ; Return to the REPL loop
```

*The **ERROR** routine recovers from errors and prints messages.*

STARTING BASIC

Once the required setup is out of the way, it's time to start up BASIC itself. If no cartridge is connected to the computer, the program continues on to boot up BASIC. While the BASIC interpreter is somewhat

complex, the main loop for it isn't that difficult to follow. As mentioned in our discussion of error handling, we keep a copy of the current 65C02 stack position for our error handler when we enter BASIC. Then a short startup message is printed. Finally, interrupts are enabled so that the timer interrupt and keyboard scanning routine will run.

```
BASIC     JSR INIT              ; Re-run BASIC initialization just to be safe

          TSX                   ; Preserve the stack register for unwinding on error condition
          STX STACKREG

          STZ OBUFLEN           ; Move to beginning of the output buffer

          LDA #MSG_GREET        ; Print the welcome message
          JSR PUTMSG
          JSR FLUSH

          LDA #MSG_READY        ; Print the ready message
          JSR PUTMSG
          JSR FLUSH

          CLI                   ; Enable interrupts and drop through to the REPL loop
```

Final steps before entering BASIC.

We then enter a read-eval-print loop (REPL) that lets the user enter text into Cody BASIC. All input is tokenized by the **TOKENIZE** routine and then examined. If a line begins with a number, we insert or delete the line from the program with a call to **ENTERLINE**. If it doesn't begin with a number, we call **EXSTMT** to execute the line as a BASIC statement.

```
REPL      STZ RUNMODE           ; Clear out RUNMODE

          STZ IOMODE            ; Direct all IO to screen and keyboard

          JSR READKBD           ; Read a line of input and advance the screen
          JSR SCREENADV

          JSR TOKENIZE          ; Tokenize the input

          LDA TBUF              ; Line number to add or execute the line immediately?
          CMP #$FF
          BNE _EXEC

          JSR ENTERLINE         ; Enter the line into the program

          BRA REPL              ; Next read-eval-print loop

_EXEC     STZ PROGOFF           ; Start at the beginning of the line

          LDA #>TBUF            ; Use the token buffer as the line we're going to run
          STA PROGPTR
          LDA #>TBUF
          STA PROGPTR+1

          JSR EXSTMT            ; Execute the statement in the token buffer

          STZ OBUFLEN           ; Move to beginning of output buffer

          LDA #MSG_READY        ; Print the ready message after each REPL operation
          JSR PUTMSG
          JSR FLUSH

          BRA REPL              ; Next read-eval-print loop
```

The implementation of Cody BASIC's read-eval-print loop.

STARTING A CARTRIDGE PROGRAM

The only exception to the above sequence occurs when a cartridge is plugged into the computer. In the event a cartridge is plugged in, we skip starting up BASIC and instead read in a binary program from the cartridge. During startup we rely on the **CARTCHECK** routine to see if a cartridge is plugged in the expansion port.

```
        JSR CARTCHECK         ; Check for cartridge plugged in
        BEQ BASIC

        STZ IOMODE            ; Cartridge found, load and run binary instead of BASIC
        STZ IOBAUD
        JMP LOADBIN
```

*The section in **MAIN** that checks for a cartridge.*

CARTCHECK toggles some lines on the expansion port to determine if a cartridge is plugged in. If a cartridge is present, the CA1 and CA2 lines on the 65C22 VIA will be connected by a trace on the cartridge's printed circuit board. If not, the CA1 line will be pulled low by a pulldown resistor built into the Cody Computer itself. We set up the 65C22 so that the CA1 line is positive-edge triggered, then bring CA2 high. If CA1 detected a positive edge, we know a cartridge is connected. If not, then no cartridge is present.

```
CARTCHECK LDA #$0D            ; Set CA2 to LOW output, CA1 to positive edge trigger
          STA VIA_PCR

          LDA VIA_IORA        ; Clear the existing CA1 flag value in the VIA_IFR register

          LDA #$0F            ; Toggle CA2 HIGH
          STA VIA_PCR

          LDA VIA_IFR         ; Push the CA1 flag value in the VIA_IFR register for later
          PHA

          LDA #$0D            ; Set CA2 to LOW output, CA1 to positive edge trigger
          STA VIA_PCR

          LDA VIA_IORA        ; Clear the existing CA1 flag value in the VIA_IFR register

          PLA                 ; Pop the stored CA1 flag value and test if bit was set
          AND #$02

          RTS
```

*The **CARTCHECK** routine for cartridge detection.*

If a cartridge is detected, the **LOADBIN** routine is called to load binary code from the cartridge's SPI EEPROM. This routine actually handles loading of binary code from both serial and SPI sources to save

space, but different underlying routines are called depending on the use case. For loading from SPI, three helper routines exist to handle SPI communications. The **CARTON** routine starts an SPI transaction, the **CARTOFF** routine ends an SPI transaction, and the **CARTXFER** routine simultaneously sends and receives a byte over SPI.

```
CARTXFER  PHX
          STA SPIOUT
          STZ SPIINP
          LDX #8              ; 8 bits to read
_LOOP     STZ VIA_IORB        ; Bring the clock low
          LDA #Ø              ; Start with no data
          ROL SPIOUT          ; Get output bit
          BCC _SEND
          ORA #CART_MOSI      ; Output bit was a 1
_SEND     STA VIA_IORB        ; Put the bit on MOSI
          ORA #CART_CLK       ; Bring the SPI clock high
          STA VIA_IORB
          ROL SPIINP          ; Rotate SPI input for next bit
          LDA VIA_IORB        ; Read the incoming MISO
          AND #CART_MISO
          BEQ _NEXT
          LDA SPIINP
          ORA #1
          STA SPIINP
_NEXT     DEX                 ; Next loop (if any remain)
          BNE _LOOP
          PLX
          LDA SPIINP
          RTS
```

*The **CARTXFER** routine transfers a single byte over SPI.*

An additional complication exists for cartridges as they have two possible address sizes: 16 bits (for cartridges up to 64 kilobytes) and 24 bits (for larger SPI memories). The **LOADBIN** routine takes this into

account, something we'll talk about when we discuss loading and saving of programs later on.

```
        LDX #2              ; Assume a cartridge with a two-byte address

        LDA VIA_IORB        ; If cart size bit is high, we have a three-byte address
        BIT #CART_SIZE
        BEQ _ADDR
        INX
```

*Portion of **LOADBIN** that checks for the cartridge's size.*

TOKENIZATION AND INTERPRETATION

Running programs in Cody BASIC is a two-step process. The first step is tokenization, where a program's contents are translated to a special internal representation of the program. The second step is interpretation, where the tokenized program is executed line by line and its statements processed. Both steps occur regardless of the nature of the program, whether it's a single line entered in REPL mode, an entire program that's been typed in by the user, or a program loaded in over a serial port.

TOKENIZATION

Certain keywords or symbols in Cody BASIC are converted into tokens. This approach, common to many 1980s BASIC implementations, serves two purposes. The first is that by reducing an entire word, such as **RETURN**, to a one-byte token like **$8A**, we save considerable space in BASIC program memory. The second is that the program can be interpreted far more quickly.

Instead of having to process each letter and determine what to do at the end of the keyword, we can just test if a byte falls within a certain range reserved for tokens. If so, we know we have a keyword or other special value. In some cases, the tokens can be used as indexes into a jump table, making our interpreter code even faster.

The tokenization occurs in the **TOKENIZE** routine. It takes the contents of a line in the input buffer **IBUF** and converts it to a tokenized line in the token buffer **TBUF**. A tokenized line at this point consists of the same text contents as its original, except that certain keywords, symbols, and literals are replaced by their token equivalents. Constants beginning with the **TOK_** prefix define the numeric values of the tokens.

```
_LOOP     LDA IBUF,X          ; Load the next character

          CMP #CHR_NL         ; End of line?
          BEQ _END

          CMP #CHR_QUOTE      ; String?
          BEQ _STR

          JSR ISALPHA         ; Letter?
          BCS _LET

          JSR ISDIGIT         ; Digit?
          BCS _NUM

          CMP #CHR_LESS       ; Rule out relational operator ranges
          BCC _CHR

          CMP #CHR_QUEST
          BCS _CHR

          JMP _OPR            ; Relational operators handled as special case
```

*Main loop of the **TOKENIZE** routine.*

Tokens always begin with a single byte that has its highest bit set to 1. As a practical matter, this means that BASIC tokens begin at **$80** in hex or 128 in decimal. Tokens for keywords are only a single byte in size. Numbers are the only exception and begin with a

sentinel value of **$FF** followed by a 16-bit unsigned number in little-endian format (lowest byte stored first). Strings are not tokenized and are delimited by ASCII double-quote characters. Contents within the strings are not tokenized.

```
_NUM      LDA #<IBUF          ; Input buffer lower byte
          STA MEMSPTR

          LDA #>BUF          ; Input buffer high byte
          STA MEMSPTR+1

          PHY                 ; Preserve current token buffer position

          TXA                 ; Move the current input buffer position into the y-register
          TAY

          JSR TONUMBER        ; Parse the number

          TYA                 ; Move the updated input buffer position back into the x-registe
          TAX

          PLY                 ; Restore the token buffer position off the stack

          LDA #$FF            ; Write the sentinel value for a number token
          JSR _PUT

          LDA NUMANS          ; Store number low byte
          JSR _PUT

          LDA NUMANS+1        ; Store number high byte
          JSR _PUT

          JMP _LOOP
```

*Part of the **TOKENIZE** routine that handles numbers.*

The actual text of the tokens is kept in a page-aligned table of string literals. To save space, instead of terminating each literal in the table with a null character, the high bit on the final character is set. This saves many bytes of space but makes reading from the table more complicated. Tokens are mapped to message constants starting at **MSG_TOKENS** from the start of the message number table. Because both the token string and message string tables are page-aligned only the low byte of the address is kept in some of the message lookup tables.

To match a substring to its token value we use a binary search algorithm. The **_TOKTABLE** in the **TOKENIZE** routine stores token values in their alphabetical order to assist with the binary search process. This table is used by the routine to more quickly match incoming text to tokens.

```
         STZ TOKENIZEL       ; Prepare for binary search
         LDA #(_TOKTABLEEND – _TOKTABLE)
         STA TOKENIZER

_TOKNEXT LDA TOKENIZEL       ; Are we done yet? (L <= R)
         CMP TOKENIZER

         BCC _TOKCOMP
         BEQ _TOKCOMP

         PLY                 ; Restore token buffer (Y) and input buffer (X) positions
         PLX

         JMP _CHR            ; Process as normal character

_TOKCOMP CLC                 ; Calculate our position in the token lookup table
         LDA TOKENIZEL
         ADC TOKENIZER
         LSR A
         TAX

         PHX

         LDA _TOKTABLE,X     ; Get the token's matching index in the string table
         TAX

         LDA TOKTABLE_L,X    ; Put the token's address in the memory destination pointer
         STA MEMDPTR
         LDA #TOKTABLE_H
         STA MEMDPTR+1

         PLX

         LDY #$00            ; Use the y register for our position in the strings

_TOKCHAR LDA (MEMDPTR),Y     ; Get the destination char and test the high bit for the end of
         BIT #$80
         PHP

         AND #$7F            ; Mask out the valid portion of the char for later comparision
         STA SYS_A

         LDA (MEMSPTR),Y     ; Get the next character from the input string and UPPERCASE it
         JSR TOUPPER

         CMP SYS_A           ; Compare it to the token string and see if we still match
         BEQ _TOKOK
         BCC _TOKLO
         BCS _TOKHI

_TOKOK   INY                 ; Move to next char

         PLP                 ; If we've reached the end of the token we're testing against, w
         BNE _TOKYES
         BRA _TOKCHAR

_TOKHI   PLP
         TXA                 ; Input token was greater, move to top partition
         INC A
         STA TOKENIZEL
```

```
          BRA _TOKNEXT

_TOKLO    PLP
          TXA               ; Input token was less, move to bottom partition
          DEC A
          STA TOKENIZER
          BRA _TOKNEXT
```

*Binary search as implemented in the **TOKENIZE** routine.*

The performance of the tokenization process is very important to the overall usability of the Cody Computer. Unlike most tokenized BASICs, Cody BASIC does not use its tokenized form when a copy is saved via **SAVE** or loaded via **LOAD**. Instead, all tokens are converted back to their plain text to make the content readable in just about any text editor. This means that when a program is loaded over a serial connection, it must also be tokenized. This also means that the loading speed of a BASIC program is largely limited by how fast the incoming text can be tokenized.

```
_REM      LDA IBUF,X          ; Skip tokenizing after a REMARK to save time

          CMP #CHR_NL         ; End of line?
          BEQ _REMEND

          JSR _PUT            ; Copy the character

          INX                 ; Next character
          BRA _REM

_REMEND   JMP _END
```

*A **TOKENIZE** optimization that skips over **REM** comments.*

LINE INSERTION AND DELETION

Once a line is tokenized it's either evaluated immediately or added to the program. The REPL loop examines the contents of the token buffer **TBUF** and checks if the line begins with a number. If it does, it

means the line is either being added, replaced, or deleted from the program, which is handled by the **ENTERLINE** routine.

It extracts the line number from the token buffer and calls **FINDLINE** to determine the line's starting location within program memory. If the line exists, the contents of program memory are shifted downward to delete the existing line. Unless the line is empty (containing only the line number), program memory is then shifted upward to make room for the new line. **INSLINE** is called to handle the actual insertion.

```
ENTERLINE PHA                ; Preserve registers

          LDA TBUF+1         ; Get the line number we're looking for
          STA LINENUM+Ø
          LDA TBUF+2
          STA LINENUM+1

          JSR FINDLINE       ; See if the line number entered already exists
          BCC _NEW

_DEL      LDA LINEPTR+Ø      ; Use matching line as destination (deleting line by copying ove
          STA MEMDPTR+Ø
          LDA LINEPTR+1
          STA MEMDPTR+1

          CLC                ; Calculate end of matching line as the source pointer
          LDA MEMDPTR+Ø
          ADC (LINEPTR)
          STA MEMSPTR+Ø
          LDA MEMDPTR+1
          ADC #Ø
          STA MEMSPTR+1

          SEC                ; Calculate number of bytes to move down from the top
          LDA PROGTOP+Ø
          SBC MEMSPTR+Ø
          STA MEMSIZE+Ø
          LDA PROGTOP+1
          SBC MEMSPTR+1
          STA MEMSIZE+1

          SEC                ; Adjust the top address in program memory because we deleted a
          LDA PROGTOP+Ø
          SBC (LINEPTR)
          STA PROGTOP+Ø
          LDA PROGTOP+1
          SBC #Ø
          STA PROGTOP+1

          JSR MEMCOPYDN      ; Delete the current line by moving memory down

_NEW      LDA TBUFLEN        ; If nothing on the new line, don't insert anything (just a dele
          CMP #4
          BEQ _END

          LDA LINEPTR+Ø      ; Is our insertion position the same as the top of program memor
          CMP PROGTOP+Ø
          BNE _MOV
```

```
         LDA LINEPTR+1
         CMP PROGTOP+1
         BNE _MOV

         BRA _INS            ; If so, we can just insert without copying memory to make space

_MOV     LDA LINEPTR+1       ; If we're on the last page of program memory just say we're out
         CMP PROGEND
         BEQ _SYS

         LDA LINEPTR+0       ; Use the insertion position as source pointer to move memory
         STA MEMSPTR+0
         LDA LINEPTR+1
         STA MEMSPTR+1

         CLC                 ; Calculate the destination pointer for copying memory
         LDA MEMSPTR+0
         ADC TBUFLEN
         STA MEMDPTR+0
         LDA MEMSPTR+1
         ADC #0
         STA MEMDPTR+1

         SEC                 ; Calculate the amount of memory to copy to make room for the ne
         LDA PROGTOP+0
         SBC MEMSPTR+0
         STA MEMSIZE+0
         LDA PROGTOP+1
         SBC MEMSPTR+1
         STA MEMSIZE+1

         JSR MEMCOPYUP       ; Copy the memory up to make room for the new line

_INS     JSR INSLINE         ; Insert the line

_END     PLA                 ; Restore registers

         RTS

_SYS     JMP RAISE_SYS       ; Indicate we're out of BASIC program memory
```

*The **ENTERLINE** routine handles lines entered into the REPL.*

The **FINDLINE** routine determines the insert location for a new line. If a line already exists with the same number, it will return that location instead. The routine works by starting at **PROGMEM**, the base of program memory, and continuing until either a matching line number is found (indicating the line is present) or a line number that is larger is found (indicating the line does not exist).

To compare line numbers it examines the second and third bytes in each line, which contain the low and high bytes of the line number. If it needs to move to the following line, the first byte of the line, containing the line length, is added to the current pointer in

LINEPTR to move forward. If **LINEPTR** is ever equal to **PROGTOP**, the top of program memory, it means the line does not exist and should be appended to the end of the program.

FINDLINE is also used by the BASIC interpreter to find destination line numbers in **GOTO** and **GOSUB** statements.

```
FINDLINE  PHA                 ; Preserve registers
          PHY

          LDA #<PROGMEM       ; Start at the beginning of program memory
          STA LINEPTR+0
          LDA #>PROGMEM
          STA LINEPTR+1

_LOOP     LDA LINEPTR+0       ; Ensure that we're not at the top of program memory already
          CMP PROGTOP+0
          BNE _COMP

          LDA LINEPTR+1
          CMP PROGTOP+1
          BNE _COMP

          BRA _NO

_COMP     LDY #2              ; Skip leading line size byte when doing line number comparison

          LDA (LINEPTR),Y     ; Compare current and desired line number high bytes
          CMP LINENUM+1
          BNE _TEST

          DEY                 ; Compare current and desired line number low bytes
          LDA (LINEPTR),Y
          CMP LINENUM

_TEST     BEQ _YES            ; Found a match

          BCS _NO             ; Current line greater than desired line number, doesn't exist

          CLC                 ; Current line less than desired line number, move to next line

          LDA LINEPTR+0       ; Add current line size to low address byte
          ADC (LINEPTR)
          STA LINEPTR+0

          LDA LINEPTR+1       ; Propagate carry to high address byte
          ADC #0
          STA LINEPTR+1

          BRA _LOOP

_NO       CLC                 ; No match found, clear carry
          BRA _END

_YES      SEC                 ; Match found, set carry

_END      PLY                 ; Restore registers
          PLA

          RTS
```

*Finding a line's insert position is handled by **FINDLINE**.*

Insertion of a line is handled by **INSLINE**. It assumes that appropriate space has already been allocated for the new line (by **ENTERLINE**) and doesn't move any contents within program memory. Instead, it copies the contents of the token buffer **TBUF** into a specified address in program memory. It also somewhat modifies the line contents, changing the first byte from **$FF** (representing the start of a number token) to the line's length in bytes. When done, the value of **PROGTOP** is incremented by the line's length to reflect the increased size of the program.

The **INSLINE** routine is also used by the **LOADBAS** routine when a BASIC program is being loaded from storage over the serial port. In this case lines are being appended to the top of the program as they come in and get tokenized. This allows us to skip over some unrelated code not needed for this special case of line insertion.

```
INSLINE   LDA LINEPTR+1       ; If we're on the last page of program memory just say we're out
          CMP PROGEND
          BEQ _SYS

          LDA TBUFLEN         ; Store token buffer length as first byte in line
          STA TBUF

          STA MEMSIZE+0       ; Set size of memory to copy into program buffer
          STZ MEMSIZE+1

          LDA #<TBUF          ; Use token buffer as source pointer
          STA MEMSPTR+0
          LDA #>TBUF
          STA MEMSPTR+1

          LDA LINEPTR+0       ; Use line pointer found for line number as destination pointer
          STA MEMDPTR+0
          LDA LINEPTR+1
          STA MEMDPTR+1

          JSR MEMCOPYDN       ; Copy the memory

          CLC                 ; Update the top of memory to the new location
          LDA PROGTOP+0
          ADC TBUFLEN
          STA PROGTOP+0
          LDA PROGTOP+1
          ADC #0
          STA PROGTOP+1

          RTS
```

```
_SYS      JMP RAISE_SYS       ; Indicate we're out of BASIC program memory
```

INSLINE *routine for inserting a line into the program.*

INTERPRETATION

Once Cody BASIC code is tokenized, it can be executed via interpretation. The core of the interpreter is a recursive-descent parser that goes through each tokenized line looking for tokens and calling the appropriate routines to handle them. The **PROGPTR** zero-page variable points to the start of the current line while another zero-page variable, **PROGOFF**, stores the current position within the line. For evaluating mathematical expressions or passing values between interpreter routines, a dedicated expression stack exists in zero page (**EXPRS_L** for low bytes, **EXPRS_H** for high bytes).

The starting point for interpretation is the **EXSTMT** routine that interprets a single statement. It examines the first token in the current line, converts it to an index into a jump table, and jumps to the appropriate routine to handle the statement type. When the called routine returns, because we did a jump rather than a subroutine call, control will return back to the routine that called **EXSTMT**. While somewhat hackish, this works around the 65C02's inability to perform an indirect subroutine call. (A more generic way around the same problem is to perform a subroutine call to the code that does the jump, but for our specific purpose, what we have works quite well.)

Note that the routines that are part of the recursive-descent interpreter are usually prefixed with **EX** to indicate they're used to execute the program. You can

see many of these routines in the jump table included below.

```
EXSTMT    STZ EXPRSNUM        ; Start at the bottom of the expression stack

          JSR EXSKIP          ; Skip any whitespace before we run into a token

          LDY PROGOFF         ; Get the current offset in the current line

          LDA (PROGPTR),Y     ; Get the current byte

          CMP #CHR_NL         ; Was it a newline? If so the entire line was blank
          BEQ _END

          CMP #TOK_SYS+1      ; Check that the byte isn't too big to be a valid token
          BCS _SYN

          SEC                 ; Subtract from the first statement token to get the index
          SBC #TOK_NEW

          BCC _ASN            ; If the result was less than that, assume it was an assignment

          ASL A               ; Multiply by two to convert the number into a jump table index
          TAX

          INC PROGOFF         ; Increment the current offset since we consumed the token

          JMP (_JMP,X)        ; Jump to the code for the statement we have

_END      RTS

_ASN      JMP EXASSIGN        ; Jump to the assignment

_SYN      JMP RAISE_SYN       ; Raise syntax error

_JMP      .WORD EXNEW
          .WORD EXLIST
          .WORD EXLOAD
          .WORD EXSAVE
          .WORD EXRUN
          .WORD EXNOP
          .WORD EXIF
          .WORD _SYN
          .WORD EXGOTO
          .WORD EXGOSUB
          .WORD EXRETURN
          .WORD EXFOR
          .WORD _SYN
          .WORD EXNEXT
          .WORD EXPOKE
          .WORD EXINPUT
          .WORD EXPRINT
          .WORD EXOPEN
          .WORD EXCLOSE
          .WORD EXREAD
          .WORD EXRESTORE
          .WORD EXNOP
          .WORD EXEND
          .WORD EXSYS
```

***EXSTMT** is the highest-level routine in the interpreter.*

The **REPL** loop relies on **EXSTMT** to run the lines of BASIC code the user enters. In this mode, each entered line that is not an edit is executed immediately. To

make this happen, **PROGOFF** is set to zero, **PROGPTR** is pointed to the token buffer, and **EXSTMT** is called to execute the line. Once the line has been executed control returns to the REPL loop for further input.

```
_EXEC     STZ PROGOFF           ; Start at the beginning of the line

          LDA #<TBUF            ; Use the token buffer as the line we're going to run
          STA PROGPTR
          LDA #>TBUF
          STA PROGPTR+1

          JSR EXSTMT            ; Execute the statement in the token buffer

          STZ OBUFLEN           ; Move to beginning of output buffer

          LDA #MSG_READY        ; Print the ready message after each REPL operation
          JSR PUTMSG
          JSR FLUSH

          BRA REPL              ; Next read-eval-print loop
```

*The **_EXEC** portion of the **REPL** code.*

Running an entire program using the **RUN** command is very similar, except that lines are interpreted in succession until the program comes to a stop. Interestingly, it's the responsibility of the interpreter itself to begin interpreting a full program, as the **RUN** statement is actually implemented within the interpreter itself. When a user enters the **RUN** statement in the REPL loop, the interpreter calls the **EXRUN** routine to execute it, running the program.

EXRUN starts out by clearing the current interpreter state back to some sane default values. It also has to set the **RUNMODE** so other code, particularly the error handler, knows that we're running a program. It positions the **PROGPTR** to the start of the program, then begins evaluating each line one at a time by calling **EXSTMT**.

As an additional complication, some statements can change the interpreter's current position in the

program. For example, a **GOTO** statement could move the current position far away from the current line, and other statements related to control flow have similar effects.

To handle these situations, **EXRUN** also calculates a **PROGNXT** pointer to the *next* line to execute before executing the current line. Once the current line is executed, it goes to the line pointed to by **PROGNXT**. Under normal circumstances this will be the line after the current one, but for statements that modify the control flow, the value can be replaced with a different one when the control statement runs.

```
EXRUN     JSR ONLYREPL        ; Only valid in REPL mode

          JSR NEWVARS         ; Reset variable memory

          JSR RESTORE         ; Reset data buffer for DATA/READ statements

          LDA #RM_PROGRAM     ; Set RUNMODE to running
          STA RUNMODE

          STZ GOSUBSNUM       ; Start out with empty GOSUB/RETURN and FOR/NEXT stacks
          STZ FORSNUM

          LDA #<PROGMEM       ; Use the start of program memory as our starting position
          STA PROGPTR
          LDA #>PROGMEM
          STA PROGPTR+1

_LOOP     LDA RUNMODE         ; Check that we're still running (e.g. no END statement was exec
          BEQ _DONE

          JSR ISEND           ; Make sure that this line isn't actually the end of the prograr
          BEQ _DONE

_CONT     CLC                 ; Prepare to calculate the NEXT line we'll be running

          LDA PROGPTR         ; Calculate the low byte by adding our pointer to the line's siz
          ADC (PROGPTR)
          STA PROGNXT

          LDA PROGPTR+1       ; Propagate the carry
          ADC #0
          STA PROGNXT+1

          LDA #4              ; Start at the first non-line-number position in the current lir
          STA PROGOFF

          JSR EXSTMT          ; Execute the statement on this line

          LDA PROGNXT         ; Copy the NEXT line's pointer over to use as the current line
          STA PROGPTR
          LDA PROGNXT+1
          STA PROGPTR+1

          BRA _LOOP           ; Repeat, run the next statement

_DONE     STZ RUNMODE         ; Clear run mode
```

```
        STZ IOMODE          ; Clear IO mode

        RTS                 ; Done
```

EXRUN *runs an entire program from within the interpreter.*

The interpreter supports 26 numeric arrays, **A** through **Z**, each capable of holding up to 128 numbers. An additional 26 string variables, **A$** through **Z$**, also exist with a maximum length of 255 characters plus a terminating NUL char. These reside in the **DATAMEM** portion of the interpreter's memory, with each array or string aligned to a single 256-byte page in the 65C02's memory. Numeric variables start at **ARRA** through **ARRZ** while string variables start at **STRA** through **STRZ**. The interpreter's **EXVAR** routine parses variables and calculates the actual memory address associated with them, including any array indexes for number variables.

```
EXVAR   JSR EXSKIP          ; Consume leading space

        LDY PROGOFF         ; Load the next character from the current line
        LDA (PROGPTR),Y

        INC PROGOFF         ; Consume the character

        JSR ISALPHA         ; If not a letter, it’s a syntax error
        BCC _SYN

        SEC                 ; Calculate the page number assuming we have an array variable
        SBC #CHR_AUPPER

        CLC                 ; Determine the actual page location based on the start of vars
        ADC #>ARRA

        STZ NUMANS          ; Assume by default we DO NOT have an index into an array
        STA NUMANS+1

        LDY PROGOFF         ; Load another character
        LDA (PROGPTR),Y

        CMP #CHR_DOLLAR     ; String variable so we need to adjust our pointer into string m
        BEQ _STR

        CMP #CHR_LPAREN     ; Array index so we need to adjust our pointer within array memo
        BNE _NUM

        JSR EXLPAREN        ; Consume left parenthesis

        LDA NUMANS+1        ; Preserve high byte of variable address (will be clobbered by e
        PHA
```

```
          JSR EXEXPR          ; Evaluate expression for array index

          PLA                 ; Restore the high byte of the variable address (just got clobbe
          STA NUMANS+1

          JSR EXRPAREN        ; Consume right parenthesis

          JSR POPONE          ; Pop the array index off the stack

          LDA NUMONE+1        ; High byte should be zero (or will be out of range)
          BNE _LOG

          LDA NUMONE          ; Low byte should be less than 128 (or will be out of range)
          BIT #$80
          BNE _LOG

          ASL A               ; Shift low byte by one (multiply by two because numbers are two

          STA NUMANS          ; Store the index as the low byte

_NUM      JSR PUSHANS         ; Store the address of the variable

          CLC                 ; Clear carry to indicate it's a number variable

          RTS                 ; All done

_STR      CLC                 ; Adjust pointer from array memory to string memory
          LDA #26
          ADC NUMANS+1
          STA NUMANS+1

          INC PROGOFF         ; Consume dollar sign

          JSR PUSHANS         ; Store the address of the variable

          SEC                 ; Set carry to indicate it's a string variable

          RTS                 ; All done

_SYN      JMP RAISE_SYN       ; Raise a syntax error

_LOG      JMP RAISE_LOG       ; Raise a logic error (array index out of bounds)
```

*The **EXVAR** routine calculates a variable's memory address.*

In addition to the many interpreter routines that execute specific statements or functions in Cody BASIC, there are helper routines used by the interpreter. Some are part of the BASIC interpreter itself, such as **EXSKIP** (used for skipping whitespace), **EXLPAREN** and **EXRPAREN** (used for parsing parentheses), and **EXCHARACT** (used for requiring that the next character in a line is a certain value). Routines such as **EXONEARG** and **EXTWOARG** consolidate code for parsing one-argument and two-

argument mathematical functions, while **EXSTRARG** does something similar for string functions.

```
EXTWOARG  JSR EXLPAREN
          JSR EXEXPR
          JSR EXCOMMA
          JSR EXEXPR
          JSR EXRPAREN
          RTS
```

EXTWOARG *combines helper routines into another helper routine.*

Other helper routines also exist outside the interpreter core. Math routines such as **MUL16**, **DIV16**, **RND16**, and **SQR16** perform 16-bit math calculations needed to implement some of Cody BASIC's mathematical functions. Other routines such as **POPONE**, **POPBOTH**, and **PUSHANS**, assist in moving values back and forth between the expression stack and the **NUMONE**, **NUMTWO**, and **NUMANS** zero-page variables used by many interpreter and helper routines.

```
POPONE    PHA                 ; Preserve registers
          PHX

          LDX EXPRSNUM        ; Fetch the current size of the expression stack

          LDA EXPRS_L-1,X     ; Store the low byte into NUMONE
          STA NUMONE

          LDA EXPRS_H-1,X     ; Store the high byte into NUMONE
          STA NUMONE+1

          DEC EXPRSNUM        ; Decrement the count by one

          PLX                 ; Restore registers
          PLA

          RTS                 ; All done
```

POPONE *removes the top value from the expression stack.*

NUMERIC AND STRING EXPRESSIONS

Cody BASIC supports numeric and string expressions. It's not possible to go over the implementation of every single command in Cody BASIC (though the code is heavily documented), but by studying how some of the math and string operations are implemented, it's possible to develop a greater understanding of how the BASIC interpreter's recursive-descent parser works in practice.

Numeric expressions, like everything in Cody BASIC, follow the language's grammar. A numeric EXPR contains a TERM followed by zero or more addition or subtraction operators and TERMs. In turn, the TERM is defined much the same, except that it begins with a single FACTOR followed by zero or more multiplication or division operators and FACTORs. Lastly, a FACTOR can be any of a variety of numeric types, including number literals, numeric functions, variables, or even a nested expression in parentheses. Note that this approach also preserves operator precedence, as individual numbers or nested expressions end up evaluated first, followed by multiplication and division, and only last are addition and subtraction performed.

An EXPR is implemented in the interpreter by the **EXEXPR** routine. It calls another routine, **EXTERM**, to handle the initial term, then loops as long as an addition or subtraction operator is present. If one is present, it parses the operator, calls **EXTERM** to get the other operand, and then performs the calculation.

Because the operands are pushed on the expression stack, the values are obtained from there and the result stored there as well.

```
EXEXPR    JSR EXTERM          ; Evaluate the left side of the (possible) operator

_LOOP     JSR EXSKIP          ; Skip any leading space

          LDY PROGOFF         ; Load the next character
          LDA (PROGPTR),Y

          CMP #CHR_PLUS       ; Addition operation
          BEQ _ADD

          CMP #CHR_MINUS      ; Subtraction operation
          BEQ _SUB

          RTS                 ; All done

_ADD      INC PROGOFF         ; Consume plus character

          JSR EXTERM          ; Evaluate the right side of the plus sign

          LDX EXPRSNUM        ; Find how many items we have on the expression stack

          CLC                 ; Prepare for addition

          LDA EXPRS_L-2,X     ; Add number low bytes together and put back on stack
          ADC EXPRS_L-1,X
          STA EXPRS_L-2,X

          LDA EXPRS_H-2,X     ; Add number high bytes together and put back on stack
          ADC EXPRS_H-1,X
          STA EXPRS_H-2,X

          DEC EXPRSNUM        ; Decrement stack by one (took two values off, put result back o

          BRA _LOOP           ; Next

_SUB      INC PROGOFF         ; Consume minus character

          JSR EXTERM          ; Evaluate the right side of the minus sign

          LDX EXPRSNUM        ; Find how many items we have on the expression stack

          SEC                 ; Prepare for subtraction

          LDA EXPRS_L-2,X     ; Subtract number low bytes and put back on stack
          SBC EXPRS_L-1,X
          STA EXPRS_L-2,X

          LDA EXPRS_H-2,X     ; Subtract number high bytes and put back on stack
          SBC EXPRS_H-1,X
          STA EXPRS_H-2,X

          DEC EXPRSNUM        ; Decrement stack by one (took two values off, put result back o

          BRA _LOOP           ; Next
```

***EXEXPR** executes the code for a numeric expression.*

The **EXTERM** routine implements the same but for TERMs. In this case, **EXFACTOR** is called to put the first operand on the expression stack. Then the code

continues to loop as long as a multiplication or division operator is present, calling **EXFACTOR** for the other operand if so.

In this case the actual calculation is less straightforward as the 65C02 does not support any hardware multiplication or division. Instead, we perform the calculation in software, calling **POPBOTH** to get the top values of the expression stack into **NUMONE** and **NUMTWO**. We then call **MUL16** or **DIV16** to perform the calculation. Lastly, we push the single result in **NUMANS** on the stack by calling **PUSHANS**.

```
EXTERM    JSR EXFACTOR        ; Evaluate the left side of the (possible) operator

_LOOP     JSR EXSKIP          ; Skip any leading space

          LDY PROGOFF         ; Load the next character
          LDA (PROGPTR),Y

          CMP #CHR_ASTERISK   ; Multiplication operation
          BEQ _MUL

          CMP #CHR_SLASH      ; Division operation
          BEQ _DIV

          RTS                 ; All done

_MUL      INC PROGOFF         ; Consume multiply operator

          JSR EXFACTOR        ; Evaluate the right side of the multiply sign

          JSR POPBOTH         ; Pop both values off the expression stack

          JSR PRE16

          PHA

          JSR MUL16           ; Multiply the numbers together

          PLA

          JSR ADJ16

          JSR PUSHANS         ; Push the result back on the stack

          BRA _LOOP           ; Next

_DIV      INC PROGOFF         ; Consume divide operator

          JSR EXFACTOR        ; Evaluate the right side of the division sign

          JSR POPBOTH         ; Pop both values off the expression stack

          JSR PRE16

          PHA

          JSR MOD16           ; Divide using the modulus operation (division result is also ca
```

```
        LDA NUMONE          ; Copy division result low byte (from the modulus) to the answer
        STA NUMANS

        LDA NUMONE+1        ; Copy division result high byte (from the modulus) to the answe
        STA NUMANS+1

        PLA

        JSR ADJ16

        JSR PUSHANS         ; Push the result back on the stack

        BRA _LOOP           ; Next
```

*Numeric terms are executed by the **EXTERM** routine.*

The **EXFACTOR** has to handle the many possiblities of a FACTOR in the grammar. Negative numbers beginning with a unary minus, expressions in parentheses, numeric variables, functions, and number literals all need to be handled. To decide what to do, it begins by examining the next token and branching to an appropriate part of its code.

For number literals, it simply pushes the value of the number on the stack. For minus signs, it attempts to interpret the next value as a number by calling **EXFACTOR** itself, then flips its sign via subtraction. For nested expressions, it parses a left parenthesis via **EXLPAREN**, an EXPR by calling **EXEXPR**, and a right parenthesis via **EXRPAREN**. For variables, it calls **EXVAR** to obtain the variable's memory address then loads the value from there. And for functions, it converts the token's value into an index into a local jump table, jumping to the appropriate routine to handle the function.

```
EXFACTOR  JSR EXSKIP          ; Skip any leading spaces

          LDY PROGOFF         ; Get the offset in the current line

          LDA (PROGPTR),Y     ; Read the character there

          CMP #CHR_MINUS      ; Is it a negative number?
          BEQ _NEG

          CMP #TOK_NUM        ; Is it a number literal?
          BEQ _NUM
```

```
        CMP #CHR_LPAREN     ; Is it a nested expression?
        BEQ _EXP

        JSR ISALPHA         ; Is it a letter for a variable name?
        BCS _VAR

        CMP #TOK_ASC+1      ; Check that the byte isn't too big to be a valid token
        BCS _SYN

        INC PROGOFF         ; Consume the token

        SEC                 ; Subtract the start of the function tokens to get our index
        SBC #TOK_TIME

        BCC _SYN            ; If the result was less than that the token value was too low

        ASL A               ; Multiply by two to convert the number into a jump table index
        TAX

        JMP (_JMP,X)        ; Jump to the code for the function we have

_NUM    INY                 ; Skip the leading $FF tag at the start of the number

        LDA (PROGPTR),Y     ; Fetch number literal low byte
        STA NUMANS
        INY

        LDA (PROGPTR),Y     ; Fetch number literal high byte
        STA NUMANS+1
        INY

        STY PROGOFF         ; Update the offset in the current line

        JSR PUSHANS         ; Push the number onto the expression stack

        RTS                 ; All done

_EXP    JSR EXLPAREN        ; Grab the left parenthesis

        JSR EXEXPR          ; Process the nested expression

        JSR EXRPAREN        ; Grab the right parenthesis

        RTS                 ; All done

_VAR    JSR EXVAR           ; Evaluate variable to get its address in memory

        BCS _SYN            ; If we read a string variable, it's a syntax error here

        JSR POPONE          ; Pop the variable's address off the stack

        LDA (NUMONE)        ; Read and store the low byte of the variable
        STA NUMANS

        INC NUMONE          ; Increment address by one (safe because of page alignment)

        LDA (NUMONE)        ; Read and store the high byte of the variable
        STA NUMANS+1

        JSR PUSHANS         ; Push the number (not its address) on the stack

        RTS

_NEG    INC PROGOFF         ; Consume the unary minus

        JSR EXFACTOR        ; Process the rest of the factor

        LDX EXPRSNUM        ; Get the current expression stack size

        SEC                 ; Prepare to subtract

        LDA #0              ; Subtract low byte from zero in place on stack
        SBC EXPRS_L-1,X
        STA EXPRS_L-1,X

        LDA #0              ; Subtract high byte from zero in place on stack
```

```
          SBC EXPRS_H-1,X
          STA EXPRS_H-1,X

_END      RTS

_SYN      JMP RAISE_SYN       ; Raise a syntax error

_JMP
          .WORD EXTIME
          .WORD EXPEEK
          .WORD EXRND
          .WORD EXNOT
          .WORD EXABS
          .WORD EXSQR
          .WORD EXAND
          .WORD EXOR
          .WORD EXXOR
          .WORD EXMOD
          .WORD EXINT
          .WORD EXLEN
          .WORD EXASC
```

EXFACTOR *handles a variety of numeric literals and values.*

String expressions are handled in a similar way. In some ways string expressions are more complex, while in others they're significantly simpler. Instead of storing values on the expression stack, string expressions are evaluated by copying their contents into the output buffer **OBUF**.

This is possible because string expressions have a significantly reduced grammar, being limited only to concatenation operations, string variables, string literals, and string functions that produce no intermediate values. In other words, a string expression (or STREXPR) consists of one or more string terms, and string terms (STRTERMs) themselves aren't particularly complicated.

```
EXSTREXPR JSR EXSKIP

          JSR EXSTRTERM       ; Evaluate the string term we started with

_LOOP     JSR EXSKIP          ; Skip any leading space

          LDY PROGOFF         ; Load the next character
          LDA (PROGPTR),Y

          CMP #CHR_PLUS       ; Concatenation operator is the only one supported
          BEQ _CAT

          RTS                 ; All done

_CAT      INC PROGOFF         ; Consume operator

          JSR EXSTRTERM       ; Evaluate the next string term to concatenate

          BRA _LOOP           ; Next

          RTS
```

***EXSTREXPR** handles a string expression.*

The **EXSTRTERM** routine is a bit more complicated, but not much so. The STRTERM can only be a string literal, a string variable, or one of a small number of functions that return a string value. String literals and string variables can be handled by copying their contents into the output buffer.

Only three string functions exist, **CHR$**, **STR$**, and **SUB$**. These are handled by checking for their token and jumping to **EXCHR**, **EXSTR**, or **EXSUB** directly. Given the small number of possibilities, a jump table probably isn't worth the overhead.

```
EXSTRTERM LDY PROGOFF         ; Load the next character
          LDA (PROGPTR),Y

          CMP #CHR_QUOTE      ; String literal
          BEQ _LIT

          CMP #TOK_CHR        ; CHR$ function (char code to string)
          BEQ EXCHR

          CMP #TOK_STR        ; STR$ function (number to string)
          BEQ EXSTR

          CMP #TOK_SUB        ; SUB$ function (substring to string)
          BEQ EXSUB

          JSR EXVAR           ; String variable is all we have left
          BCS _VAR

          JMP RAISE_SYN       ; Otherwise it's a syntax error, nothing we can do
```

```
_LIT      INY                 ; Skip the leading quote

_LITLOOP  LDA (PROGPTR),Y     ; Read the next character

          CMP #CHR_NL         ; Newlines shouldn't happen, but if they do, stop immediately
          BEQ _LITDONE

          INY                 ; Consume whatever character we read

          CMP #CHR_QUOTE      ; End quote means we're done with the string literal
          BEQ _LITDONE

          JSR PUTOUT          ; Otherwise just copy the character to the output buffer

          BRA _LITLOOP        ; Repeat

_LITDONE  STY PROGOFF         ; Update the offset in the current line

          RTS                 ; All done

_VAR      JSR POPONE          ; Pop the variable address off the stack

          LDY #Ø              ; Start at the beginning

_VARLOOP  LDA (NUMONE),Y      ; Read the character from the string (zero/NUL indicates end of
          BEQ _VARDONE

          JSR PUTOUT          ; Put the character from the string into the output buffer

          INY                 ; Consume the character

          BEQ _SYS            ; If we wrapped around then we never found a terminating NUL

          BRA _VARLOOP

_VARDONE  RTS                 ; All done

_SYS      JMP RAISE_SYS       ; Raise system error indicating we didn't find a NUL
```

***EXSTRTERM** handles the few possibilities for a term in a string expression.*

The general approach shown for expression evaluation is also the core of the recursive descent mechanism. A more general routine handles a more complicated part of the BASIC language, then calls down into more specific subroutines to handle more specific parts.

For example, printing a numeric calculation's result on the screen would involve **EXSTMT** determining that a **PRINT** statement was to be executed, then jumping to **EXPRINT** to print it. **EXPRINT** would look ahead and see that a numeric expression was in play and call **EXEXPR** to evaluate it. **EXEXPR** would call **EXTERM**, which in turn calls **EXFACTOR**.

CONTROL AND DATA STATEMENTS

Cody BASIC has some special statements that handle control flow and data literals in BASIC programs. While implemented using the same interpreter logic as the rest of Cody BASIC, they have additional effects that set them apart from more straightforward operations such as math calculations or updating variables. These statements also often maintain information outside of the core BASIC interpreter, such as line pointers, and take actions that in some ways override the normal interpreter behavior.

One set of such statements are the control flow statements that change the course of a running program. Cody BASIC supports the typical BASIC commands for such operations: **IF**, **GOTO**, **GOSUB**/**RETURN**, and **FOR**/**NEXT** statements are all implemented.

Many of these statements rely on a similar underlying implementation. Under normal conditions the interpreter sets the value of **PROGNXT** to the start of the next line after **PROGPTR**, but individual statements can overwrite the value to change the path through the program. Different types of control flow statements also have to maintain additional information unique to their own special situations, such as pointers to return lines or terminating loop values.

Another set of statements are those that handle reading of data literals within a program. Many BASIC dialects supported the use of **DATA** statements. A user

could enter raw data separated by commas into these statements, which would be ignored under normal operation of the interpreter. However, when a **READ** statement was executed, values from the **DATA** statements scattered through the program would be stored in variables.

Cody BASIC supports a limited form of this mechanism inspired by Commodore BASIC. To do so, it maintains some external information regarding the current data pointer position and the contents of previous **DATA** statements.

IF STATEMENTS

The **IF** statement is one of the most simple control flow statements. It evaluates a relational expression (an expression that compares two terms). If the expression evaluates to true, it runs the remainder of the statement after the **THEN** keyword. If the expression is false then it skips over the rest of the statement and proceeds to the next line.

The implementation is somewhat complicated because there are two kinds of relational expressions. One is for numbers and compares the results of two numeric expressions. The other is for strings and compares a string variable's contents to a string expression. The typical equal, not-equal, greater-than, less-than, greater-than-or-equal, and less-than-or-equal are all available for both kinds of expressions.

Because there are different kinds of comparisons that must be performed, the comparison testing logic is also somewhat complicated. Once the appropriate comparison has been performed, the code loads a

constant indicating what relational operators would be true given the inputs. This value is ANDed with a constant for the relational operator to determine if the result is true or false.

```
EXIF      JSR EXSKIP          ; Skip any leading space after the "IF"

          LDY PROGOFF         ; Read the first character to see if it could be a string var
          LDA (PROGPTR),Y

          JSR ISALPHA         ; If we have a string var it has to start with a letter
          BCC _NUM

          INY                 ; Read the next character to see if it's a dollar sign
          LDA (PROGPTR),Y

          CMP #CHR_DOLLAR     ; If we have a string var it ends with a dollar sign
          BNE _NUM

_STR      JSR EXVAR           ; Parse a string variable (syntax error if not a string)
          BCC _SYN

          JSR _RELOP          ; Evaluate the relational operator and store the index temporari
          PHA

          STZ OBUFLEN         ; Evaluate the right hand side as a string into the output buffe
          JSR EXSTREXPR

          LDX OBUFLEN         ; Append a NUL to the end of the buffer to make the comparison e
          LDA #0
          STA OBUF,X

          JSR POPONE          ; Pop the string variable address off the stack

          LDY #0              ; Loop over the string in the buffer

_STRLOOP  LDA (NUMONE),Y      ; Compare the characters in the string and the output buffer
          CMP OBUF,Y

          BEQ _STRNEXT        ; Branch depending on the result of the comparison
          BCC _LT
          BRA _GT

_STRNEXT  CMP #0              ; If we have a null char for both, the strings are equal
          BEQ _EQ

          INY                 ; Increment the position in the output buffer to compare to

          BRA _STRLOOP        ; Next character

_SYN      JMP RAISE_SYN       ; Raise a syntax error (needs to be here for branch distance pur

_NUM      JSR EXEXPR          ; Evaluate left hand side of the relational operator

          JSR _RELOP          ; Evaluate the relational operator and store the index temporari
          PHA

          JSR EXEXPR          ; Evaluate the right hand side of the relational operator

          JSR POPBOTH         ; Pop both numbers off the stack

          LDA NUMONE+1        ; Compare high bytes using a signed comparison
          CMP NUMTWO+1

          BEQ _LO
          BMI _LT
          BPL _GT

_LO       LDA NUMONE          ; Compare low bytes using an unsigned comparison
          CMP NUMTWO
```

```
          BEQ _EQ
          BCC _LT
          BRA _GT

_EQ       LDA #(REL_LE | REL_GE | REL_EQ)     ; Equals is true for "<=", ">=", or "="
          BRA _THEN

_LT       LDA #(REL_LE | REL_LT | REL_NE)     ; Less than is true for "<=", ">" or "<>"
          BRA _THEN

_GT       LDA #(REL_GE | REL_GT | REL_NE)     ; Greater than is true for ">=", ">" or "<>"
          BRA _THEN

_THEN     PLX                 ; Get the index in our table for the relational operator

          AND _BITS,X         ; AND the table entry with the possible matches we have

          BEQ _DONE           ; If nothing matches, then the result of the comparison was fals

          LDA #TOK_THEN       ; We expect a "THEN" token after the string
          JSR EXCHARACT

          JMP EXSTMT          ; Then evaluate the rest of the line as its own statement

_DONE     RTS                 ; Nothing to do since condition was false

_BITS     .BYTE REL_LE        ; Lookup table that matches valid relop results with relops
          .BYTE REL_GE
          .BYTE REL_NE
          .BYTE REL_LT
          .BYTE REL_GT
          .BYTE REL_EQ

_RELOP    JSR EXSKIP          ; Skip any leading space

          LDY PROGOFF         ; Load the next character from the line (should be a relop toker
          LDA (PROGPTR),Y

          INC PROGOFF         ; Consume the token

          CMP #(TOK_EQ+1)     ; Was the token out of the expected range (too high)?
          BCS _SYN

          SEC                 ; Adjust token into lookup table value (and check if too low)
          SBC #TOK_LE
          BCC _SYN

          RTS                 ; All done, leave index in accumulator
```

***EXIF** processes **IF** statements and their **THEN** clauses.*

GOTO STATEMENTS

Another simple control flow statement, the **GOTO** statement, simply looks up the line number to go to, then sets the **PROGNXT** pointer to that line's pointer. On the next iteration the interpreter will run the destination line.

```
EXGOTO    JSR ONLYRUN         ; Only valid in RUN mode

          JSR EXEXPR          ; Evaluate the line number to jump to

          JSR POPONE          ; Pop the number off the stack

          LDA NUMONE          ; Copy line number to LINENUM before we search
          STA LINENUM
          LDA NUMONE+1
          STA LINENUM+1

          JSR FINDLINE        ; Try to find a matching line (control flow error if none)
          BCC _LOG

          LDA LINEPTR         ; Use the pointer we found as the next line to execute
          STA PROGNXT
          LDA LINEPTR+1
          STA PROGNXT+1

          RTS                 ; All done

_LOG      JMP RAISE_LOG       ; Indicate the line number was invalid
```

*The **EXGOTO** routine handles **GOTO** statements.*

GOSUB AND RETURN STATEMENTS

GOSUB and **RETURN** statements are somewhat more complicated as the line to return to must be stored somewhere. In Cody BASIC this information is stored in a gosub-return stack using zero-page variables **GOSUBS_L** (for low bytes) and **GOSUBS_H** (for high bytes) containing the return line's address. When a **GOSUB** is executed, the current **PROGNXT** pointer is stored on the stack before jumping to the destination line by delegating to the **EXGOTO** routine. A check is performed to ensure that sufficient space exists in the gosub-return stack.

```
EXGOSUB   JSR ONLYRUN         ; Only valid in RUN mode

          LDX GOSUBSNUM       ; Do we have room on the GOSUB/RETURN stack?
          CPX #MAXSTACK
          BCS _SYS

          LDA PROGNXT         ; Store the NEXT line pointer to execute as our return position
          STA GOSUBS_L,X
          LDA PROGNXT+1
          STA GOSUBS_H,X

          INC GOSUBSNUM       ; Increment stack count (we just pushed an item on it)

          JMP EXGOTO          ; The rest of our statement is just like a GOTO, so go there

_SYS      JMP RAISE_SYS       ; Indicate the GOSUB-RETURN stack is out of memory
```

EXGOSUB *preserves the next line pointer before branching.*

When a **RETURN** statement is executed, the top value on the gosub-return stack is popped and used as the new value for **PROGNXT**. This returns control to the line after the **GOSUB** that pushed the value on the stack, working just as we'd expect. We also have to do a check to ensure there's a value on the stack at all, otherwise we have a **RETURN** without a matching **GOSUB**.

```
EXRETURN  JSR ONLYRUN         ; Only valid in RUN mode

          LDX GOSUBSNUM       ; Load the number of GOSUB/RETURN entries (control flow error if
          BEQ _LOG

          LDA GOSUBS_L-1,X    ; Copy the top item on the GOSUB/RETURN stack as our next line t
          STA PROGNXT
          LDA GOSUBS_H-1,X
          STA PROGNXT+1

          DEC GOSUBSNUM       ; Decrement count (we just removed an item from the stack)

          RTS                 ; All done

_LOG      JMP RAISE_LOG       ; Indicate we have a RETURN without a GOSUB
```

EXRETURN *pops the line pointer and returns control to that location.*

FOR AND NEXT STATEMENTS

Implementing **FOR** and **NEXT** statements is somewhat more complex. The line to return to in the **FOR** loop must be preserved similar to the return line in a **GOSUB**. However, we also have to keep a pointer to the **FOR** loop's variable so we can update it on each loop. We also have to keep the stop value so we know when the end of the loop has been reached. Cody BASIC's solution is to use a stack that is similar to the gosub-return loop, but with extra values for a variable pointer and a stop value. This information is kept in the **FORLINE_L/FORLINE_H**, **FORVARS_L/FORVARS_H**, and **FORSTOP_L/FORSTOP_H** zero-page variables.

```
EXFOR     JSR ONLYRUN         ; Only valid in RUN mode

          JSR EXVAR           ; Evaluate the loop variable as an lvalue (only number vars)
          BCS _SYN

          JSR EXEQUALS        ; Consume equals

          JSR EXEXPR          ; Evaluate starting expression

          LDA #TOK_TO         ; Consume "TO"
          JSR EXCHARACT

          JSR EXEXPR          ; Evaluate ending expression

          LDX FORSNUM         ; Do we have room on the FOR/NEXT stack?
          CPX #MAXSTACK
          BCS _SYS

          LDA PROGNXT         ; Store the line pointer to execute as our return position
          STA FORLINE_L,X
          LDA PROGNXT+1
          STA FORLINE_H,X

          JSR POPONE          ; Pop the ending value for the FOR loop off the stack

          LDA NUMONE          ; Store the ending value into the FORSTOPs
          STA FORSTOP_L,X
          LDA NUMONE+1
          STA FORSTOP_H,X

          JSR POPBOTH         ; Pop the variable address and the initial value off the stack

          LDA NUMONE          ; Store the variable address into the FORVARS
          STA FORVARS_L,X
          LDA NUMONE+1
          STA FORVARS_H,X

          LDA NUMTWO          ; Store the low byte of the initial loop value
          STA (NUMONE)
```

```
          INC NUMONE          ; Move to the high byte (relies on page alignment to be safe)

          LDA NUMTWO+1        ; Store the high byte of the initial loop value
          STA (NUMONE)

          INC FORSNUM         ; Increment stack count (we just pushed an item on it)

          RTS                 ; All done

_SYN      JMP RAISE_SYN       ; Raise syntax error

_SYS      JMP RAISE_SYS       ; Indicate the FOR-NEXT stack is out of memory
```

***EXFOR** handles the beginning of a **FOR**-**NEXT** loop.*

Surprisingly, much of the **FOR** loop is actually handled by the **NEXT** statement. When a **NEXT** statement is executed, it checks to see if the value in the loop's variable is equal to the stop value. If so, the loop is done and popped from the for-next stack, while control proceeds to the next line. If it's not equal, the variable is incremented by one and **PROGNXT** updated with the first line in the loop's body, similar to how a **RETURN** statement works. A sanity check also ensures that a matching **FOR** exists.

```
EXNEXT    JSR ONLYRUN         ; Only valid in RUN mode

          LDX FORSNUM         ; Load the number of FOR/NEXT entries (logic error if none)
          BEQ _LOG

          LDA FORVARS_L-1,X   ; Assemble the variable address from the low and high bytes
          STA MEMSPTR
          LDA FORVARS_H-1,X
          STA MEMSPTR+1

          LDY #Ø              ; Compare low bytes
          LDA (MEMSPTR),Y
          CMP FORSTOP_L-1,X
          BNE _LOOP

          INY                 ; Compare high bytes
          LDA (MEMSPTR),Y
          CMP FORSTOP_H-1,X
          BNE _LOOP

          DEC FORSNUM         ; This loop is done, remove it from the stack

          BRA _DONE           ; All done here

_LOOP     CLC                 ; Prepare to increment the variable by one

          LDY #Ø              ; Increment low byte
          LDA (MEMSPTR),Y
          ADC #1
          STA (MEMSPTR),Y

          INY                 ; Increment high byte (with carry)
          LDA (MEMSPTR),Y
          ADC #Ø
```

```
          STA (MEMSPTR),Y

          LDA FORLINE_L-1,X   ; Copy the top item on the FOR/NEXT stack as our next line to ru
          STA PROGNXT
          LDA FORLINE_H-1,X
          STA PROGNXT+1

_DONE     RTS                 ; All done

_LOG      JMP RAISE_LOG       ; Indicate a NEXT-without-FOR error
```

*Much of the loop is actually implemented by **EXNEXT**.*

DATA AND READ STATEMENTS

Cody BASIC supports a form of the **RESTORE**, **DATA**, and **READ** statements common to many 8-bit BASIC dialects. A **DATA** statement specifies comma-delimited number literals that can be read into variables using the **READ** statement. When data is to be read, the interpreter starts at the top of the program, going through each line until a new **DATA** statement is found.

To repeat the process from the beginning, the **RESTORE** statement can be called to move the current data pointer back to the beginning of the program. In many respects the behavior is a number-only subset of the **DATA** statements in Commodore BASIC.

Some zero-page variables and memory locations are very important to the processing of **DATA** statements. The **DATAPTR** variable points to the next line to search for data. Because the content read from **DATA** statements is stored in a buffer until it is read, **DBUFL** and **DBUFH** point to the start of storage for the data's low and high bytes respectively. **DBUFLEN** stores the number of items held in the current data buffer, while **DBUFPOS** stores the current index within the buffer for **READ** statements.

Loading data begins with the **MOREDATA** routine, which is called whenever a **READ** statement needs data and the buffer is empty. **MOREDATA** starts at the current **DATAPTR** and continues until a line with a **DATA** statement is found. If a matching **DATA** statement is found, the numbers in that statement are parsed and stored in **DBUFL** and **DBUFH**.

Because parsing a **DATA** statement is in some ways similar to the parsing of any other statement, the routine temporarily replaces **PROGPTR** with the current value of **DATAPTR** to reuse some of the existing routines. When a **DATA** statement is encountered during the normal interpretation of a program, it's skipped over entirely. **DATA** statements only get processed when a call to **READ** needs more data and reading has advanced to a given line.

```
MOREDATA  LDA PROGPTR         ; Preserve the current program pointer
          PHA
          LDA PROGPTR+1
          PHA

          LDA PROGOFF         ; Preserve the current program line offset
          PHA

          LDA DATAPTR         ; Temporarily use the line pointer as the data pointer
          STA PROGPTR
          LDA DATAPTR+1
          STA PROGPTR+1

_LINE     JSR ISEND           ; Are we at the end of the program?
          BNE _LINEOK

          JMP _DONE           ; End of program (need JMP because of distance)

_LINEOK   LDA #4              ; Start after line number in the current line
          STA PROGOFF

          JSR EXSKIP          ; Skip whitespace

          LDY PROGOFF         ; Read the next token
          LDA (PROGPTR),Y
          INC PROGOFF

          CMP #TOK_DATA       ; If a DATA statement, process the line
          BEQ _LOOP

          JSR _NXTLINE        ; Otherwise go to the next line

          BRA _LINE

_LOOP     JSR EXSKIP          ; Skip whitespace

          LDY PROGOFF         ; Load the next character from the current line
```

```
          LDA (PROGPTR),Y

          INY                 ; Consume number token symbol

          CMP #CHR_NL         ; Newline means we're done
          BEQ _EOL

          CMP #CHR_MINUS      ; Minus means a negative number
          BEQ _NEG

          CMP #TOK_NUM        ; Otherwise just a number (or a syntax error)
          BNE _SYN

_POS      LDX DBUFLEN         ; Load the current data buffer length

          LDA (PROGPTR),Y     ; Store data low byte
          STA DBUFL,X
          INY

          LDA (PROGPTR),Y     ; Store data high byte
          STA DBUFH,X
          INY

          BRA _NXT            ; Next number in list

_NEG      STY PROGOFF         ; Update program offset

          JSR EXSKIP          ; Skip any trailing space after the minus sign

          LDY PROGOFF         ; Load the next character from the current line
          LDA (PROGPTR),Y

          CMP #TOK_NUM        ; Must be a number
          BNE _SYN
          INY

          LDX DBUFLEN         ; Load the current data buffer length

          SEC                 ; Prepare to subtract

          LDA #Ø              ; Subtract low byte from zero and store in buffer
          SBC (PROGPTR),Y
          STA DBUFL,X
          INY

          LDA #Ø              ; Subtract high byte from zero and store in buffer
          SBC (PROGPTR),Y
          STA DBUFH,X
          INY

_NXT      STY PROGOFF         ; Update program offset

          INC DBUFLEN         ; Update data buffer length (overflow shouldn't happen)

          JSR EXSKIP          ; Skip any trailing space after the number

          LDY PROGOFF         ; Read and consume the next character in the line
          LDA (PROGPTR),Y
          INC PROGOFF

          CMP #CHR_NL         ; Newline means we're done
          BEQ _EOL

          CMP #CHR_COMMA      ; Otherwise it needs to be a comma
          BNE _SYN

          BRA _LOOP           ; Next data value in list

_EOL      JSR _NXTLINE

_DONE     PLA                 ; Restore the program line offset
          STA PROGOFF

          PLA                 ; Restore the program pointer
          STA PROGPTR+1
          PLA
          STA PROGPTR+Ø
```

```
        RTS

_SYN    JMP RAISE_SYN

_NXTLINE CLC                ; Move to the next line by adding the line length

        LDA PROGPTR
        ADC (PROGPTR)
        STA PROGPTR
        STA DATAPTR

        LDA PROGPTR+1
        ADC #Ø
        STA PROGPTR+1
        STA DATAPTR+1

        RTS
```

MOREDATA *fills the data buffer with more data when called.*

The **EXREAD** routine implements the read functionality. It loops over one or more variables, attempting to populate each of the variables with data. When the data buffer is empty (**DBUFLEN** is zero), it calls **MOREDATA** to read more data. If nothing is found, an out of data error condition exists. On the other hand, if data was found and stored in the buffer, it begins copying data out of the buffer and into the variable list.

```
EXREAD

_LOOP   JSR EXVAR           ; Read the variable to read into, it has to be a number variable
        BCS _SYN

        LDA DBUFLEN         ; Verify that we still have data in the buffer to read
        BNE _READ

        STZ DBUFPOS         ; Out of data, need to read more in from the program
        JSR MOREDATA

        LDA DBUFLEN         ; Did we find any more data in the program?
        BEQ _LOG

_READ   JSR POPONE          ; Pop the variable address into NUMONE

        LDX DBUFPOS         ; Read current index in the data buffer

        LDA DBUFL,X         ; Copy low byte
        STA (NUMONE)

        INC NUMONE          ; Move on to high byte (relies on page alignment)

        LDA DBUFH,X         ; Store high byte
        STA (NUMONE)

        DEC DBUFLEN         ; Decrement data buffer size and increment buffer position
        INC DBUFPOS
```

```
          JSR EXSKIP          ; Skip any whitespace

          LDY PROGOFF         ; Load the next character from the current line
          LDA (PROGPTR),Y

          CMP #CHR_NL         ; Newline means we're done with this statement
          BEQ _DONE

          CMP #CHR_COMMA      ; If it's not a comma then it's a syntax error
          BNE _SYN

          INC PROGOFF         ; Consume the comma

          BRA _LOOP           ; Next variable

_DONE     RTS

_SYN      JMP RAISE_SYN
_LOG      JMP RAISE_LOG
```

***EXREAD** implements the **READ** statement.*

For the last statement in this group, the **RESTORE** statement, the **EXRESTORE** routine is called. However, **EXRESTORE** only calls the **RESTORE** routine already used when a program is being run. It resets the **DBUFLEN** and **DBUFPOS** to zero, then moves the **DATAPTR** to the start of program memory.

```
RESTORE   STZ DBUFLEN         ; Reset data buffer positions
          STZ DBUFPOS

          LDA #<PROGMEM       ; Move data line pointer to start of program
          STA DATAPTR+0
          LDA #>PROGMEM
          STA DATAPTR+1

          RTS
```

***RESTORE** resets the handling of **DATA** statements.*

INPUT AND OUTPUT STATEMENTS

Cody BASIC supports input and output similar to many other BASIC dialects. **INPUT** and **PRINT** statements handle generic input and output. **OPEN** and **CLOSE** statements select either the keyboard and screen or a serial port as the current I/O device. Within the BASIC interpreter there are several

routines that work together to implement input and output.

Input and output in Cody BASIC, much like Tiny BASIC, is line-based, with two buffers set up to store input data and output data. **IBUF** is an input buffer that stores up to 255 characters read from the keyboard or a serial port. **OBUF** is an output buffer that also stores 255 characters to be printed to the screen or sent to a serial port. The length of the contents of each buffer are stored in **IBUFLEN** and **OBUFLEN**.

The I/O routines support a combined keyboard-screen device and the Cody Computer's two serial ports. Two zero page variables, **IOMODE** and **IOBAUD**, contain the current I/O mode (the device) and a value representing the baud rate (only used for serial ports). These are set either by code internal to the interpreter (such as when loading or saving programs) or by user code in the BASIC program.

OPEN AND CLOSE STATEMENTS

The **OPEN** and **CLOSE** statements are used to redirect input and output to specific devices, either the screen/keyboard combination (in the default case) or one of the Cody Computer's two serial ports.

The **OPEN** statement is implemented by the **EXOPEN** routine. It sets the **IOMODE** and **IOBAUD** values to configure the input and output. If a serial port is selected, it also calls the **SERIALON** routine to set up the UART for the selected serial device.

```
EXOPEN    JSR ONLYRUN         ; Only valid in RUN mode

          JSR EXEXPR          ; Read device number

          JSR EXCOMMA         ; Comma separator

          JSR EXEXPR          ; Baud rate (1 through 15)

          JSR POPBOTH         ; Get both values off the stack

          LDA NUMTWO          ; Baud rate (1 through 15)
          STA IOBAUD

          LDA NUMONE          ; Device number
          STA IOMODE

          BEQ _DONE           ; If a UART was selected turn serial on
          JSR SERIALON

_DONE     RTS
```

*The **EXOPEN** routine configures input and output.*

The **CLOSE** statement is implemented by the **EXCLOSE** routine. It calls **SERIALOFF** to disable the UART for the selected serial port (for keyboard/screen operation this reduces to a no-op). Once the UART is shut down, it clears out the **IOMODE** and **IOBAUD** variables to return input and output to the keyboard and screen.

```
EXCLOSE   JSR ONLYRUN         ; Only valid in RUN mode

          JSR SERIALOFF       ; Turn serial off (routine should check if IOMODE is actually se

          STZ IOMODE          ; Clear IO mode and IO baud settings (defaults back to screen/ke
          STZ IOBAUD

          RTS
```

*The **EXCLOSE** routine restores I/O to the screen and keyboard.*

PRINT STATEMENTS

The **EXPRINT** routine handles a **PRINT** statement to write text to the screen. It accepts string expressions that are stored in the output buffer and later written to

the current I/O device's output via **FLUSH**. It also supports some control codes and format specifiers to handle clearing the screen, changing text colors, aligning text, and moving the cursor, though these are only relevant when the screen is the output device. Some of the functionality for these features is actually implemented in the screen routines rather than in **EXPRINT** itself.

```
EXPRINT   STZ OBUFLEN         ; Start at beginning of output buffer

_LOOP     JSR EXSKIP          ; Skip any leading space

          LDY PROGOFF         ; Load the next character in the current line
          LDA (PROGPTR),Y

          CMP #TOK_AT         ; "AT()" format specifier to change screen location
          BEQ _AT

          CMP #TOK_TAB        ; "TAB() format specifier to advance position in line
          BEQ _TAB

          CMP #CHR_QUOTE      ; Quote means a string expression
          BEQ _STR

          CMP #TOK_STR        ; "STR$" function means a string expression
          BEQ _STR

          CMP #TOK_CHR        ; "CHR$" function means a string expression
          BEQ _STR

          CMP #TOK_SUB        ; "SUB$" function means a string expression
          BEQ _STR

          CMP #CHR_NL         ; Newline means the end of the line
          BEQ _ADV

          CMP #CHR_SEMICOLON  ; Semicolon means the end of the line without advancing
          BEQ _END

          JSR ISALPHA         ; At this point, the only possibility left is a string variable
          BEQ _NUM

          INY                 ; Look ahead one character
          LDA (PROGPTR),Y

          CMP #CHR_DOLLAR     ; String variables end with a dollar sign ("$")
          BEQ _STR
```

*Excerpt from **EXPRINT** showing possible arguments.*

When the statement is done, it sends its output via the **FLUSH** routine. **FLUSH** goes over the contents in the output buffer **OBUF** and sends them to the current IO device. It checks the current value of **IOMODE** and calls either **SCREENPUT** or **SERIALPUT** to print out the

individual characters in the buffer. Other routines that populate the output buffer also call **FLUSH** to print out the contents.

```
FLUSH     PHA                   ; Preserve registers
          PHX
          PHY

          LDY IOMODE            ; We'll be checking the IO mode a lot

          LDX #0                ; Start at the beginning

_LOOP     CPX OBUFLEN           ; Check that we have more characters to print
          BEQ _END

          LDA OBUF,X            ; Load the next character from the output buffer
          INX

          CPY #0                ; Determine whether to use screen or serial output
          BEQ _SCREEN

_SERIAL   JSR SERIALPUT         ; Print it to the serial port (current UART)
          BRA _LOOP

_SCREEN   JSR SCREENPUT         ; Print it on the screen
          BRA _LOOP

_END      STZ OBUFLEN           ; Clear the length of the output buffer (we're empty now)

_NOOFF    PLY                   ; Restore registers
          PLX
          PLA

          RTS                   ; All done
```

*The **FLUSH** routine writes the output buffer to the current output.*

INPUT STATEMENTS

The **EXINPUT** routine implements the internals for Cody BASIC's **INPUT** statement. It reads a line of input from the current I/O device into the input buffer and then attempts to parse it into the variable list passed to the statement. Both numbers and strings are supported. As part of its operations, the routine has to check the current I/O mode and call either **READKBD** or **READSER** depending on the mode.

```
_READ     LDA IOMODE          ; Determine where to read from
          BEQ _KBD

_SER      JSR READSER         ; Read our input line from the UART
          BRA _INP

_KBD      JSR READKBD         ; Read out input line from the keyboard
```

*Portion of **EXINPUT** selecting the input source.*

Unlike the common **FLUSH** routine for sending out printed output, no similar single routine for reading input exists. Instead, the **READKBD** routine populates the input buffer **IBUF** from keyboard input, updating the screen contents as the user types. This routine relies on a variety of other routines related to screen output and keyboard scanning covered elsewhere in this chapter.

```
READKBD   PHA                 ; Preserve registers
          PHX

          LDX #Ø              ; Start at beginning of input buffer

_NEXT     LDA JIFFIES

_WAIT     JSR BLINK           ; Wait for jiffies to change to know we got a new keyboard sca
          CMP JIFFIES
          BEQ _WAIT

          JSR KEYDECODE       ; Decode whatever key was pressed (if anything)

          LDA KEYCODE         ; Debounce keys by making sure we read the same code twice in
          CMP KEYDEBO
          STA KEYDEBO
          BNE _NEXT

          LDA KEYCODE         ; Suppress repeated key presses by comparing to last key read
          CMP KEYLAST
          STA KEYLAST
          BEQ _NEXT

          CMP #$6Ø            ; Check for CODY + META (shift lock) toggle
          BEQ _TOG

          BIT #$1F            ; Suppress key codes when no keys (aside from modifiers) were
          BEQ _NEXT

          JSR KEYTOCHR        ; Convert key code to CODSCII code and preserve on stack
          PHA

          LDA KEYLOCK         ; Check if the shift lock is set
          BEQ _KEY

          PLA                 ; Convert CODSCII code to lowercase
          JSR TOLOWER
          PHA

_KEY      PLA                 ; Restore keyboard CODSCII code from stack
```

```
          CMP #CHR_CAN          ; Skip cancel character
          BEQ _NEXT

          CMP #CHR_BS           ; Check for backspace character
          BEQ _DEL

          CPX #$FE              ; Check for space to store character
          BEQ _NEXT

          STA IBUF,X            ; Put the character in the buffer
          INX

          CMP #CHR_NL           ; Check for newline character (end of line)
          BEQ _DONE

          JSR SCREENPUT         ; Echo to the screen

          BRA _NEXT

_DEL      CPX #0                ; Check that we have something in the buffer to delete
          BEQ _NEXT

          DEX                   ; Back up one position the buffer and remove the char from the
          JSR SCREENDEL

          BRA _NEXT

_TOG      LDA KEYLOCK           ; Toggle shift lock
          EOR #$01
          STA KEYLOCK

          BRA _NEXT

_DONE     STX IBUFLEN           ; Update input buffer length

          LDA #20               ; TODO: CLEAR BLINKING CURSOR (MAKE THIS BETTER, ALSO SEE ABOV
          STA (CURSCRPTR)

          PLX                   ; Restore registers
          PLA

          RTS
```

*The **READKBD** routine reads a line from the keyboard.*

For serial operations, the **READSER** routine will populate **IBUF** with the contents read from the serial port's UART. The routine stops when a carriage return or newline character are read from the serial input. This is essentially the serial equivalent of the **READKBD** routine. It relies on the serial routines covered later in the chapter.

```
READSER   PHA
          PHX

          LDX #0                ; Start at beginning of buffer

_READ     JSR SERIALGET         ; Poll for next character
          BCC _READ

          STA IBUF,X            ; Store the character and increment the buffer position
          INX
```

```
          CPX #$FE              ; Do we still have space in the buffer?
          BCS _SYS

          CMP #CHR_NL           ; Newline characters can be an end of line
          BEQ _DONE

          CMP #CHR_CR           ; Carriage return characters can be an end of line
          BEQ _DONE

          BRA _READ             ; Continue

_DONE     STX IBUFLEN           ; Store the input line length

          PLX
          PLA

          RTS

_SYS      JMP RAISE_SYS         ; Indicate we're out of space in the input buffer
```

READSER *uses serial routines to read a line of text from a UART.*

LOADING AND SAVING PROGRAMS

Cody BASIC supports the **LOAD** and **SAVE** commands for loading and saving programs. With the exception of loading binary programs over the serial port or from a cartridge, load and save operations rely almost entirely on other functionality in Cody BASIC.

When loading a BASIC program, input is redirected from the serial port, and each incoming line is tokenized as though the user had typed the program in. When saving a program, output is redirected to the serial port, and the program is listed as though a **LIST** command had been executed.

LOAD STATEMENTS

The **EXLOAD** routine implements the BASIC portion of **LOAD** statements. It parses parameters containing the device number and mode before calling the appropriate routine to do the operation. In the event

that the program to be loaded is a BASIC program, it calls **LOADBAS**, and for binary programs, it calls **LOADBIN** instead.

```
EXLOAD    JSR ONLYREPL        ; Only valid in REPL mode

          LDA #RM_COMMAND     ; Running without a line number so we can break
          STA RUNMODE

          JSR EXEXPR          ; Device argument

          JSR EXCOMMA         ; Comma separator

          JSR EXEXPR          ; Mode argument (0 for BASIC, 1 for binary)

          JSR POPBOTH         ; Pop results

          LDA #$F             ; Read at 19200 baud
          STA IOBAUD

          LDA NUMONE          ; Use device number as UART number
          STA IOMODE

          LDA NUMTWO          ; Read BASIC or binary file as appropriate
          BNE _BIN

_BAS      JSR LOADBAS         ; Load the BASIC program

          STZ RUNMODE         ; Reset run mode and return
          RTS

_BIN      JMP LOADBIN
```

EXLOAD *implements the* ***LOAD*** *statement in Cody BASIC.*

LOADBAS loads BASIC programs over the serial port. Each line is read into the input buffer **IBUF** just as a user would enter the code line by line, with each line being tokenized and appended at the end of the program. When the routine encounters a line with no characters, it considers the load completed and returns to the REPL loop.

Unlike many other 8-bit systems, Cody BASIC doesn't save its BASIC programs in their tokenized format. This makes it easier to exchange BASIC files with other computers, but it also makes it slower to load because of the retokenization. As the speed of tokenization is the main limit to loading programs quickly, optimization of the tokenizer is very

important. This also means that terminal programs talking to the Cody Computer usually need to insert a delay after each line so that the tokenizer can keep up.

Some simple optimizations and sanity checks are added to this code path to speed up loading and guard against obvious errors (such as out-of-order line numbers). Much like what happens when input statements are redirected to serial, **LOADBAS** sends a question-mark character before waiting for each incoming line. If the device sending the program recognizes this, it can immediately skip to the program's next line rather than waiting for a fixed period for each line.

```
LOADBAS   JSR NEWPROG         ; Clear out the current program

          STZ LINENUM         ; Start at "line zero" as the first line
          STZ LINENUM+1

          JSR SERIALON        ; Turn serial port on

_LOOP     LDA #CHR_QUEST      ; Send question mark prompt (for more advanced loaders)
          JSR SERIALPUT

          JSR READSER         ; Read a line of input

          LDX IBUFLEN         ; Make sure we actually read a full line
          CPX #2
          BCC _DONE

          DEX                 ; Replace trailing character with a newline (could be a carriage
          LDA #CHR_NL
          STA IBUF,X

          JSR TOKENIZE        ; Tokenize the line

          LDA TBUF            ; Basic validity check (must start with line number)
          CMP #$FF
          BNE _SYS

          LDA TBUF+2          ; Another validity check (ensure line numbers ascending)
          CMP LINENUM+1
          BNE _LINE

          LDA TBUF+1
          CMP LINENUM
          BEQ _SYS
_LINE     BCC _SYS

          LDA PROGTOP         ; Set destination as the top of the program
          STA LINEPTR
          LDA PROGTOP+1
          STA LINEPTR+1

          JSR INSLINE         ; Insert the line into the program

          LDA TBUF+1          ; Update last line number for future tests
          STA LINENUM
```

```
          LDA TBUF+2
          STA LINENUM+1

          BRA _LOOP           ; Read the next line

_DONE     JSR SERIALOFF       ; Turn off serial port

          STZ IOMODE          ; Clear I/O settings back to screen/keyboard
          STZ IOBAUD

          STZ RUNMODE         ; Not "running" any more

          RTS

_SYS      JMP RAISE_SYS       ; Indicate IO error during read
```

*The **LOADBAS** routine loads a BASIC program into memory.*

For loading binary files, the **LOADBIN** routine is used instead. Loading a binary file is somewhat easier as it's essentially a direct read of bytes into the Cody Computer's memory, followed by a jump to the loading address. Because binary programs can be loaded from the serial ports (in BASIC) or from a cartridge (on system startup), **LOADBIN** has to take into account both possibilities. It also supports returning to BASIC at the end of a binary program, but the results may vary depending on the state the computer was left in. However, this permits carefully-written binary programs to remain resident in memory to extend the system or for later use in BASIC code.

The Cody Computer's binary format is simple. Two bytes contain the start address, two bytes contain the end address, and the remainder consists of raw bytes for the program. To load the program the computer needs only to point a destination pointer at the start address, read and store a byte, and continue reading until the destination pointer equals the end address.

```
LOADBIN   LDA IOMODE
          BEQ _INITSPI

_INITSER  JSR SERIALON        ; Start running serial port

          BRA _LOAD
```

```
_INITSPI  JSR CARTON          ; Begin SPI transaction

          LDA #$03            ; Command 3 to begin reading
          JSR CARTXFER

          LDX #2              ; Assume a cartridge with a two-byte address

          LDA VIA_IORB        ; If cart size bit is high, we have a three-byte address
          BIT #CART_SIZE
          BEQ _ADDR
          INX

_ADDR     LDA #$00            ; Send the appropriate number of zeroed address bytes
          JSR CARTXFER
          DEX
          BNE _ADDR

_LOAD     JSR _READ           ; Read starting address (low and high bytes)
          STA MEMSPTR
          STA PROGPTR

          JSR _READ
          STA MEMSPTR+1
          STA PROGPTR+1

          JSR _READ           ; Read ending address (low and high bytes)
          STA MEMDPTR

          JSR _READ
          STA MEMDPTR+1

_LOOP     JSR _READ           ; Read and store another byte
          STA (MEMSPTR)       ; Store it in memory

          LDA MEMSPTR         ; If not at the destination address, read another byte
          CMP MEMDPTR
          BNE _INCR

          LDA MEMSPTR+1
          CMP MEMDPTR+1
          BNE _INCR

          LDA IOMODE          ; Finished loading, shutdown for SPI vs serial is different
          BEQ _DONESPI
          BNE _DONESER

_INCR     INC MEMSPTR         ; Increment source pointer by one
          BNE _LOOP
          INC MEMSPTR+1
          BRA _LOOP

_DONESER  JSR SERIALOFF       ; Stop running serial port

          STZ IOMODE          ; Clear I/O settings back to screen/keyboard
          STZ IOBAUD

          BRA _DONE

_DONESPI  JSR CARTOFF

_DONE     STZ RUNMODE         ; Ensure run mode is zero before jumping to loaded binary

          SEI                 ; Disable interrupts for BASIC (keyboard scan and clock)

          LDX STACKREG        ; Roll back the BASIC stack
          TXS

          JSR _JUMP

          JMP BASIC           ; If it returns for some reason, restart BASIC and hope

_JUMP     JMP (PROGPTR)       ; Jump to the load address (indirect JSR workaround)

_READ     LDA IOMODE          ; Determine what mode we're running in
          BNE _READSER
```

```
_READSPI  LDA #$00          ; Read value and return as accumulator
          JSR CARTXFER
          RTS

_READSER  JSR SERIALGET     ; Busy-wait for another byte
          BCC _READSER
          RTS
```

READBIN *loads binary programs from serial ports or cartridges.*

SAVE STATEMENTS

Saving programs is somewhat more straightforward because Cody BASIC only supports saving the current BASIC program in memory as text. No provision is mode for dumping an arbitrary region of memory to serial output as raw bytes, and BASIC programs can only be saved to serial ports, not cartridges.

To save a program, output is redirected to one of the serial ports, the entire program is listed by calling **LISTPROG**, and a blank line is written to mark the end of the program. Because of its overall simplicity this is entirely implemented in the **EXSAVE** routine used by the interpreter.

```
EXSAVE    JSR ONLYREPL      ; Only valid in REPL mode

          LDA #RM_COMMAND   ; Running without a line number so we can break
          STA RUNMODE

          JSR EXEXPR        ; Read the device number for the UART
          JSR POPONE

          LDA NUMONE        ; Use it as the UART number
          STA IOMODE

          LDA #$F           ; Save at 19200 baud
          STA IOBAUD

          LDA #<PROGMEM     ; Start at the beginning of program memory
          STA LINEPTR
          LDA #>PROGMEM
          STA LINEPTR+1

          LDA PROGTOP       ; Stop at the top of program memory
          STA STOPPTR
          LDA PROGTOP+1
          STA STOPPTR+1

          JSR SERIALON      ; Start the serial port
```

```
        JSR LISTPROG        ; List the program out the serial port to "save" it

        STZ OBUFLEN         ; Write an empty line to mark the end (the loader expects this!)
        LDA #CHR_NL
        JSR PUTOUT
        JSR FLUSH

        JSR SERIALOFF       ; Stop the serial port

        STZ RUNMODE         ; Reset run mode

        STZ IOBAUD          ; Go back to screen/keyboard IO when we're done
        STZ IOMODE

        RTS
```

***EXSAVE** is a short routine that implements the **SAVE** command.*

Most of the actual work in saving a program is done by the **LISTPROG** routine. This same routine is also called when a user enters the **LIST** statement at the BASIC prompt, except that in this case we're listing the program to a serial port instead. **LISTPROG** works opposite to a tokenizer, starting at the beginning of the BASIC program, going through each tokenized line, and looking up the actual values of each token to put them into the output buffer. Once an entire line is decoded, it's flushed to the current output device.

```
LISTPROG  PHA                 ; Preserve registers
          PHX
          PHY

_LOOP     LDA LINEPTR+0       ; Always do a sanity check (data can come from LIST)
          CMP PROGTOP+0
          BNE _SANE

          LDA LINEPTR+1
          CMP PROGTOP+1
          BNE _SANE

          BRA _DONE

_SANE     LDA LINEPTR+0       ; Are we at the line we're supposed to stop at?
          CMP STOPPTR+0
          BNE _LINE

          LDA LINEPTR+1
          CMP STOPPTR+1
          BNE _LINE

_DONE     PLY                 ; No more lines in program, restore registers
          PLX
          PLA

          RTS                 ; All done

_LINE     STZ OBUFLEN         ; Start at the beginning of the output buffer
```

```
        LDY #1              ; Start at beginning of line (skipping line length byte)

        LDA (LINEPTR),Y     ; Copy line number low byte
        STA NUMONE+Ø
        INY

        LDA (LINEPTR),Y     ; Copy line number high byte
        STA NUMONE+1
        INY

        JSR TOSTRING        ; Write the number's digits to the output buffer

_PART   LDA (LINEPTR),Y     ; Load the next byte in the line

        CMP #$FF            ; Do we have a number token?
        BEQ _NUM

        BIT #$8Ø            ; Do we have a token to decode?
        BNE _TOK

        JSR PUTOUT          ; Normal character, put it into the output buffer
        INY

        CMP #CHR_NL         ; If it was a newline, move on to the next source line
        BEQ _NEXT

        BRA _PART           ; Next part of the current line

_TOK    AND #$7F            ; Mask out the number of the actual token

        CLC                 ; Adjust the token number into the message table
        ADC #MSG_TOKENS

        JSR PUTMSG          ; Put the token's text into the output buffer

        INY                 ; Consume the token

        BRA _PART           ; Next part of the current line

_NUM    INY                 ; Skip leading number token tag

        LDA (LINEPTR),Y     ; Copy integer low byte
        STA NUMONE+Ø
        INY

        LDA (LINEPTR),Y     ; Copy integer high byte
        STA NUMONE+1
        INY

        JSR TOSTRING        ; Print integer

        BRA _PART           ; Next part of the current line

_NEXT   JSR FLUSH           ; Flush the output buffer

        CLC                 ; Move the pointer to the next line
        LDA LINEPTR+Ø
        ADC (LINEPTR)
        STA LINEPTR+Ø
        LDA LINEPTR+1
        ADC #Ø
        STA LINEPTR+1

        BRA _LOOP           ; Next line
```

LISTPROG *is used internally to both list and save programs.*

SERIAL ROUTINES

When input and output have been redirected to one of the serial ports (**IOMODE** of 1 or 2), serial routines are called to configure the appropriate UART and perform reads and writes. The **SERIALON** routine starts up the serial UART, **SERIALPUT** places a byte in its transmit buffer, **SERIALGET** reads a byte from its receive buffer, and **SERIALOFF** turns it off. Together these provide enough features to support Cody BASIC's line-based input and output when a serial port is enabled.

Because the register layout for each UART is identical, the relevant assembly code uses indirect addressing to access them. Either **UART1_BASE** or **UART2_BASE** is stored into the **UARTPTR** zero page variable when **SERIALON** is called, and all subsequent calls to serial routines use the specified pointer to access the current UART.

```
SERIALON  PHA
          PHY

          LDA IOMODE              ; What UART are we using?
          CMP #1
          BEQ _UART1
          BCS _UART2

          JMP RAISE_SYS           ; Indicate an IO error (should never happen!)

_UART1    LDA #<UART1_BASE        ; Running UART 1
          STA UARTPTR
          LDA #>UART1_BASE
          STA UARTPTR+1

          BRA _INIT

_UART2    LDA #<UART2_BASE        ; Running UART 2
          STA UARTPTR
          LDA #>UART2_BASE
          STA UARTPTR+1

_INIT     LDA #Ø

          LDY #UART_RXTL          ; Clear out buffer registers
          STA (UARTPTR),Y

          LDY #UART_TXHD
```

```
         STA (UARTPTR),Y

         LDA IOBAUD              ; Set baud rate
         AND #$0F
         LDY #UART_CNTL
         STA (UARTPTR),Y

         LDA #01                 ; Enable UART
         LDY #UART_CMND
         STA (UARTPTR),Y

         LDY #UART_STAT          ; Wait for UART to start up
_WAIT    LDA (UARTPTR),Y
         AND #$40
         BEQ _WAIT

         PLY
         PLA

         RTS                     ; All done
```

SERIALON *configures a UART to transmit and receive.*

Turning off serial communications is somewhat simpler, as it only waits for any pending bytes to be transmitted and then turns off the UART. The check for transmitting data is a two-step process, ensuring that the transmit buffer is empty, then checking to ensure no byte is currently stored and being sent out.

```
SERIALOFF PHA
          PHY

          LDA IOMODE             ; Special check in case this was called incorrectly
          BEQ _DONE

_WAITBUF  LDY #UART_TXHD         ; Wait for any pending characters to transmit
          LDA (UARTPTR),Y
          LDY #UART_TXTL
          CMP (UARTPTR),Y
          BNE _WAITBUF

          LDY #UART_STAT         ; Wait for any pending byte to be sent out
_WAITBIT  LDA (UARTPTR),Y
          AND #$10
          BNE _WAITBIT

_SHUTOFF  LDA #0
          LDY #UART_CMND
          STA (UARTPTR),Y        ; Clear bit to stop UART

          LDY #UART_STAT
_WAITOFF  LDA (UARTPTR),Y        ; Wait for UART to stop
          AND #$40
          BNE _WAITOFF

_DONE     PLY
          PLA

          RTS
```

SERIALOFF *turns off serial communication.*

To transmit data, the **SERIALPUT** routine is called with a single byte. The routine checks to see if there's room in the transmit ring buffer, and if not, blocks until a space exists in the buffer. Once a space exists, the byte is added to the buffer and the head position of the buffer incremented. Calling this routine when a UART is not running will cause the routine to block indefinitely once the buffer is full.

```
SERIALPUT PHA
          PHX
          PHY

          PHA                    ; Preserve character to store

_WAIT     LDY #UART_TXHD         ; Get current head position
          LDA (UARTPTR),Y

          INC A                  ; Increment by one (to test if overflow)
          AND #$07

          LDY #UART_TXTL         ; Compare to current tail position (equals means we overflow
          CMP (UARTPTR),Y
          BEQ _WAIT

          TAX                    ; Store new head position (we'll need it really soon)

          LDY #UART_TXHD         ; Use current head position to calculate offset
          CLC
          LDA (UARTPTR),Y
          ADC #UART_TXBF
          TAY

          PLA                    ; Store character in buffer
          STA (UARTPTR),Y

          LDY #UART_TXHD         ; Update head position
          TXA
          STA (UARTPTR),Y

          PLY
          PLX
          PLA

          RTS
```

*The **SERIALPUT** routine enqueues bytes for transmission.*

Receiving data is handled by the **SERIALGET** routine. It checks whether a byte exists in the receive ring buffer, and if so, copies the byte and increments the receive buffer's tail position to consume it. If no byte exists, the routine returns without any action being

taken. Because a value of zero would be valid, the 65C02's carry flag is used to indicate whether or not a byte was read. Unlike the **SERIALPUT** routine, this routine won't block if the UART wasn't turned on, but neither will it read any data.

```
SERIALGET PHY

          LDY #UART_STAT          ; Get current control register
          LDA (UARTPTR),Y

          BIT #$06                ; Test that no error bits are set
          BNE _SYS

          LDY #UART_RXTL          ; Get current tail position
          LDA (UARTPTR),Y

          LDY #UART_RXHD          ; Compare to head position
          CMP (UARTPTR),Y

          BEQ _EMPTY              ; If they match then the buffer is empty

          CLC                     ; Calculate the buffer position and read the character
          ADC #UART_RXBF
          TAY
          LDA (UARTPTR),Y

          PHA                     ; Keep the character around for later

          LDY #UART_RXTL          ; Update tail position since we read from the buffer
          LDA (UARTPTR),Y
          INC A
          AND #$07
          STA (UARTPTR),Y

          PLA                     ; Pull the character we read off the stack

          PLY
          SEC                     ; Set carry to indicate a character was read
          RTS

_EMPTY    PLY
          CLC                     ; Clear carry to indicate no character read
          RTS

_SYS      JMP RAISE_SYS           ; Indicate we detected an IO error
```

*The **SERIALGET** routine reads a byte from the receive buffer.*

SCREEN OUTPUT

Cody BASIC has a set of routines to handle text output to the screen. Similar in some ways to a terminal device, the routines not only display

characters but will move the cursor location, clear the screen, and change the foreground and background colors of text based on control codes. The **SCREENPUT**, **SCREENDEL**, **SCREENCLR**, **SCREENADV**, and **SCREENPOS** routines contain the necessary code for screen output.

Screen display routines share a few zero page variables that encapsulate the current state of screen output. The cursor position is actually represented two different ways. The **CURCOL** and **CURROW** zero-page variables contain the current x and y coordinates of the cursor, while the **CURSCRPTR** and **CURCOLPTR** values point to the corresponding positions in screen and color memory. Because the routines also allow changes to foreground and background colors, another zero-page variable, **CURATTR**, contains the current foreground and background colors to use for new output.

The **SCREENPUT** routine displays a single character on the screen at the current cursor position. It also takes into account special control codes that change the foreground and background colors or clear the screen, and must also account for scrolling the screen when the cursor reaches the bottom.

```
SCREENPUT CMP #CHR_CLEAR          ; Clear screen
          BEQ _CLR

          CMP #CHR_REVERSE        ; Reverse field
          BEQ _REV

          CMP #CHR_NL             ; Newline (advance screen)
          BEQ _NL

          CMP #$FØ                ; Foreground color special character
          BCS _FG

          CMP #$EØ                ; Background color special character
          BCS _BG
```

*Excerpt showing control codes handled by **SCREENPUT**.*

Like other screen routines, it also has to ensure that certain critical sections of code aren't changed by the timer interrupt, which could happen if the user attempts to break out of the program. If this happened at a particularly bad time, internal variables related to the cursor position could be corrupted. This would cause future output to be broken and could potentially have knock-on effects for the rest of the system, particularly if the values of the pointers are corrupted.

```
          PHP                     ; Store flags and disable interrupts (cursor/pointer updat
          SEI

          STA (CURSCRPTR)         ; Store the character in the screen buffer

          PHA                     ; Store the cursor attribute in the color memory buffer
          LDA CURATTR
          STA (CURCOLPTR)
          PLA

          INC CURSCRPTR+Ø         ; Increment screen memory location
          BNE _ATTR
          INC CURSCRPTR+1

_ATTR     INC CURCOLPTR+Ø         ; Increment color memory location
          BNE _DOIT
          INC CURCOLPTR+1

_DOIT     LDA CURCOL              ; Increment the cursor x position
          INC A
          STA CURCOL
          CMP #4Ø
          BNE _INT

          STZ CURCOL              ; Increment the cursor y position (when needed)
          LDA CURROW
          INC A
          STA CURROW
          CMP #25
```

```
          BNE _INT

          STZ CURCOL               ; Move the cursor to the start of the last row (0, 24)
          LDA #24
          STA CURROW

          PLP                      ; Out of critical section, copying memory can take a lot o

          JMP _SCR                 ; Jump to scroll the memory (moved outside to make branche

_INT      PLP                      ; Pull processor flags to re-enable the previous interrupt
```

*Critical section in **SCREENPUT** that writes a character.*

When the user is typing and wants to delete a character, we need to have a way to remove it from the screen. In this situation **SCREENDEL** is called, which clears the screen content for the cursor and the previous position. To ensure everything matches up, it also moves the cursor position and memory pointers back by one, also taking into consideration the possibility that the cursor went back an entire line. This routine is needed by **READKBD** when the user wants to delete part of their newly-typed input.

```
SCREENDEL PHA

          DEC CURCOL        ; decrement column
          BPL _DEL
          LDA #39           ; wrapped to previous column
          STA CURCOL
          DEC CURROW        ; decrement row since we wrapped around
          BPL _DEL
          STZ CURCOL        ; wrapped off screen, need to correct that
          INC CURROW
          BRA _DONE

_DEL      LDA #$20          ; clear current cursor position
          STA (CURSCRPTR)
          SEC               ; subtract one from the cursor pointer
          LDA CURSCRPTR+0
          SBC #1
          STA CURSCRPTR+0
          LDA CURSCRPTR+1
          SBC #0
          STA CURSCRPTR+1
          LDA #$20          ; replace the character with the current cursor attributes to clea
          STA (CURSCRPTR)

          LDA CURATTR       ; clear current cursor position
          STA (CURCOLPTR)
          SEC               ; subtract one from the cursor pointer
          LDA CURCOLPTR+0
          SBC #1
          STA CURCOLPTR+0
          LDA CURCOLPTR+1
          SBC #0
          STA CURCOLPTR+1
          LDA CURATTR       ; replace with the current cursor attributes to clear it
          STA (CURCOLPTR)
```

```
_DONE     PLA
          RTS
```

***SCREENDEL** deletes a character and handles related calculations.*

Other routines also exist to handle particular aspects of screen output. The **SCREENADV** routine advances the screen by a single line, while **SCREENPOS** moves the cursor position and memory pointers based on new column and row coordinates. **SCREENCLR** clears the contents of screen memory and sets the contents of color memory, also moving the cursor back to the top of the screen. These routines are used within the codebase to handle special output needs.

```
SCREENCLR PHA
          PHP                  ; Disable interrupts (critical section)
          SEI

          STZ CURCOL           ; Reset the cursor x and cursor y to (0, 0)
          STZ CURROW

          STZ TABPOS           ; Reset tab position

          LDA #<SCRRAM         ; Reset the cursor pointer to the start of text memory
          STA CURSCRPTR+0
          LDA #>SCRRAM
          STA CURSCRPTR+1

          LDA #<COLRAM         ; Reset the cursor color pointer to the start of color memory
          STA CURCOLPTR+0
          LDA #>COLRAM
          STA CURCOLPTR+1

          PLP                  ; Restore interrupts (critical section)

          LDA #<SCRRAM         ; Fill the contents of text memory with spaces
          STA MEMDPTR+0
          LDA #>SCRRAM
          STA MEMDPTR+1
          LDA #<1000
          STA MEMSIZE+0
          LDA #>1000
          STA MEMSIZE+1
          LDA #$20
          JSR MEMFILL

          LDA #<COLRAM         ; Fill the contents of color memory with the current attribute
          STA MEMDPTR+0
          LDA #>COLRAM
          STA MEMDPTR+1
          LDA #<1000
          STA MEMSIZE+0
          LDA #>1000
```

```
        STA MEMSIZE+1
        LDA CURATTR
        JSR MEMFILL

        PLA

        RTS
```

SCREENCLR *clears the screen and moves the cursor back to the top left.*

4

Assembly Instructions

INTRODUCTION

This chapter describes how to build your own Cody Computer, including the assembly of a small mechanical keyboard, the main printed circuit board, and the computer's case. Each part is broken out into its own section, and inside each section the assembly is broken into multiple steps. Photos are also provided to point out aspects of the assembly process. You should read the chapter in its entirety before beginning the build.

> Just because something worked well for me doesn't mean it will work as well for you. As you go through the build, you'll want to consider what you're doing and evaluate your own results. The Cody Computer is more like a garage kit, particularly with the 3D printing side, so you'll want to build accordingly.

NOTES ON 3D PRINTING

The Cody Computer is heavily dependent on 3D printing for its construction, so you will need to either print the parts yourself or find someone who can print them for you. When developing the Cody Computer we were able to print all the parts on a more or less stock Ender 3 Pro, with the only major modifications being a glass bed and an eventual extruder replacement.

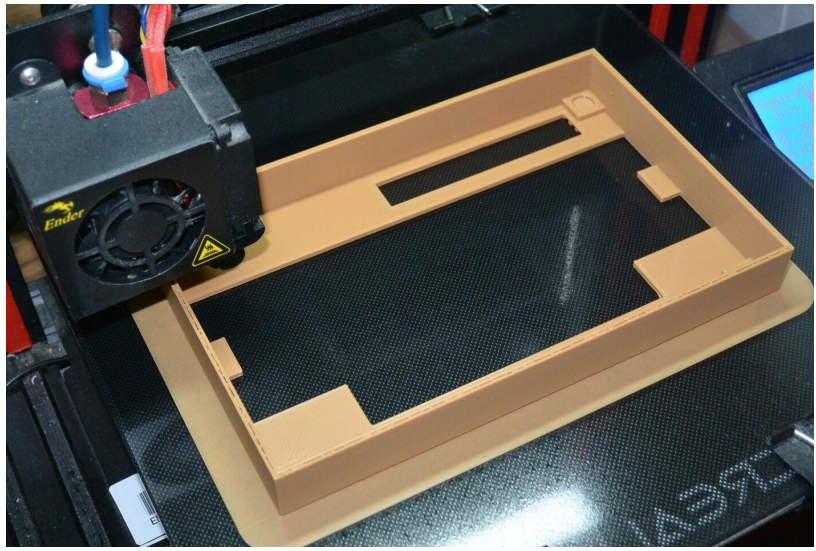
Because of differences between 3D printers, you may need to make adjustments to obtain suitable results. It's assumed your printer is dialed in with a reasonably high level of accuracy. If not you should be comfortable making your own adjustments to the printer and ensuring the fit of finished parts as they come off. The OpenSCAD design files are also provided if you need to make major adjustments to some of the dimensions for the build.

It's also worth planning the order in which you print the parts. One option is to print the parts for each step as needed, checking for proper fit at that time. Another option is to print the parts up front, perhaps even batching some of them together, and perform many of the basic test-fits up front as well. Whatever approach you use, make sure that you perform the test fits mentioned in the various assembly steps. If you decide to group your prints together by color, see the following:

- Black PLA filament (Hatchbox Black, Inland Black, or equivalent):
    - Alphanumeric keycaps (**KeycapA.stl** through **KeycapZ.stl**)
    - Cody keycap (**KeycapCody.stl**)
    - Meta keycap (**KeycapMeta.stl**)
    - Arrow keycap (**KeycapArrow.stl**)
    - Spacebar (**Spacebar.stl**)
    - Keyboard plate (**KeyboardPlate.stl**)
    - Case badge (**CaseBadge.stl**)
    - LED holder (**LEDHolder.stl**)
    - Left mounting bracket (**KeyboardBracketWithoutHoles.stl**)

    - Right mounting bracket (**KeyboardBracketWithHoles.stl**)
- Beige PLA filament (Inland Light Brown or equivalent):
    - Case top (**CaseTop.stl**)
    - Case bottom (**CaseBottom.stl**)
- White PLA (if using paint) or various color PLA:
    - Case badge inlay, red (**CaseBadgeInlay.stl**)
    - Case badge inlay, orange (**CaseBadgeInlay.stl**)
    - Case badge inlay, yellow (**CaseBadgeInlay.stl**)
    - Case badge inlay, green (**CaseBadgeInlay.stl**)
    - Case badge inlay, blue (**CaseBadgeInlay.stl**)

When printing consider the orientation of the parts on the print bed. For large pieces such as the case top and bottom, we printed them upside down to avoid the large overhead of supports for such pieces. The keyboard brackets were printed upright despite a need for some supports to avoid dimensionality problems for the magnet and screw pilot holes. Keycaps were printed face-down on a glass bed with good leveling to minimize gaps for later application of the air-dry clay.



A Creality Ender 3 Pro printing the Cody Computer's case top. Note the upside-down print orientation to avoid printing supports.

Also consider the infill and resolution settings when you run the STL files through your slicer. For parts with very specific dimensional requirements, such as the keycaps and their stems, use a standard or high resolution. For larger parts that take a long time and require significant strength, such as the case top and bottom, consider a lower resolution or draft print. You will want to take into account your own printer's characteristics and your tolerance for long builds when making such decisions.

KEYBOARD ASSEMBLY

Your first step in building the Cody Computer is to assemble its keyboard module. It's a good place to start because it combines all the things you'll need to

do in later steps, from 3D printing (with reasonably tight tolerances) to soldering up a circuit board.

If you have any problems in this step, it may indicate that you want to work them out before going on to later steps. For example, if your printer isn't calibrated enough or you need to make your own adjustments to the design files, there's a good chance you'll find that out here. Likewise, if you run into problems with soldering, it's better to solve those problems now before you start soldering the main logic board. In general, the keyboard is going to be a lot more forgiving of mistakes.

MAKING THE KEYCAPS

In this step we'll print out and make the keycaps for the keyboard. The keycaps have Cherry MX compatible stems, but they have a smaller spacing, so you can't use standard keycaps with the Cody Computer. There are 30 keycaps including a spacebar key.

Many early computer keycaps were manufactured using "double-shot" injection moulding. This meant that one color of plastic was shot into the mould for the keycap itself, while a second color of plastic was shot into the mould for the legend on it. You can do something similar with 3D printing in multiple colors (and we actually did that as well), but we obtained the best results using air-dry clay deposited into recessed legends in the 3D printed keycaps.

Before your get too far into the build process, it's a good idea to print a single keycap and test the fit against one of the Cherry MX switches if you haven't done so already. If adjustments are needed to your printer or to the OpenSCAD models to work with your printer or keyswitches, you want to do that before you've made a useless set of keycaps.

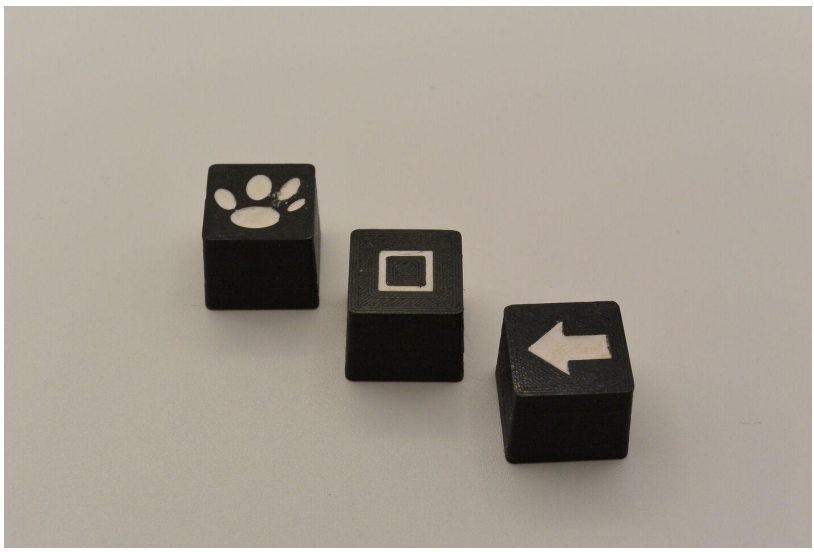
For this step, you'll need the following:

- 26 alphanumeric keycaps (**KeycapA.stl** through **KeycapZ.stl**)
- 1 Cody keycap (**KeycapCody.stl**)
- 1 Arrow keycap (**KeycapArrow.stl**)
- 1 Meta keycap (**KeycapMeta.stl**)
- 1 Spacebar keycap (**Spacebar.stl**)
- White air-dry clay (Sculpey Air-Dry or equivalent)
- Wet cloth
- Dry cloth

Before beginning the assembly, wash and dry the keycaps. This will help the air-dry clay adhere to the plastic. Once the keycaps are dry, do the following for each keycap except the spacebar:

1. Take a small amount of air-dry clay and roll it into the keycap legend.
2. Wipe away the excess from the keycap using your finger.

3. Clean up any remainder from the keycap surface with the wet cloth. Be careful not to wipe away much of the clay in the legends.
4. Dry off the top of the keycap by gently blotting with the dry cloth. Be careful not to dislodge the clay in the legends.



A close-up of some keycaps after the air-dry clay has been applied. From left are the Cody key, the Meta key, and the Arrow key.

MAKING THE KEYBOARD CABLE

You'll also need to make an 11-pin cable to connect the keyboard to the Cody Computer's main circuit board. Rather than making a real cable it's a minimal approach using some jumper wires and electrical tape to create a cable by taping the connectors together. One of the actual connectors the cable will connect to

is used as a jig to hold the connectors during the assembly.

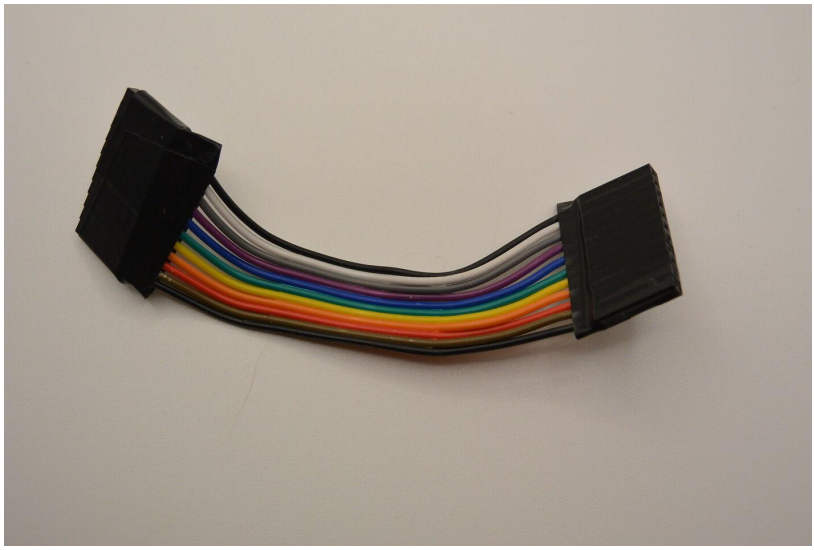
For the jumper wire in this step, use the kind that comes in a strip and can be peeled apart. You're basically trying to make a custom cable on the cheap, so if the wires are connected, you can just tape the connectors together with electrical tape and end up with a reasonable substitute. Jumper wire like this is colloquially referred to as "jumper jerky" and can be found at many retailers.

For this step you'll require only a few parts:

- 1 11-pin male .100" header, right angle
- 11 10cm jumper wire with .100" female connector (from "jumper jerky")
- Electrical tape
- Scissors

Once you've collected the above, proceed with the assembly:

1. Insert one end of the connected jumper wire onto the right-angle header.
2. Wrap electrical tape around the female connectors on that end to secure them together.
3. Remove the connected jumper wire from the right-angle header.
4. Insert the untaped end of the connected jumper wire onto the right-angle header.
5. As before, wrap electrical tape around the female connectors to secure them together.
6. Remove the cable from the connector.



The assembled keyboard cable. Note the electrical tape holding the connectors on each end together.

ASSEMBLING THE KEYBOARD

Once you have the keycaps it's time to build the keyboard. You need to be careful and follow the steps in order. You'll be soldering a connector onto a board that ends up hidden by a keyboard plate. You'll also be inserting switches through a keyboard plate into a printed circuit board and then soldering them. If you do the steps in the wrong order, you'll end up in a situation where further assembly may be mechanically impossible.

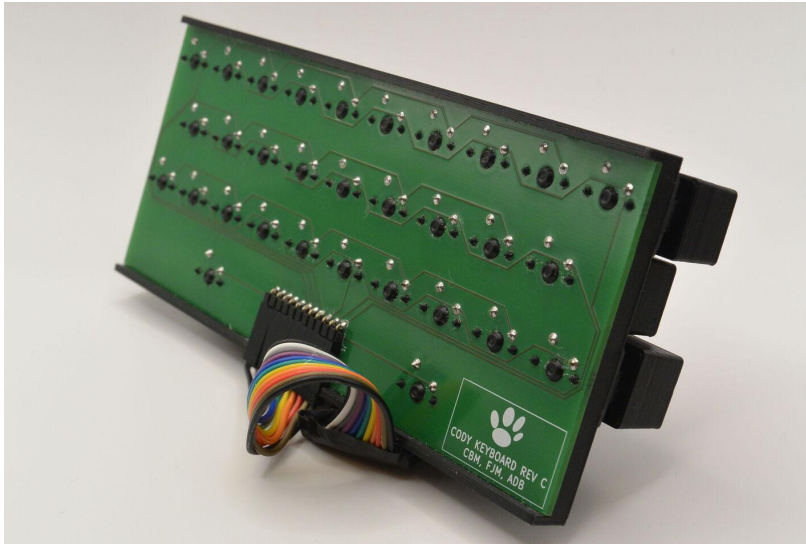
This step requires the following:

- 30 keycaps including spacebar
- 31 keyswitches, 5 pin, PCB mount (Cherry MX or equivalent)
- 1 11-pin male .100" header, right angle

- Keyboard plate (**KeyboardPlate.stl**)
- Keyboard cable
- Solder
- Soldering iron

Refer to the above caution about following the assembly steps. As with anything, it's worth going through the instructions using the parts as a dry run, making sure you understand what you're doing. When adding the spacebar keycap, equal force on both switches is necessary, and you may need to sand the interior of the spacebar to avoid jamming. When you're ready, assemble the keyboard module through the following steps:

1. Solder the 11-pin right angle male connector to J1. Ensure the connector is flat and the solder joints are good.
2. Place the keyboard plate over the keyboard printed circuit board. Ensure the notch in the keyboard plate aligns with the connector.
3. Insert the Cherry MX switches into the circuit board through the keyboard plate. Ensure the keyswitches are fully seated into the circuit board and hold the plate securely.
4. Solder each of the keyswitches to the circuit board.
5. Press each of the keycaps onto the appropriate switch. Use the photo below to determine the location for each key.
6. Connect one end of the keyboard cable to connector J1. The cable should fit through the notch in the keyboard plate.



The back of the assembled keyboard. Note the placement of the printed circuit board inside the keyboard plate with the keyswitches soldered from the bottom. Also note connector J1 soldered from the now-hidden front of the board, now with attached keyboard cable.

The front of the assembled keyboard. Use this photo as a reference when placing the keycaps.

PRINTED CIRCUIT BOARD ASSEMBLY

The next step is to assemble the printed circuit board for the Cody Computer. This board is the motherboard or logic board for the entire computer, containing all the chips and discrete components necessary for the computer to run (with the exception of the keyboard).

It's important to proceed with the assembly methodically and use good soldering technique at each step. Ensure that components are held to the board by a clamp or piece of tape if needed and check for cold solder joints or solder bridges.

INSTALLING INTEGRATED CIRCUIT SOCKETS

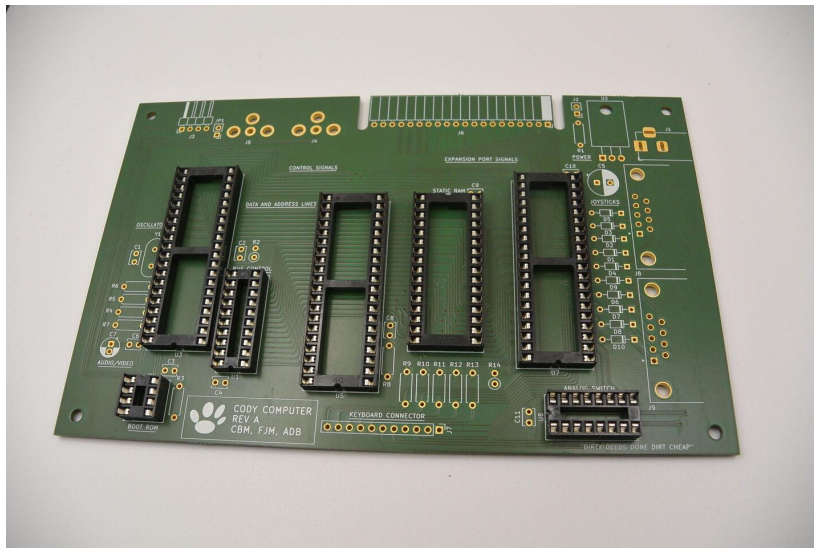
To begin we'll install the sockets for the integrated circuits. Rather than solder the chips directly to the board, we install sockets and add them at a later step. While unlikely to ever happen, this makes it easier to replace one of them if something goes wrong. It also makes it less likely to mess one of them up while soldering, as they're not installed until the end. This step requires:

- 3 40-pin wide DIP sockets
- 1 32-pin wide DIP socket
- 1 20-pin DIP socket
- 1 16-pin DIP socket
- 1 8-pin DIP socket

When installing the sockets, note if your socket contains a notch, dot, half-circle, or other identifier to indicate the top of the IC. If so, ensure they are rotated the same way as the silkscreen on the printed circuit board. Once the sockets have been collected, proceed with the assembly:

1. Solder a 40-pin wide DIP socket into U3 rotated 180 degrees.
2. Solder a 40-pin wide DIP socket into U5 rotated 180 degrees.
3. Solder a 40-pin wide DIP socket into U7 rotated 180 degrees.
4. Solder the 32-pin wide DIP socket into U6.
5. Solder the 20-pin DIP socket into U1 rotated 180 degrees.

6. Solder the 16-pin DIP socket into U8 rotated 90 degrees counterclockwise.
7. Solder the 8-pin DIP socket into U4.



The printed circuit board with the IC sockets soldered in. Note the varying orientations and corresponding notches in the IC sockets.

INSTALLING DIODES

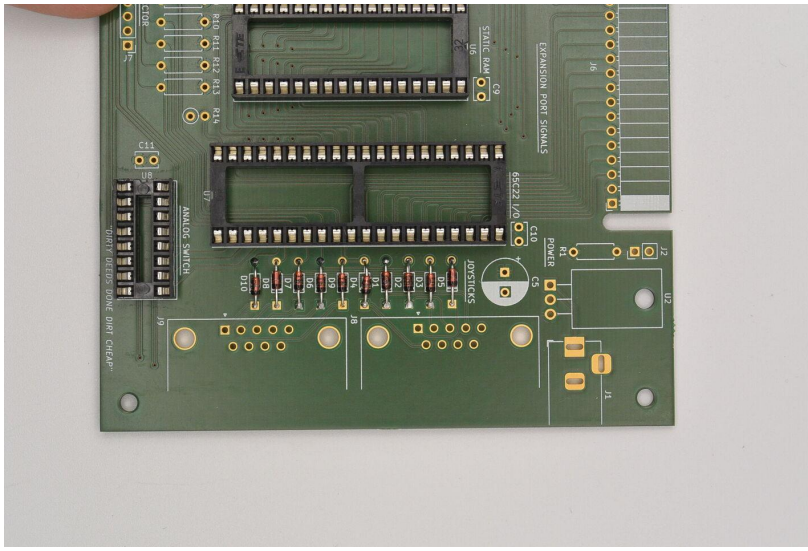
In this step we'll install the diodes for the joystick ports. The Cody Computer uses the same circuit to read the joystick ports as it does to scan the keyboard. Without these diodes, the joystick ports could interfere with each other, causing false reads when both joysticks are in use. You will need:

- 10 1N4148 small-signal diodes

Note that diodes have a polarity. This means that if you solder them in backwards, they won't work as

expected. Each diode has a stripe on it indicating the diode's cathode, and this should be aligned to the corresponding stripe on the silkscreen. Proceed with the assembly starting in order on the PCB:

1. Solder 1N4148 diodes into D5, D3, D2, D1, D4, D9, D6, D7, D8, and D10.



The diodes soldered next to U7 and the future joystick port connectors. Note the stripes and their orientation.

INSTALLING DECOUPLING CAPACITORS

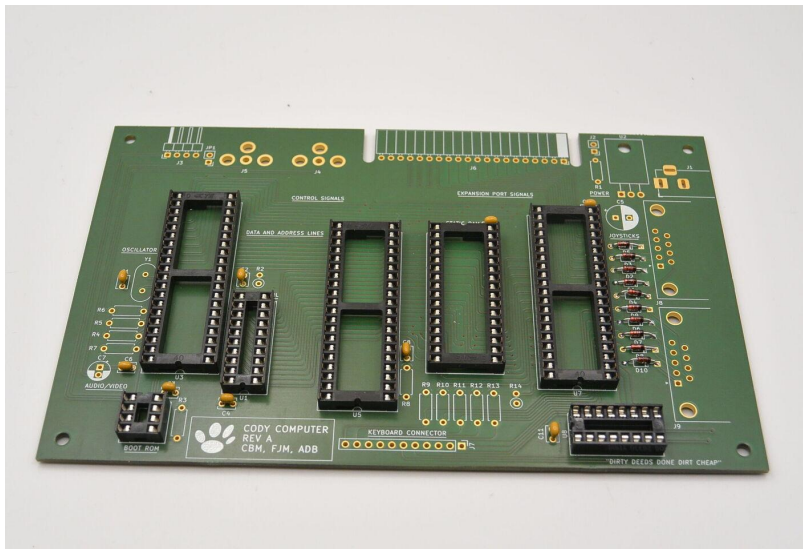
Next we'll install the decoupling capacitors. These are small capacitors that help filter out tiny blips in the Cody Computer's power supply and ensure reliable operation. They're located next to the power supply pins for the integrated circuits. (One of these, C6, is actually part of the audio circuit, but as it has the same

capacitance value, we include it in this step.) You'll need the following:

- 9 0.1µF ceramic capacitors (KEMET C315C104K1R5TA or equivalent)

These are ceramic capacitors and have no polarity, so you don't have to worry about the direction you solder them in (other than, perhaps, for aesthetic purposes). Make sure you solder all of the following:

1. Solder 0.1µF ceramic capacitors into C1, C2, C6, C3, C4, C8, C9, C10, and C11.



The board with decoupling capacitors (plus C6, part of the audio circuit) installed.

INSTALLING THE EXPANSION CONNECTOR

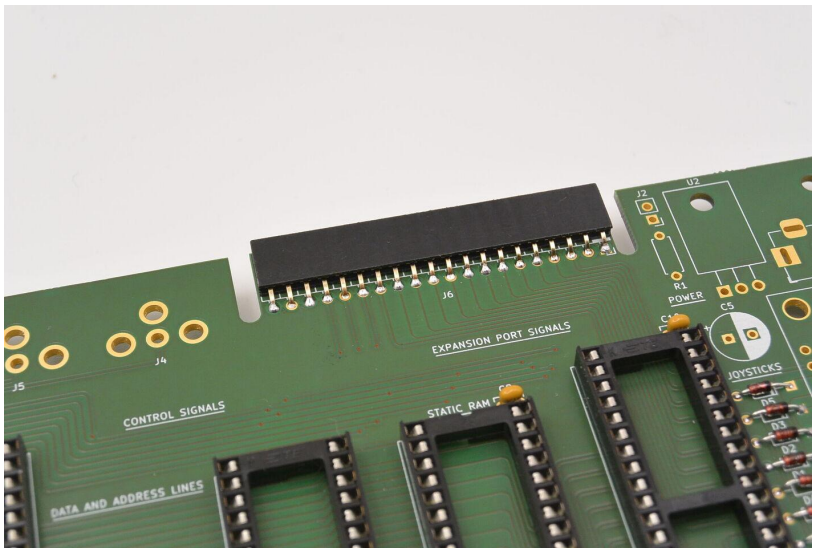
The Cody Computer has an expansion port for DIY experiments, cartridges, or third-party peripherals.

The mechanical connection is a 20-pin right angle . 100" female connector. For this step you'll need the following:

- 1 Raspberry Pi Pico stackable header

Because of their ubiquity, we use one from a set of stackable Raspberry Pi Pico headers (the kind with the long pins) and bend it to fit. Note that the port isn't electrically compatible. We're just using the header, and any standard right-angle female header cut to size would also suffice. For this step do the following:

1. Insert the stackable header into J6 and bend until aligned with the board edge.
2. Solder the stackable header to J6.



The board with the Raspberry Pi Pico stackable header bent into place and soldered.

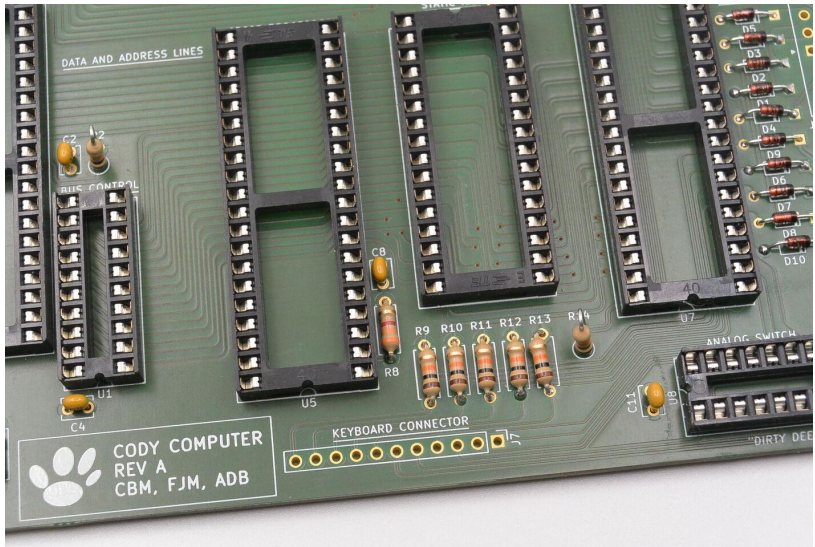
INSTALLING PULL-UP RESISTORS

In this step we'll install several pull-up resistors. Most of these are used by the keyboard matrix, but there are also others. R2 is used to pull up the Propeller's RESET pin, R3 is used as a pull-up for I2C EEPROM communication, and R8 pulls the 65C02's RDY pin high to protect it in the event of a wait-for-interrupt instruction. This step requires the following resistors:

- 8 10kΩ (brown-black-orange) resistors, 1/4 watt, 5% tolerance
- 1 3.3kΩ (orange-orange-red) resistors, 1/4 watt, 5% tolerance

Installation should proceed as follows:

1. Solder 10kΩ resistors to R3, R9, R10, R11, R12, and R13.
2. Solder 10kΩ resistors to R2 and R14 in a vertical orientation (see photo).
3. Solder the 3.3kΩ resistor to R8.



A close-up of some of the resistors after being soldered to the board. Note the vertical orientations of R2 and R14.

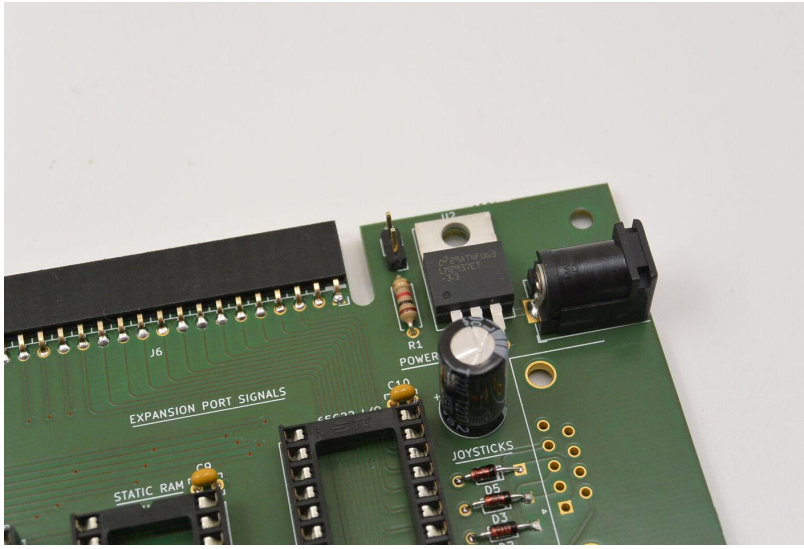
INSTALLING POWER SUPPLY COMPONENTS

The Cody Computer's power supply circuit is located at the top right of the printed circuit board. It consists of a voltage regulator, a large electrolytic capacitor, some connectors, and a resistor. This step requires the following parts:

- 1 LM2937ET-3.3 voltage regulator IC
- 1 1000µF electrolytic capacitor (Rubycon 10ZLH1000MEFC8X16 or equivalent)
- 1 1kΩ (brown-black-red) resistor, 1/4 watt, 5% tolerance
- 1 2.0x6.5mm DC barrel jack (CUI PJ-102A or equivalent)
- 1 2-pin male .100" vertical header

The voltage regulator needs to be bent at a 90-degree angle so that the body and heat sink match the silkscreen on the circuit board. The electrolytic capacitor is polarized and must be installed according to the silkscreen. For this assembly step do the following:

1. Solder the LM2937ET-3.3 to U2. Ensure the IC is placed and bent horizontally as shown in the photo.
2. Solder the 1000µF capacitor to C5. Verify the longer lead is on the positive side and the stripe on the case is on the negative side, following the silkscreen.
3. Solder the 1kΩ resistor to R1.
4. Solder the DC barrel jack to J1.
5. Solder the male header pins to J2.



The power supply circuit including the horizontally-aligned voltage regulator and properly-oriented electrolytic capacitor. Also note the DC barrel jack.

INSTALLING PROPELLER COMPONENTS

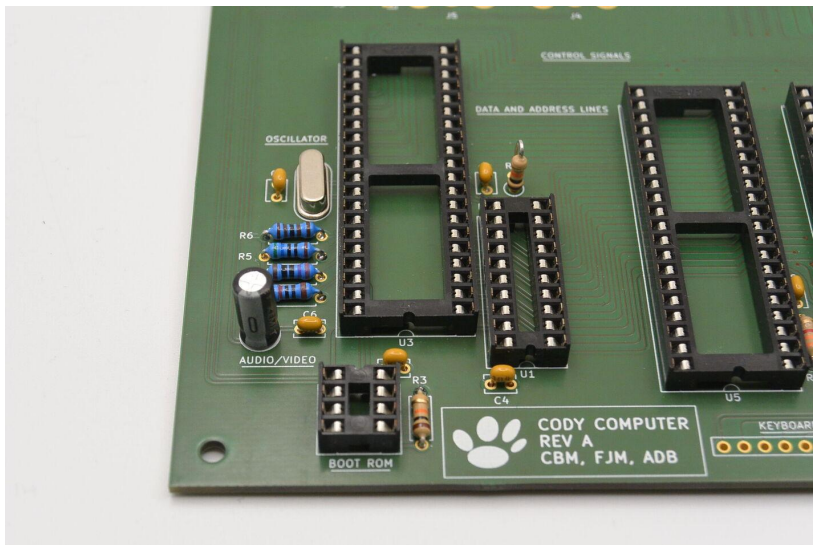
There are still some discrete components to install for the Propeller. These include a 5 MHz crystal that serves as the Propeller's external clock signal as well as some resistors and capacitors used for audio and video output. This step uses the following:

- 1 5Mhz 20pF HC-49/US crystal (ECS ECS-50-20-4X or equivalent)
- 1 10µF electrolytic capacitor (KEMET ESL106M050AC3AA or equivalent)
- 1 1.1kΩ (brown-brown-red) resistor, 1/4 watt, 1% tolerance
- 1 560Ω (green-blue-brown) resistor, 1/4 watt, 1% tolerance

- 1 270Ω (red-violet-brown) resistor, 1/4 watt, 1% tolerance
- 1 220Ω (red-red-brown) resistor, 1/4 watt, 1% tolerance

Once you've found all the components solder the following:

1. Solder the 20pF crystal to Y1.
2. Solder the 1.1kΩ resistor to R6.
3. Solder the 560Ω resistor to R5.
4. Solder the 270Ω resistor to R4.
5. Solder the 220Ω resistor to R7.
6. Solder the 10µF capacitor to C7.



The extra components needed for the Propeller. To the left of the socket, note from the top the crystal oscillator, video DAC resistors, and capacitors and resistor for the audio circuit.

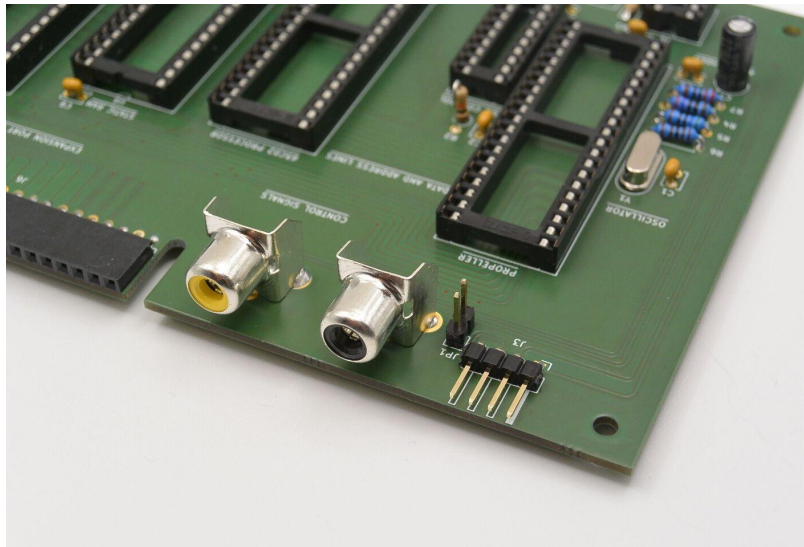
INSTALLING ADDITIONAL REAR CONNECTORS

In this step we'll finish adding the remaining connectors along the back of the Cody Computer. These include the audio and video jacks, a jumper used for firmware programming, and a four-pin connector wired into the Propeller as a serial port. The RCA jack colors are not required but are specified to help tell the video and audio jacks apart once the Cody Computer is assembled. You'll need the following parts for this step:

- 1 RCA jack, black color (CUI RCJ-011 or equivalent)
- 1 RCA jack, yellow color (CUI RCJ-014 or equivalent)
- 1 2-pin male .100" header, vertical
- 1 4-pin male .100" header, right-angle

Add the following connectors:

1. Solder the 4-pin right-angle male header to J3.
2. Solder the 2-pin vertical male header to JP1.
3. Solder the black RCA jack to J5.
4. Solder the yellow RCA jack to J4.



Additional connectors on the back of the Cody Computer. Note from left to right the NTSC video output jack, audio output jack, jumper pins (without jumper attached), and Propeller Plug connector.

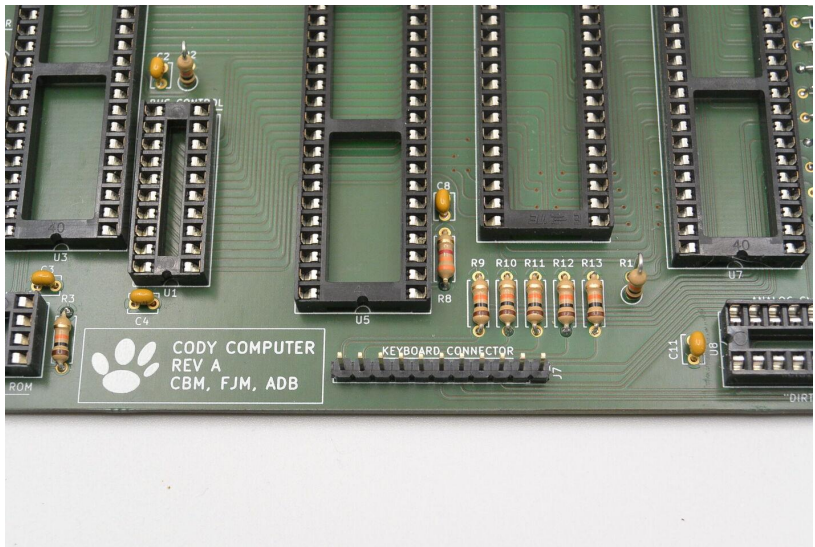
INSTALLING KEYBOARD AND JOYSTICK CONNECTORS

In this step we'll add the connectors for the joystick ports and the keyboard. The DB9 connectors used for the joystick ports as they must have a very specific shape to fit in the alloted space on the board. When ordering you should check the mechanical diagrams to ensure the parts will actually fit. Collect the following:

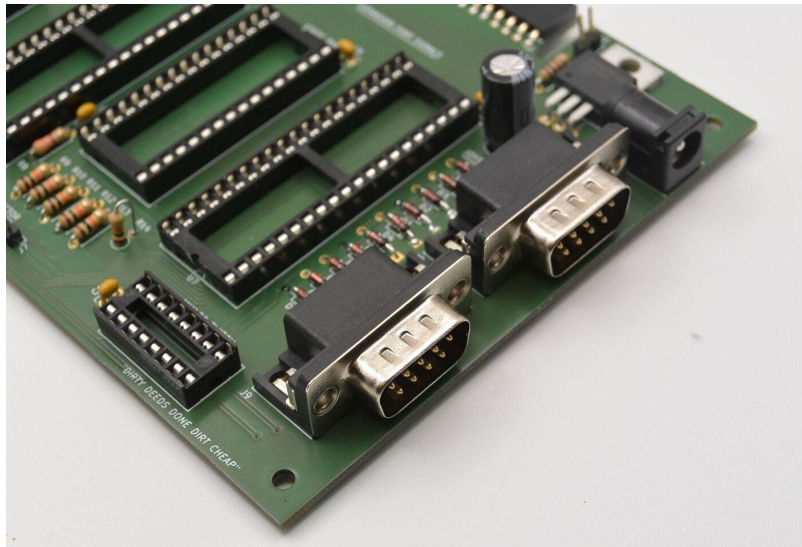
- 2 male DB9 connectors, .318" footprint (NorComp 182-009-113R531 or equivalent)
- 1 11-pin male .100" header, vertical

Solder the remaining components:

1. Solder the 11-pin vertical male header to J7.
2. Solder the two male DB9 connectors to J8 and J9.



The Cody Computer's keyboard connector soldered at the bottom of the board.



The Cody Computer's joystick ports soldered along the right side of the board.

POWER TEST

Now that the printed circuit board has been assembled (except for inserting the ICs), we can begin to test the circuit. We'll start by testing the power supply to ensure we're getting the expected 3.3 volts. If we're not, it's likely a sign of a solder bridge, PCB problem, or an issue with the power supply. It's better to find that out before we insert any chips into their sockets. For this step you will need:

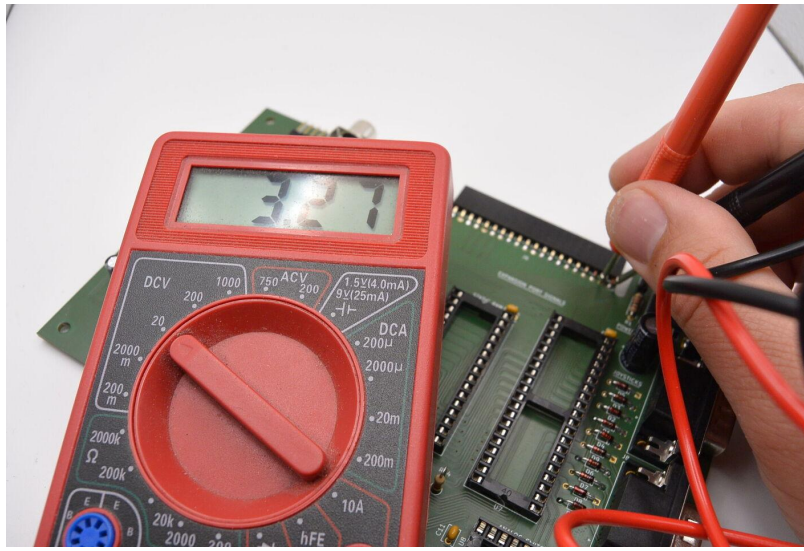
- 5-volt (or similar) DC power supply with 5.5mm x 2.1mm connector
- Voltmeter/multimeter

Any wall-wart transformer or power supply with a suitable plug and an output voltage of 5V (or slightly

above) should work well for this test. To test the circuit do the following:

1. Ensure the printed circuit board is resting on a nonconductive surface.
2. Plug the power supply's barrel plug into the DC power jack on the circuit board.
3. Connect the power supply into a wall outlet.
4. Use your voltmeter to measure the voltage across pins 1 (GND) and 2 (3.3V) on the expansion port.
5. Verify the voltage is 3.3V or very close to it.
6. For advanced builders, find the power supply pins on some of the IC sockets, and test those also.
7. Disconnect the power supply.

If the test fails, check the power supply circuit on the printed circuit board. Also check the voltage from the DC power supply is correct. If none of this yields a result, examine the rest of the printed circuit board for defective traces or solder bridges.



Use a voltmeter to check that the output from the power supply circuit is correct. You should measure a steady voltage around 3.3 volts.

FIRMWARE PROGRAMMING

In this step we'll program the Propeller's firmware. To do so you'll need to insert the first two integrated circuits, the Propeller and its 32-kilobyte EEPROM, into the matching sockets on the board. Once you've done that you'll use Propeller software to write the program into the EEPROM. Before you begin, you'll want to download the software (Propeller IDE or similar) for your computer and familiarize yourself with it.

Also pay attention to the jumper JP1 during assembly. When closed, the Propeller's reset pin connects to the Prop Plug's reset pin, allowing the Prop Plug to reset the Propeller and enter programming mode. When open, the two are disconnected and the

Propeller's reset pin is held high. The latter configuration is the normal mode of operation, but you'll want to remember the jumper exists in case you ever program your own custom firmware.

You will need the following for this step:

- 1 Propeller P8X32A integrated circuit (DIP-40)
- 1 24LC256 32-kilobyte I2C EEPROM or equivalent (DIP-8)
- 1 Prop Plug with USB cable
- 1 2-pin jumper/shunt (Harwin M7583-46 or equivalent)
- Computer running Propeller IDE (or similar programming software)

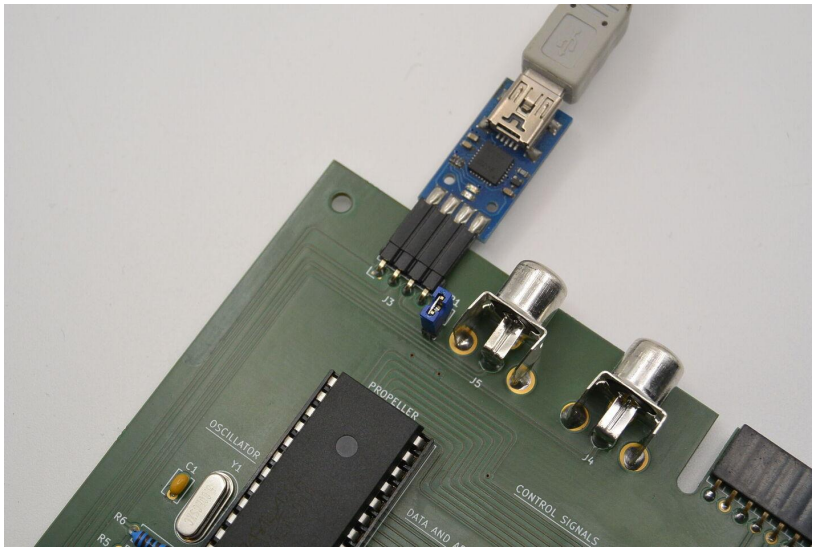
When inserting the integrated circuits, ensure that they're fully seated into their sockets and none of the pins are bent.

The exact steps for programming the firmware will differ depending on the IDE you use, so you will need to refer to the tool's documentation for exact steps. The overall procedure will be the same:

1. Ensure power is turned off to the printed circuit board.
2. Insert the Propeller IC into U3 rotated 180 degrees.
3. Insert the 24LC256 I2C EEPROM into U4.
4. Place the jumper over both pins of JP1.
5. Plug the Prop Plug into J3. Verify the pinout (the rightmost pin next to the jumper, pin 4, is GND).
6. Plug the Prop Plug's USB cable into your computer.
7. Connect power to the printed circuit board.

8. Launch your Propeller software (for example, Propeller IDE).
9. Open the main firmware (**cody_computer.spin**) and write it.
10. Verify that the software states the program was successfully written.
11. Turn off power to the printed circuit board.
12. Unplug the Prop Plug from J3.
13. Remove the jumper. To avoid losing it reattach to only 1 pin on JP1.

If your programming software doesn't recognize the Prop Plug, try disconnecting and reconnecting the cable and/or Prop Plug. If that does not work, ensure that the programming software has permissions to the Prop Plug's USB. If programming the Propeller fails, check the solder connections and ensure the Propeller and its EEPROM are properly seated in their sockets. Also ensure the jumper is correctly attached.



The Prop Plug connected to the serial port on the printed circuit board. Note jumper JP1 in the firmware programming position with both pins covered.

INSTALLING THE INTEGRATED CIRCUITS

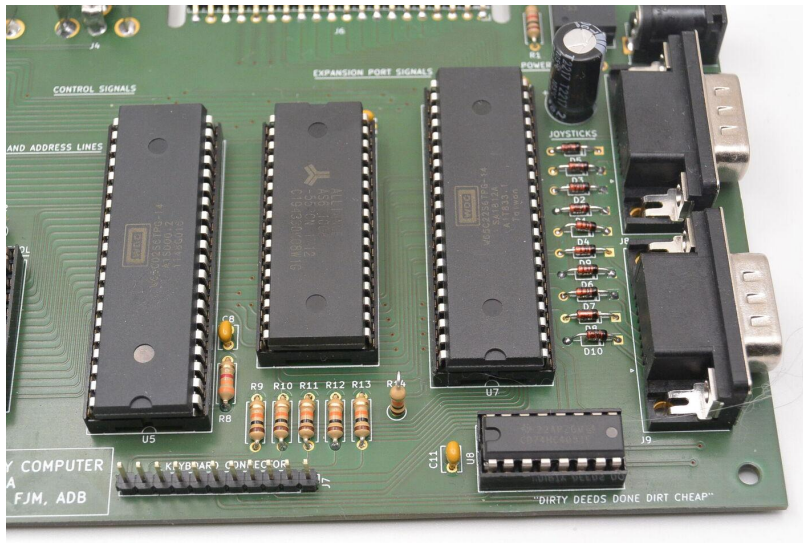
In this step we'll insert the remaining ICs into their sockets. It's very important to make sure that power is disconnected for this step. You will need:

- 1 74HC541 octal line driver (DIP-20)
- 1 W65C02 microprocessor (DIP-40)
- 1 AS6C1008 128-kilobyte static RAM (DIP-32)
- 1 W65C22 Versatile Interface Adapter (DIP-40)
- 1 CD4051 1-of-8 analog multiplexer (DIP-16)

It's also very important to check that the orientation of the integrated circuits matches the silkscreen. Many of the ICs are installed rotated by 90 or 180 degrees.

As before, make sure that each IC goes into the socket fully with no bent pins. Insert the ICs as follows:

1. Insert the 74HC541 into U1. Note U1 is rotated 180 degrees.
2. Insert the W65C02 into U5. Note U5 is rotated 180 degrees
3. Insert the AS6C1008 into U6.
4. Insert the W65C22 into U7. Note U7 is rotated 180 degrees
5. Insert the CD4051 into U8. Note U8 is rotated 90 degrees counterclockwise.



Close up of several integrated circuts securely inserted into their sockets. Note the differing orientations and how the notches on the ICs match with the sockets and silkscreen markings.

CASE ASSEMBLY

Once the printed circuit board and keyboard have been assembled, it's time to begin assembling the Cody Computer's case. We'll start with the top of the case and its components, including the case badge and power LED. From there we'll assemble the rest from the bottom up, installing the printed circuit board and keyboard brackets into the case bottom. Once the bottom portion is finished we'll attach the keyboard to it as well, connecting the keyboard cable to the main printed circuit board. Lastly, we'll affix magnets to hold the case together, connect the power LED, and finish our assembly.

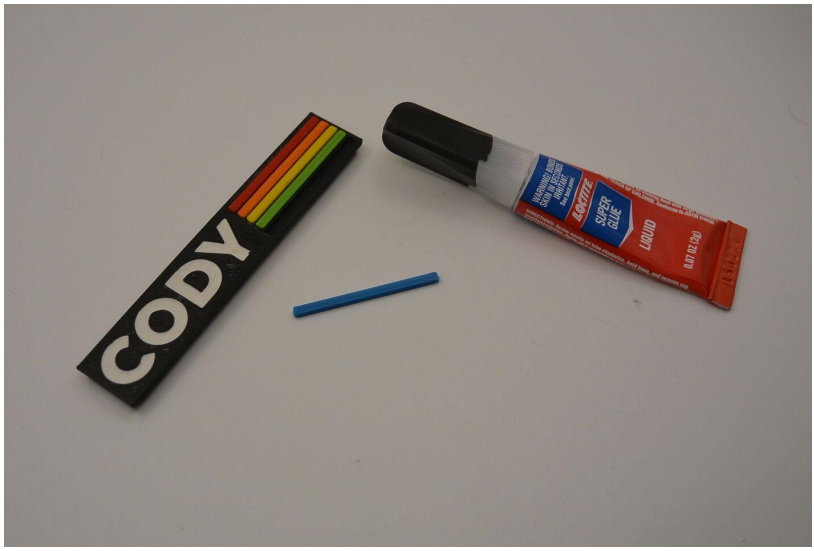
CASE BADGE ASSEMBLY

First we'll assemble the case badge. You should have already printed the case badge and the case badge inlays before beginning this step. Note that if you didn't print the case badge inlays in different colors, you'll have to paint them as part of this assembly step. For this step you'll need:

- 1 case badge (**CaseBadge.stl**)
- 5 case badge inlays (**CaseBadgeInlays.stl**)
- White air-dry clay
- Cyanoacrylate glue
- Optional: Paint (red, orange, yellow, green, and blue) for inlays

Once you're prepared and have collected the parts, proceed with the following:

1. Wash and dry the case badge and case badge inlays. This will help the air-dry clay (and paint if needed) adhere to the plastic.
2. Test-fit the case badge inlays into the slots on the case badge. Sand if necessary.
3. Insert air-dry clay into the "CODY" legend on the case badge. Wipe away excess with a cloth and water.
4. If the inlays were not printed using color filaments, paint the inlays (red, orange, yellow, green, and blue).
5. Allow the air-dry clay to dry completely. If you painted the inlays, allow these to dry then remove any paint from the gluing surfaces.
6. Glue the inlays into the case badge slots (top: red, orange, yellow, green, and blue).



An almost-completed Cody Computer case badge. Air-dry clay was pressed into the legend and all but the blue inlay have been glued into place.

POWER LED ASSEMBLY

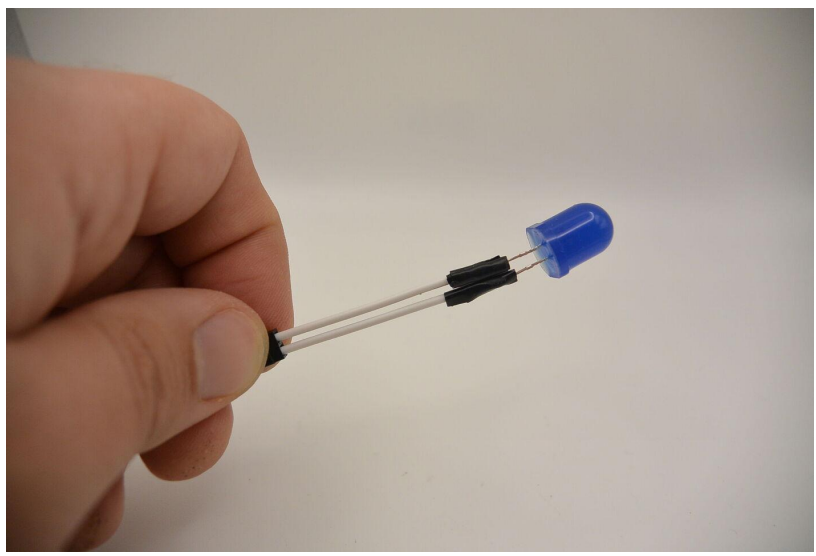
Next we need to assemble the power LED. We're going to solder some leads to the LED and make some other adjustments so that it can be inserted into the Power LED holder. It may be helpful to refer to the attached photo. This step requires the following parts and tools:

- 1 10mm LED (blue)
- 1 10cm jumper wire with .100" female connector
- Electrical tape
- Solder
- Soldering iron
- Scissors
- Wire cutters

- Sharpie (or other marker)

The assembly steps are as follows:

1. Bend the female jumper wire into two equal lengths and secure the connector end with the tape.
2. Cut the jumper wire into two pieces at the bend and strip two or three millimeters from the cut ends.
3. Twist and affix the wire ends onto the LED leads, marking the wire connected to the cathode (longer lead).
4. Solder the wire ends to the LED leads, then trim the excess from the soldered LED leads.
5. Wrap some electrical tape around the soldered portions of the leads to prevent shorts.



The power LED soldered to the jumper wire and female connector.

CASE TOP ASSEMBLY

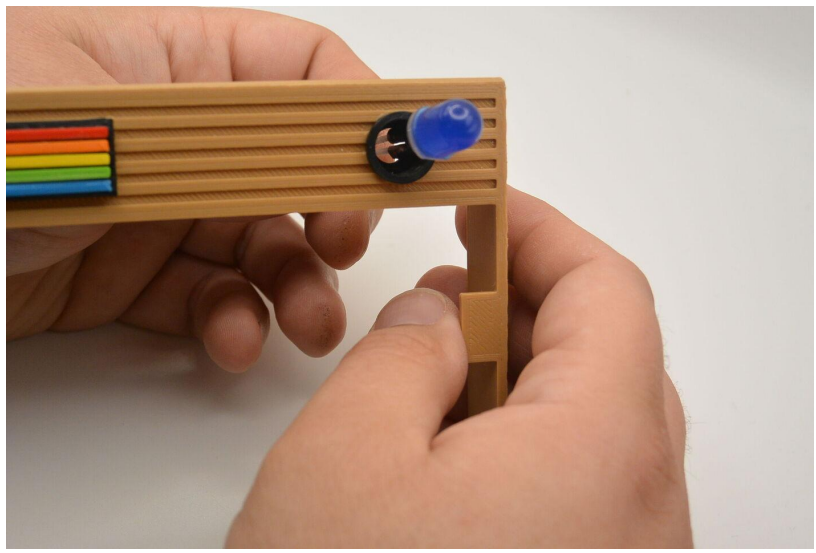
Once the case badge and power LED are ready, we can attach them to the top of the case. In this step we'll glue the case badge and power LED holder to the case, then place the power LED in the holder. You'll need the following:

- 1 case top (**CaseTop.stl**)
- 1 LED holder (**LEDHolder.stl**)
- 1 assembled case badge
- 1 assembled LED with connector
- Cyanoacrylate glue

After collecting the parts proceed with the assembly:

1. Test-fit the power LED in the power LED holder. It should fit without a great deal of force.
2. Glue the case badge into the rectangular slot on the case top.
3. Glue the LED holder (without the LED) into the round slot on the case top.
4. Allow the glue to dry.
5. Place the LED into the LED holder from the front. Don't worry if the LED is too loose as we'll be removing it temporarily in a following assembly step.

The case badge being glued into the case top. The LED holder is visible in the background.



The power LED being inserted into the LED holder from the front.

CASE BOTTOM ASSEMBLY

In this step we assemble the bottom portion of the case including the printed circuit board and keyboard brackets. This step is somewhat trick as it involves lining up the brackets, board, and case bottom in an inverted position, then screwing the case bottom to the brackets. For this portion you will require:

- 1 case bottom (**CaseBottom.stl**)
- 1 left mounting bracket (**KeyboardBracketWithoutHoles.stl**)
- 1 right mounting bracket (**KeyboardBracketWithHoles.stl**)
- 4 M3 x 10mm self-tapping screws, round/pan head (US #4 x 3/8")
- Screwdriver

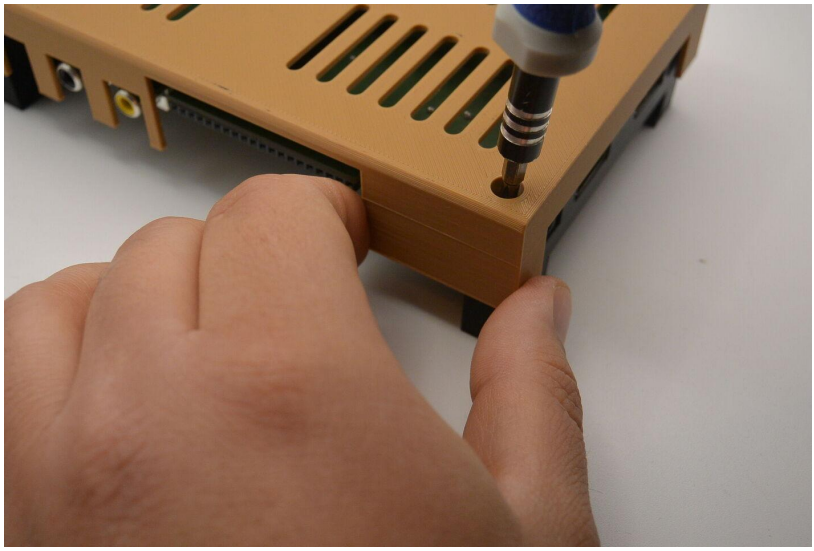
Once you have the parts collected, assemble the bottom of the case:

1. Place the printed circuit board flat on a table (or other surface) with the components facing up.
2. Align the right mounting bracket on to the right side of the printed circuit board. Test the fit for the joystick and power connectors.
3. Align the left mounting bracket on to the left side of the printed circuit board.
4. Flip the entire assembly upside down so that the tops of the brackets are on the table and the bottom of the board is facing up.
5. Align the case bottom (upside down) to the top of the brackets. The rear ports should align with

the slots in the back of the case and the screw holes should align with those in the brackets.

6. Screw the parts together ensuring that the alignment is not disturbed. It may help to screw in from opposite corners to ensure the case and brackets remain aligned.

Testing the keyboard bracket's fit with the joystick and power connectors.



Assembling the case bottom, printed circuit board, and keyboard brackets using screws.

INSTALLING THE KEYBOARD

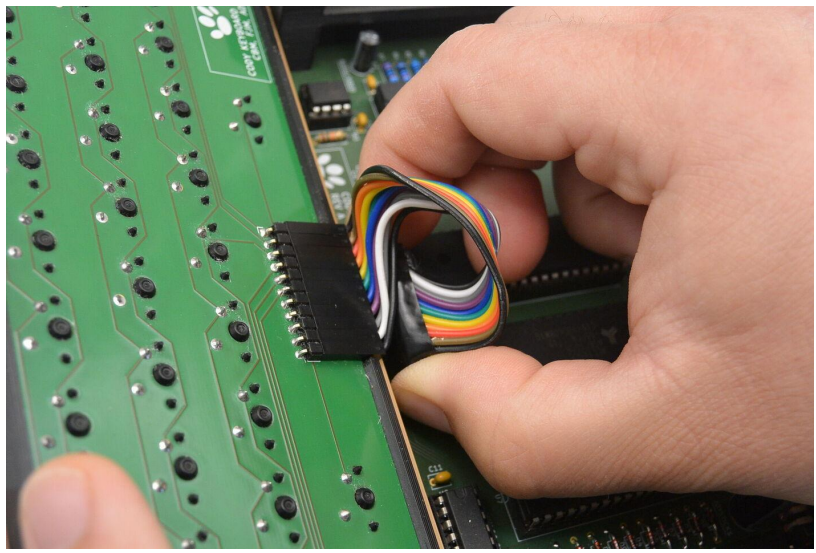
Once the bottom of the Cody Computer is assembled the keyboard module must be attached. The keyboard module's cable must be connected to the keyboard connector on the main printed circuit board. Once the cable is connected the keyboard module must be inserted into place. This step requires:

- 1 assembled case bottom
- 1 assembled keyboard module

Proceed with installing the keyboard as follows:

1. Test-fit the keyboard module ends against the slots in the brackets. This can be done by sliding from the outside of the brackets.

2. Ensure that the keyboard cable is snugly attached to the connector on the keyboard module.
3. Note the wire that corresponds to pin 1 on the keyboard module side of the conector.
4. Identify the matching pin 1 annotation on the main printed circuit board.
5. Attach the keyboard connector to the main printed circuit board. The cable will need to be twisted around to line up.
6. Ensure the keyboard connector is still snugly attached to both connectors.
7. Slide the keyboard into the slots in the brackets from the inside, first one side, then the other.
8. Line up the sides of the keyboard module with the sides of the brackets.



Connecting the keyboard to the main printed circuit board. Note the intentional twist in the cable.

Sliding the keyboard module into the mounting slots on the brackets. Start with one side and then slide in the other.

INSTALLING MAGNETS

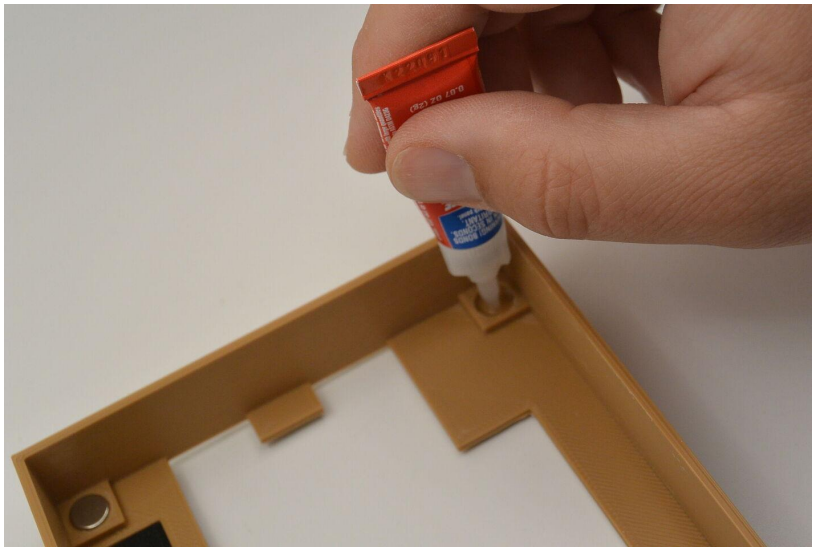
The case is held together with a set of eight rare-earth magnets to permit easy access. As an educational computer, the intention is to make it as open as possible, both metaphorically and literally. With magnets the case can be opened to show off the interior. Be careful that your magnets are glued in with the proper orientation. If you don't the case won't fit together correctly because the magnets will repel instead of attract. You'll need the following:

- 1 assembled case top
- 1 assembled case bottom
- 8 8mm x 2mm rare earth disc magnets (US 5/16" x 5/64")

- Cyanoacrylate glue

Assembly is rather straightforward except for the warning about ensuring the magnets are aligned. One option is to mark each magnet with a Sharpie or other semi-permanent means. Proceed as follows:

1. Temporarily remove the power LED from the case top. Place it in a safe location.
2. Test-fit the magnets into their holes and the assembled case with the magnets in place.
3. Mark one side of each magnet with a marker. Be sure that you are consistent with the side you are marking or the case will not attach correctly.
4. Glue four magnets into the holes in the keyboard slots with the marked side visible, ensuring that the magnets are fully inserted. Be careful not to get glue onto the keyboard by accident.
5. Glue four magnets into the holes in the case top with the marked side not visible. Again, ensure that the magnets are fully inserted.
6. Allow the glue to dry thoroughly.



Installing magnets into the case top. Remember that magnets with opposite orientation need to be installed into the case bottom as well.

Watch out for the magnets as they're not to be swallowed by man or beast. If you have issues with the glue holding them into place, you may want to try a different adhesive. If this happens, consider printing an extra part off for testing purposes.

FINAL ASSEMBLY

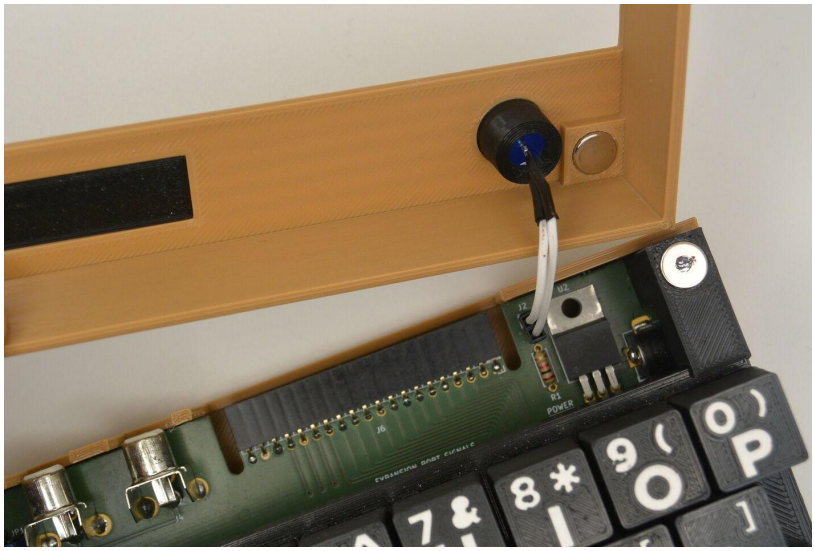
Once the keyboard is connected the only remaining step is to attach the top part of the case to the rest of the Cody Computer. We'll also have to connect the

power LED prior to snapping the case together. You'll need the two parts of the computer:

- 1 assembled case top
- 1 assembled case bottom

The assembly steps are as follows:

1. Reinsert the power LED into the LED holder on the case top. If the LED is too loose, the LED leads can be bent and tape affixed from the bottom to hold it in place.
2. Connect the power LED connector to the printed circuit board. Ensure that the wire you previously marked as the cathode (the long LED lead) is aligned to pin 1 on the LED connector.
3. Align the case top and place it onto the case bottom and brackets, using the magnets to hold the case tight. You may need to push on the LED and/or LED wires to ensure a successful fit without the LED popping out.



Close-up of the connected power LED and magnets. Note the magnets on the brackets have their marked side outward while the magnets on the case have their marked side inward.

The fully-assembled Cody Computer from the front. The case is held together with magnets.

INITIAL SETUP

Now that the Cody Computer is built, it's time to plug it in and test it out. You'll need a few last items that you may have to get from the audiovisual section of your local store:

- RCA video and audio cable (red, white, and yellow plugs)
- RCA audio Y-splitter
- DC power supply (from earlier steps)
- Inline switch for power supply cable (recommended)
- Television with NTSC composite RCA inputs

You're ready to connect the Cody Computer and power it up for the first time:

1. Plug the splitter into the computer's audio port.
2. Plug the red and white audio cables into the splitter.
3. Plug the yellow cable into the computer's video port.
4. Plug the red, white, and yellow cables into the TV.
5. Plug the DC power supply cable into the inline switch.
6. Plug the inline switch into the computer's power jack.
7. Plug the DC power supply into the wall.
8. Turn on the television.
9. Flip the inline switch to turn on the Cody Computer.

The Cody Computer with audio, video, and power connected. Note the inline power switch to the right of the computer.

If all goes well, after a second or two the Cody Computer will boot into Cody BASIC. You'll see a short welcome message, the READY prompt, and a blinking cursor. From here you can learn to program the Cody Computer as well as load and save programs, all of which we'll be covering in the next chapters.

On startup the Cody Computer boots into Cody BASIC.

5

Using Cody BASIC

INTRODUCTION

Now that you have your Cody Computer set up and running, it's time to learn how to use it. In this chapter you'll learn the fundamentals of Cody BASIC, the simple programming language built into the Cody Computer. Cody BASIC is inspired by Tiny BASIC, a 1970s programming language written for resource-constrained hobbyist computers. It also has a lot of influence from Commodore BASIC, a BASIC originally written by Microsoft and modified by Commodore. Cody BASIC is a very simple BASIC but it provides a good starting point for your explorations.

This chapter assumes that you have at least some programming background. If you don't, you can probably still follow along, but it won't be as easy. It doesn't assume any particular familiarity with BASIC dialects of the 8-bit era, which themselves were quite different from any modern BASIC you may have encountered.

USING THE KEYBOARD

You'll be using the keyboard to enter commands in Cody BASIC, so before we begin, we need to cover a little bit about how to use the Cody Computer's keyboard and its special keys. The keyboard is a simplified QWERTY layout with a total of 26 alphabetic characters. Each key contains a letter of the alphabet, and most contain special characters on the top-left and top-right. Pressing the key by itself will

give you the letter, but pressing it with other special keys will give you the special characters instead.

The QWERTY keys as an example of the Cody Computer's keyboard layout. Note the additional characters on the top left and top right.

The Cody Computer's keyboard also contains three additional keys used for special functions: The **Cody** key, the **Meta** key, and the **Arrow** key. These are similar to the modifier keys on more modern computers. On the Cody Computer, they let you type the other special characters just discussed, but they also have some other special functions.

The Cody Computer's special keys. From left, the Cody key (a stylized depiction of Cody's pawprint), the Meta key (depicted as a hollow square), and the Arrow key (containing a left-pointed arrow).

The **Arrow** key is the simplest of the three. When pressed by itself, it acts as a Return key and enters the current line of input. In combination with other keys it can also be used to delete content or break out of running programs.

The **Meta** key is used to make existing keys assume some other function. Pressing it with one of the

alphabetic keys generates the punctuation or math symbol printed on the top right of the key. For example, if you pressed **Meta** followed by **Q**, you would get an exclamation mark. Holding it down when pressing **Arrow** deletes the character previously typed.

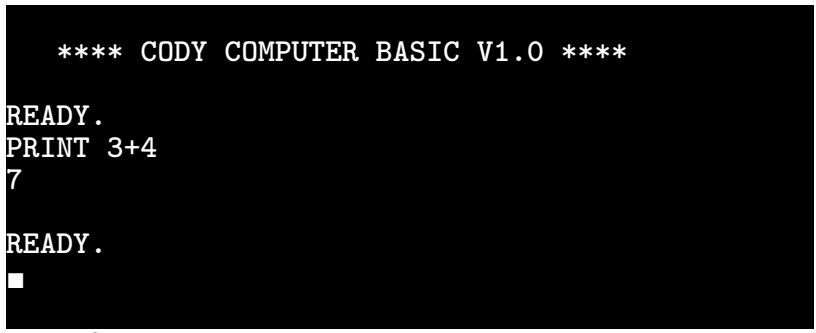
The **Cody** key is another special key. It can be used to obtain extra characters or for system-related functions. When it's pressed with an alphabetic key, it generates the digit printed on the key's top left. If you pressed **Cody** followed by **Q**, you would actually get the number 1. When pressed with **Arrow** it signals Cody BASIC to break out of the current program. When pressed with **Meta**, it toggles the shift mode so that alphabetic keys will be lowercase instead of uppercase (or vice-versa).

THE READ-EVAL-PRINT LOOP

Cody BASIC is an interpreted language as opposed to a compiled one. You can directly interact with Cody BASIC by typing in statements and getting the results back. If you do something that doesn't make sense to it, Cody BASIC will tell you as soon as it finds out about it. You'll interact with the Cody BASIC interpreter in what's called a Read-Eval-Print Loop (REPL), where the Cody Computer reads what you typed, attempts to evaluate it, and prints out a result of what happened if relevant.

To see this in action, start up your Cody Computer. After a moment you should see the welcome message and **READY** prompt at the top of the screen. This indicates the Cody Computer is ready for your

commands. At the blinking cursor, type **PRINT 3 + 4**. Once it's typed in, press **Arrow**. Cody BASIC should print the result, **7**, on the screen, followed by another **READY** prompt.



Your first statement and its output.

If you encountered a syntax error, carefully review what you typed in. Remember that when typing a line, you can use **Meta** + **Arrow** to delete characters. Also remember that you can use the **Cody** and **Meta** keys to enter special characters such as numbers or punctuation. In the above example, to enter **3 + 4**, you would type **Cody** + **E** to get a 3, **Meta** + **F** to get a plus sign, and **Cody** + **R** to get a 4.

TYPING AND EDITING PROGRAMS

When you want to run more than one command at a time, you need to type in a program. Cody BASIC has a built-in way to enter programs using line numbers. First you type in the line number followed by the content for that line, then press **Arrow**. The line is entered into the program. The cursor moves on to the next line.

```
10 PRINT "HELLO"
■
```
Entering a single line into the current program.

To see the current program in memory, you can use the **LIST** command. Entering **LIST** and pressing **Arrow** will show each line in the program.

```
LIST
10 PRINT "HELLO"

READY.
■
```
Listing your simple single-line program.

Because the program is stored in memory, it doesn't run when you type it in. It's waiting for you to tell Cody BASIC to run it, which you can do by entering the **RUN** command.

```
RUN
HELLO

READY.
■
```
Running the single-line example program from above.

If you later want to remove a line, entering the line number by itself (with no spaces) and pressing **Arrow** will delete it.

```
10
LIST

READY.
■
```
Removing line 10 from the program.

If you want to delete the entire program in memory, you can use the **NEW** command instead of turning the Cody Computer off and on. The **NEW** command performs a soft reset of Cody BASIC, clearing out program memory along with associated data and variables.

```
NEW

READY.
■
```
*Using **NEW** before each new program is entered.*

INPUT AND OUTPUT

An important part of writing computer programs is making them interact with the user. In Cody BASIC the **PRINT** and **INPUT** statements handle the most common user interaction. **PRINT** lets you print out information to the user, while **INPUT** lets you get information from the user.

Both statements can use a variety of different types of data, but for now, we'll begin with a simple example

you should type in. Remember to run **NEW** first if you had already typed other programs in.

```
10 PRINT "WHAT IS YOUR NAME";
20 INPUT N$
30 PRINT "HOW OLD ARE YOU";
40 INPUT A
50 PRINT N$," IS ",A," YEARS OLD."
■
```

*A small program demonstrating **PRINT** and **INPUT** statements.*

Line 10 prints out a message asking for the user's name, while line 20 prompts the user and stores the result as text in a variable called **N$**. Line 30 prints out a message asking for the user's age, while line 40 stores the result as a number in a variable called **A**. The last line, line 50, prints out the user's name and age in a message to the user. The semicolons are a special hint to the **PRINT** statement to avoid advancing to another line on the screen, while the commas split up the arguments to the **PRINT** statement.

If you run the program you'll get something like the following:

```
RUN
WHAT IS YOUR NAME? CODY
HOW OLD ARE YOU? 14
CODY IS 14 YEARS OLD.

READY.
■
```

An example run of the above program.

If you encounter any errors, remember that you can **LIST** your program and check the offending line for any typos. If you find any, retype the line correctly and re-run the program. A more detailed discussion of error messages is found later in the chapter if you get stuck, but for this program, you probably won't need it. Just make sure what you typed in matches the program, and refer to the earlier section on typing in programs whenever you need to.

VARIABLES, NUMBERS, AND STRINGS

Variables are used to store data in your programs. In the previous input-output example, variables held the name (in variable **N$**) and age (in variable **A**) of the user. Most programs will use variables for a variety of purposes, so it's important to understand them and what they can hold.

Variables can be one of two types, corresponding to the two data types supported by Cody BASIC. Number variables contain numbers, while string variables contain text. The two cannot be directly substituted for one another in a program, but functions exist to convert between the two types. Other functions also exist for special operations that pertain to each type, such as square roots for numbers or extracting substrings for strings.

NUMBERS AND NUMBER VARIABLES

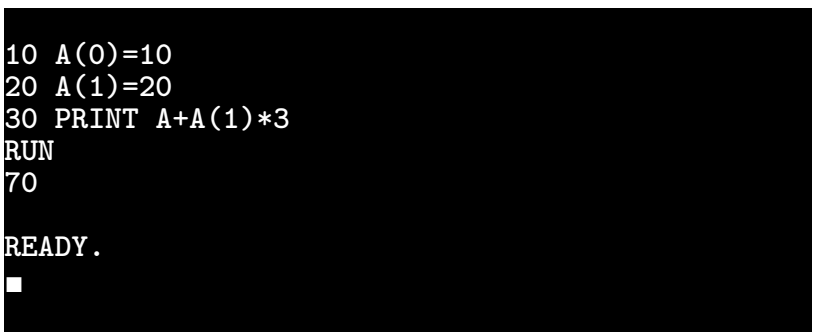
Numbers in Cody BASIC are 16 bits and represent integers between -32768 and 32767, inclusive.

Numbers can be used in mathematical expressions, such as addition, subtraction, multiplication, and division, as well as in various mathematical functions. They are also the return type of most Cody BASIC functions. Most data in a Cody BASIC program is likely to be numeric in nature.

Number literals are just the number typed in, for example **10** or **1234**. These values can be used just about anywhere that a number is required.

Number variables are represented by a letter between **A** and **Z**. Number variables are temporary storage for numeric data in a program, and each can hold one number in its assigned memory.

Number variables in Cody BASIC are somewhat unique in that they also act as arrays. There are a total of 128 indexes into a number array, with each index itself a number between 0 and 127. The use of a number variable without an index is actually just a shorthand for the first element in the array, meaning that **A(0)** and **A** are actually the same variable.



*An example type-in program demonstrating numbers, number variables, and arrays. Note how **A** is used as an alias for **A(0)**.*

STRINGS AND STRING VARIABLES

Strings in Cody BASIC are text information. Each string can consist of up to 255 characters plus a terminating NULL character, and internally strings are represented as C-style byte arrays. Cody BASIC has somewhat limited support for strings and string handling, but it does support a minimum set of string functions suitable for most beginner-to-intermediate programs. These functions include limited string concatenation and substring extraction.

String literals consist of characters contained in double quotes. For example, **"HELLO"** and **"1234"** are both string literals, even though the latter is a string containing numbers.

Cody BASIC also has 26 string variables **A$** through **Z$**, each of which contains a single string. Each variable has its own assigned memory and there is no overlap with the number variables **A** through **Z**. String arrays are not supported.

```
10 M$ = "HELLO "
20 N$ = "WORLD!"
30 PRINT M$,N$
RUN
HELLO WORLD!

READY.
■
```

An example type-in program demonstrating strings and string variables.

CONTROL STATEMENTS

Cody BASIC has several statements that allow you to change the course of a running program. Most programs need to be able to do this to respond to internal or external situations as well as to perform processing within a running program. The **IF** statement allows the program to take different branches based on conditional expressions. The **GOTO** statement allows the program to jump to a different line in a program. **GOSUB** and **RETURN** allow programs to call subroutines on other lines and return back to the calling location. **FOR** and **NEXT** allow a program to loop for a defined number of iterations, incrementing a variable as a side effect.

IF STATEMENTS

The **IF** statement makes a decision based on the result of an expression. These statements are the primary way of controlling the behavior of a program based on data or user input. When the expression is true, the portion of the statement after **THEN** is evaluated. If not, then the remainder of the statement is skipped entirely. **IF** statements are often combined with **GOTO** or **GOSUB** to pass control to other parts of the program based on the results of decision criteria.

For numeric data, the expression consists of numeric expressions on the left hand and right hand sides. The expression also contains a relational operator that acts as the decision-maker, with the less-than (**<**), greater-than (**>**), less-than-or-equal (**<=**), greater-than-or-

equal (**>=**), equal-to (**=**), and not-equal (**<>**) relations supported.

```
10 INPUT N
20 IF N<0 THEN PRINT "NEGATIVE"
30 IF N=0 THEN PRINT "ZERO"
40 IF N>0 THEN PRINT "POSITIVE"
RUN
? 3
POSITIVE

READY.
■
```

Example program using if-statements and relational operators for numbers.

IF statements can also use strings in their expressions. The same relational operators are used and comparisons are performed lexicographically using the CODSCII value for each character.

```
10 INPUT S$
20 IF S$<"B" THEN PRINT "LESS"
30 IF S$="B" THEN PRINT "EQUAL"
40 IF S$>"B" THEN PRINT "GREATER"
RUN
? BA
GREATER

READY.
■
```

Example program using if-statements with strings.

GOTO STATEMENTS

The **GOTO** statement behaves like a high-level version of a jump instruction, moving control to another line in the program without any direct possibility of returning. **GOTO** statements are often frowned upon in modern programming, but they were a common technique in the early days of BASIC programming.

```
10 PRINT "A"
20 GOTO 40
30 PRINT "B"
40 PRINT "Z"
RUN
A
Z

READY.
■
```

*A program using **GOTO** to skip to another line.*

GOSUB AND RETURN STATEMENTS

The **GOSUB** and **RETURN** statements implement subroutine calls in Cody BASIC. The **GOSUB** statement tells the program to call a subroutine starting at a specific line number. The **RETURN** statement tells the program to go back to the line after the most recent **GOSUB**.

Using these together allows Cody BASIC programs to have a simple form of subroutines similar to those in early BASIC interpreters. The statements don't

support additional features of more modern languages, such as parameter passing or return values. Such features need to be explicitly handled by passing data in variables.

```
10 PRINT "A"
20 GOSUB 50
30 PRINT "C"
40 END
50 PRINT "B"
60 RETURN
RUN
A
B
C

READY.
■
```

*An example of a subroutine using **GOSUB** and **RETURN**.*

FOR AND NEXT STATEMENTS

The **FOR** and **NEXT** statements implement a counting loop in Cody BASIC. Each **FOR** statement takes a number variable (which can include an array index), a starting number or expression, and an ending number or expression.

The following **NEXT** statement repeats the body of the **FOR** loop until the variable equals the ending number from the **FOR** statement. On each loop, the value of the variable is incremented by one.

```
10 FOR I=1 TO 5
20 PRINT I
30 NEXT
RUN
1
2
3
4
5

READY.
■
```

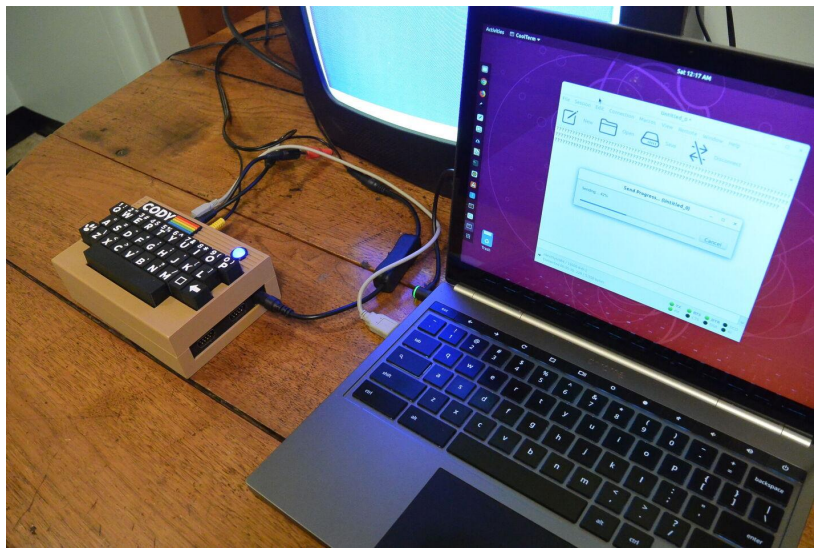
A simple for-loop that prints out the loop variable's value.

LOADING AND SAVING PROGRAMS

You don't always have to type in programs to load them. Cody BASIC supports **LOAD** and **SAVE** statements for loading existing programs and saving the current program. These commands rely on the existence of another device connected to the Cody Computer via the Prop Plug, typically a computer or mobile device running some type of terminal program. BASIC programs are stored as plain text files that can be transmitted and received by any terminal software that has the appropriate features.

To load and save BASIC programs the terminal software you use will need to support regular serial communications at 19200 baud, 8-N-1 (eight data bits, no parity bit, and 1 stop bit), and ASCII linefeeds for the end-of-line character. When transmitting files, it

should allow for a configurable per-line delay of up to 40 or 50 milliseconds. This final requirement is necessary so that Cody BASIC can tokenize an incoming program.



Loading a Cody BASIC program from a Chromebook Pixel running Ubuntu. The Linux version of CoolTerm is used as the terminal program.

You should be able to use any terminal program that meets the above requirements. I used Roger Meier's cross-platform *CoolTerm* during development because it supports all the necessary features to transmit and receive files with Cody BASIC. For Android devices, Kai Morich's *Serial USB Terminal* is a good choice once you have the configuration sorted out.

SAVING A PROGRAM

To save a program we'll need a program to save in the first place. Type in the following and verify the program contents using the **LIST** command.

```
10 PRINT "SAVED PROGRAM"
■
```

A boilerplate program to use for our saving and loading example.

Once you have the program entered in, go to your terminal program on the other computer. Using the software, save a text file from the Prop Plug at 19200 baud, serial setting 8-N-1, and line feeds for the end of line. The software should be waiting for you to save the program.

At this time, run the SAVE command on I/O port 1, the Prop Plug:

```
SAVE 1

READY.
■
```

Saving the sample program.

Once you see the **READY** prompt, the program has been sent. In your terminal software, stop receiving, then verify the contents of the received file. You should see a two-line text file, one containing the print statement, and another completely blank line indicating the end of the BASIC program. (If you

encounter problems during this step or the next, you may want to examine the file in more detail using a hex editor.)

```
10 PRINT "SAVED PROGRAM"
```

Saved program from the terminal program. Note the required blank line marking the end of the program in the saved file.

LOADING A PROGRAM

Now that you've saved a program, it's time to load it and verify that all is in working order. To begin, clear out program memory using the **NEW** command, then **LIST** the current program to verify nothing is there. The **LOAD** command replaces the current program, but for testing purposes, we want to be sure before we proceed.

Once you're sure there's no program in memory, run the **LOAD** command, We're loading from I/O port 1, the Prop Plug, in mode 0. Mode 0 indicates we're loading a Cody BASIC program, while mode 1 indicates that we're loading a binary program, something we'll cover later.

```
LOAD 1,0
```

Loading the previously-saved program.

Now that the Cody Computer is waiting for the program, go back to your terminal and send the program. You'll want to send it as a text file, again at 19200 baud and 8-N-1 with ASCII linefeeds as the

end-of-line character. Also remember to insert a per-line delay, perhaps starting around 40 or 50 milliseconds to be conservative.

Once the program has been received, the **LOAD** command will stop with a **READY** prompt. List the program to verify its contents, then run it.

```
READY.
LIST
10 PRINT "SAVED PROGRAM"

READY.
RUN
SAVED PROGRAM

READY.
■
```

Transcript of loading and verifying the sample program.

If you encounter any problems, verify the serial connection and serial software is working correctly. Also note that the per-line delay can be raised or lowered on a per-program basis, as the time required to parse the longest line in the program depends on the line's complexity.

Cody BASIC actually sends an ASCII question mark before waiting for the next line of the incoming program. A dedicated program or peripheral could also check for this as an optimization along with the normal line delay. This would speed up the loading of Cody BASIC programs without having an effect on anything else.

UNDERSTANDING ERROR MESSAGES

Sometimes when entering or running a program, things can go wrong. Cody BASIC has a small set of error messages to try and help you diagnose the underlying problem. Cody BASIC is patterned after Tiny BASIC and has only three error types, but given Cody BASIC's relative simplicity, these are sufficient. The error messages are inspired by the later Commodore BASIC, and while they may not tell you everything, they should tell you enough to investigate what happened.

The three error types represent syntax errors (when Cody BASIC couldn't parse what you typed in), logic errors (when your program tried to do something that made no sense), and system errors (something about the current computer's state made it impossible to do what was asked).

Errors can occur when entering lines into the REPL or when a program is run. If an error occurs while a program is running the line number in the program

will be included in the error message. If the error occurs in REPL mode, there isn't any associated line number, and none will be shown.

```
RUN

LOGIC ERROR IN 10


READY.
■
```

An example error message that includes a line number.

SYNTAX ERRORS

Syntax errors occur when something you've typed in doesn't fit with Cody BASIC's grammar. Cody BASIC, like any programming language, is defined by a strict grammar specifying what statements and expressions are valid. If you type in something that's invalid, Cody BASIC can't understand what you mean and prints out a syntax error.

```
PRINT !!!

SYNTAX ERROR


READY.
■
```

A syntax error in REPL mode resulting from invalid characters in a PRINT statement.

LOGIC ERRORS

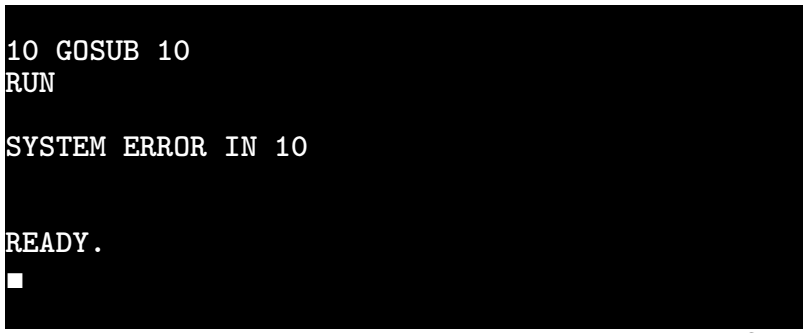
Logic errors result when Cody BASIC is asked to do something nonsensical. This can be something obvious, such as attempting to divide by zero or specifying an invalid value for a character or constant. It can also be something less obvious, such as attempting to read data that doesn't exist or trying to change the current position in the program in a way that doesn't make sense.

```
PRINT 1/0

LOGIC ERROR


READY.
■
```

A logic error in REPL mode resulting from a division by zero.

SYSTEM ERRORS

System errors happen when Cody BASIC isn't able to perform a requested operation that's otherwise valid. This can occur if some of Cody BASIC's internal data areas overflow, making it impossible to run some of its control structures or evaluate complex expressions. It can also happen during I/O operations if errors are detected or if invalid data is passed to certain functions.



A system error in a program caused by infinite recursion in a GOSUB.

6

Advanced Cody BASIC

INTRODUCTION

Now that you're familiar with some of the basics of Cody BASIC, it's time to learn about its more advanced features. While "advanced" is relative and Cody BASIC is intentionally simplified, it has a set of features consistent with many 8-bit BASIC dialects. It has support for minimal mathematics and string operations, literal data, text file input and output, reading and writing memory, and even the ability to call into machine code from BASIC programs.

WORKING WITH NUMBERS

Cody BASIC supports many of the more common mathematical operations, although with some limitations. Numbers in Cody BASIC are integers ranging from -32768 to 32767, so many mathematical operations are limited by necessity. A handful of math functions are also implemented. More complicated functions must be implemented by the user either in BASIC or using machine language and calling it from your program.

ARITHMETIC OPERATIONS

For arithmetic operations, the standard addition, subtraction, multiplication, and division are supported. Cody BASIC obeys the normal order of operations, with multiplication and division performed first, followed by addition and subtraction. Expressions that

are very complex may cause Cody BASIC's expression stack to overflow and produce a system error.

```
PRINT 4+5*6-10
24

READY.
■
```

Cody BASIC follows the order of operations.

Because all numbers in Cody BASIC are integers, the result of division will sometimes be different than you would expect. The result of a division is the integer portion without any remainder because fractional or decimal values aren't supported.

```
PRINT 16/5
3

READY.
■
```

Numbers in Cody BASIC are integers, so integer division is used.

Parentheses are used to group subexpressions. Expressions in parentheses are evaluated first, starting with the most nested set of parentheses and working outward. As with expressions, deeply nested parentheses can cause problems with the interpreter, so it's best to keep expressions simple.

```
PRINT 3*((8+2)/2)
15

READY.
■
```

Using nested expressions in Cody BASIC.

Negative numbers are supported by adding a leading minus sign (known as a unary minus). The leading minus works like it does in normal arithmetic, so it can be used in front of variables and expressions as well as in front of numbers.

```
PRINT -(1+2)
-3

READY.
■
```

An example of a leading minus sign in front of an expression.

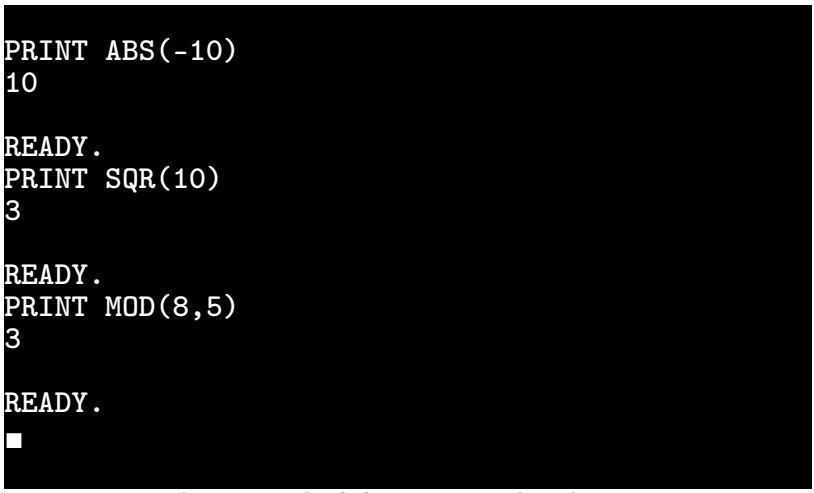
In fact, number variables can be used just about anywhere that a number would be used in Cody BASIC. Unlike many BASIC dialects, both numbers and numeric expressions can be used as the destination for **GOTO** and **GOSUB** statements.

```
10 A=20
20 B=2
30 PRINT -A*B
RUN
-40

READY.
■
```

A program showing the use of variables in an expression.

MATHEMATICAL FUNCTIONS

Cody BASIC has a limited set of mathematical functions. The **ABS()** function returns the absolute value of a number. Another function, **SQR()**, returns the square root of a number with the limitation that only the integer part is represented. **MOD()** returns the modulus (remainder left over after a division) of two numbers.



*Examples of the **ABS**, **SQR**, and **MOD** functions.*

The **RND()** function exists to generate random numbers between 0 and 255. The function has two forms, one that accepts a number as the random seed value, and a no-argument form that returns the next random number in the sequence. For a given seed value the resulting sequence will always be the same. A seed value of zero is invalid and will be replaced with the system's default seed value.

```
PRINT RND(10)
0

READY.
PRINT RND()
186

READY.
PRINT RND()
57

READY.
■
```

*Using the **RND** function to generate pseudorandom numbers.*

A common trick is to use the **TI** time variable to seed a random number sequence at the start of a program, discarding the initial result. The **TI** variable is discussed later in the section on timekeeping.

```
PRINT RND(TI)
52

READY.
PRINT RND()
81

READY.
■
```

*Seeding the **RND** function with the current timekeeping value.*

BITWISE FUNCTIONS

Cody BASIC also has bitwise functions that perform binary operations on numbers. These work on the raw bits in each number, which means it's important to consider how the numbers themselves are stored as zeroes and ones. **NOT()** returns the negation of the bits in the number, **AND()** returns the bitwise and, **OR()** returns the bitwise or, and **XOR()** returns the bitwise exclusive-or.

```
10 INPUT A
20 INPUT B
30 PRINT "NOT ",NOT(A)
40 PRINT "AND ",AND(A,B)
50 PRINT "OR ",OR(A,B)
60 PRINT "XOR ",XOR(A,B)
RUN
? 1
? 0
-2
0
1
1

READY.
■
```

A program that lets you experiment with the output of bitwise functions.

TEXT MANIPULATION AND STRINGS

Cody BASIC supports rudimentary string manipulation. Each of the 26 string variables is a

separate buffer that can store up to 255 characters plus a terminating null character (similar to a string in the C programming language). A separate buffer allows string concatenation in string expressions, and a handful of functions exist to work with string data.

STRING CONCATENATION

Strings can be concatenated together in string expressions. Unlike mathematical expressions, string expressions are very simple and can contain only strings, string variables, and string functions, and the only supported operator is the addition sign (representing string concatenation in this case).

Because Cody BASIC has minimal string support, string expressions can appear in a limited number of places. The most common case is in assignment to string variables where the right hand side of the assignment is a string expression. String expressions can also appear as arguments in **PRINT** statements, where string functions are often used to print out only portions of a string.

```
10 A$="HELLO"
20 B$="WORLD"
30 C$=A$+", "+B$+"!"
40 PRINT C$
RUN
HELLO, WORLD!

READY.
■
```

An example of a string expression in an assignment.

STRING COMPARISONS

As mentioned in the previous chapter, **IF** statements in Cody BASIC have a special case that supports string comparisons. This form is more limited and requires a string variable as the left hand side of the comparison and a string expression as the right hand side of the comparison. Usually the right hand side is just a string or another string variable, but the right hand side may be a full string expression if needed.

```
10 INPUT A$
20 INPUT B$
30 IF B$=A$+"!" THEN PRINT "MATCH"
RUN
? HELLO
? HELLO!
MATCH

READY.
■
```

*A contrived example of using string concatenation in an **IF** statement.*

FUNCTIONS IN STRING EXPRESSIONS

Cody BASIC has three string functions which may appear in a string expression. The **SUB$()** function returns a substring from a string variable. The **CHR$()** function, on the other hand, lets you build a string from one or more numbers representing CODSCII characters. The last function, **STR$()**, returns a string representation of a number. Functions that return

strings are marked by a dollar-sign (**$**) as their last character, similar to Commodore BASIC.

The **SUB$()** function takes three parameters, a string variable, a starting position within the string, and the number of characters to extract. The first argument must always be a string variable because of Cody BASIC's internal implementation. String literals are not supported, and string expressions cannot be nested like mathematical expressions.

```
10 A$="POMERANIAN"
20 PRINT SUB$(A$,0,3)
RUN
POM

READY.
■
```

*Printing out a substring using the **STR$** function.*

To generate a string from a series of character values, you use the **CHR$()** function. Much like a secret code, strings in Cody BASIC are made up of CODSCII characters between 0 and 255. (CODSCII is just an extended ASCII with the Commodore graphical characters moved into the extended ASCII range.) You simply pass one or more numbers (or numeric expressions) to the function and it will return a string with the equivalent characters. This is typically used for printing control codes or graphical characters, but can be used with any valid character code.

```
PRINT CHR$(67,111,100,121)
Cody
READY.

■
```

*Converting numbers to characters using the **CHR$** function.*

The last string function, **STR$()**, converts a number to its string equivalent. For example, the number **10** would be converted to a string equivalent to the literal **"10"**. Many of these conversions happen automatically in **PRINT** statements, but using the **STR$()** function directly lets you use the result in string expressions and assignments.

```
10 INPUT N
20 S$=STR$(N)
30 PRINT S$
RUN
? 123
123

READY.
■
```

A silly example of converting a number to a string for later use.

ADDITIONAL STRING FUNCTIONS

Cody BASIC also has some functions that work with strings but return numbers. To parse a string variable containing a number, the **VAL()** function can be used.

For finding the length of a string, the **LEN()** function is available. And for returning the CODSCII value of a character in a string, the **ASC()** function exists.

The **VAL()** function is relatively simple to use. It takes a string variable and returns the number it was able to parse from the beginning of the string. Leading minus signs are supported. In situations where there were no valid digits to parse, the function returns zero. In many respects this function can be considered the inverse of the **STR$()** function.

```
10 INPUT S$
20 N=VAL(S$)
30 PRINT N*2
RUN
? 10
20

READY.
■
```

Converting a string containing a number into an actual number.

The **LEN()** function returns the length of a string variable, not including the terminating null character. If a stored string is somehow corrupted or poorly-formed, **LEN()** raises a system error when the terminating null is not found.

```
10 INPUT S$
20 PRINT LEN(S$)
RUN
? KODACHROME
10

READY.
■
```

Finding the length of a string.

The **ASC()** function returns the character code for the first character in a string variable. If the string is empty, the null character is returned instead. In many respects this is the inverse operation of the **CHR$()** function, except that the **ASC()** function only works on the first character of the string.

```
10 INPUT S$
20 PRINT ASC(S$)
RUN
? CARRABELLE
67

READY.
■
```

Obtaining the character code for the first character in a string.

To find character codes for other than the first character, you need to use the **STR$()** function to extract a substring into a temporary variable. The temporary variable can then be used as the input for **ASC()**. This has significantly more overhead because

of the temporary string, but in situations where it is needed, this is the typical solution.

```
10 INPUT S$
20 INPUT N
30 T$=SUB$(S$,N,1)
40 PRINT ASC(T$)
RUN
? FOLKSTON
? 2
76

READY.
■
```

Obtaining a different character code using a temporary string.

PRINT FORMATTING

Cody BASIC's **PRINT** statement provides ways of formatting your output. The formatting can be very simple, such as moving the cursor on the screen or aligning data in columns. More complicated formatting can include clearing the screen, changing the foreground and background colors on a per-character basis, or using graphical characters alongside the typical letters, digits, and punctuation marks.

PRINT statements support output formatting in two ways. One is using the special formatting functions **AT()** and **TAB()**. The other is to print special control character codes using the **CHR$()** function which are later handled by the Cody BASIC interpreter.

POSITIONING THE CURSOR

The current cursor position can be updated within **PRINT** statements using the **AT()** function. The **AT** function takes two numbers as arguments, one for the new cursor column and the other for the new cursor row. When called the current output buffer (anything before this that hasn't been printed yet) will be printed to the screen and the cursor moved to the new position.

```
10 FOR I=0 TO 9
20 PRINT AT(I,I),"HELLO, WORLD!"
30 NEXT
RUN
HELLO, WORLD!
 HELLO, WORLD!
  HELLO, WORLD!
   HELLO, WORLD!
    HELLO, WORLD!
     HELLO, WORLD!
      HELLO, WORLD!
       HELLO, WORLD!
        HELLO, WORLD!
         HELLO, WORLD!
          HELLO, WORLD!

READY.
■
```

*Moving the cursor using the **AT()** function. When the program is actually run the output will start at the top left corner of the screen.*

Note that the **AT()** function only works when the output is going to the screen. If you are writing to a file over a serial device (discussed below), cursor positioning makes no sense.

ALIGNING OUTPUT WITH TABS

In many programs, particularly those concerned with displaying calculations, summaries, or reports, it helps to be able to align output into columns. Cody BASIC doesn't handle every possible case, but the **TAB()** output function does allow you to align output to specific columns on the screen.

The function takes only one argument, the column number from 0 to 39. When it runs, it generates spaces in the output buffer until the next output position matches the desired position. This means that on a line-by-line basis you can ensure the same information will be printed on the same columns, so long as the data isn't so big that it overflows the available space.

```
10 FOR I=1 TO 10
20 PRINT I,TAB(5),I*I,TAB(20),"MESSAGE"
30 NEXT
RUN
1     1               MESSAGE
2     4               MESSAGE
3     9               MESSAGE
4     16              MESSAGE
5     25              MESSAGE
6     36              MESSAGE
7     49              MESSAGE
8     64              MESSAGE
9     81              MESSAGE
10    100             MESSAGE

READY.
■
```

*Aligning output to specific columns using the **TAB()** function.*

This function is also useful when writing to output files. As you'll learn in the upcoming section on reading and writing to files, it's usually easier to store one piece of information on each line when writing to a file. However, if you decide to store multiple pieces of information on the same line, aligning each piece to known columns will make it easier to split apart when you read it back in later.

CLEARING THE SCREEN

The simplest control code clears the screen. Character code 222 will clear the screen and move the cursor back to the very top. This can be useful to start from a known position in your Cody BASIC programs. It's also a good way to focus the user on what you want

them to see by clearing out any leftover input or output from earlier.

```
10 PRINT CHR$(222)
RUN

READY.
■
```

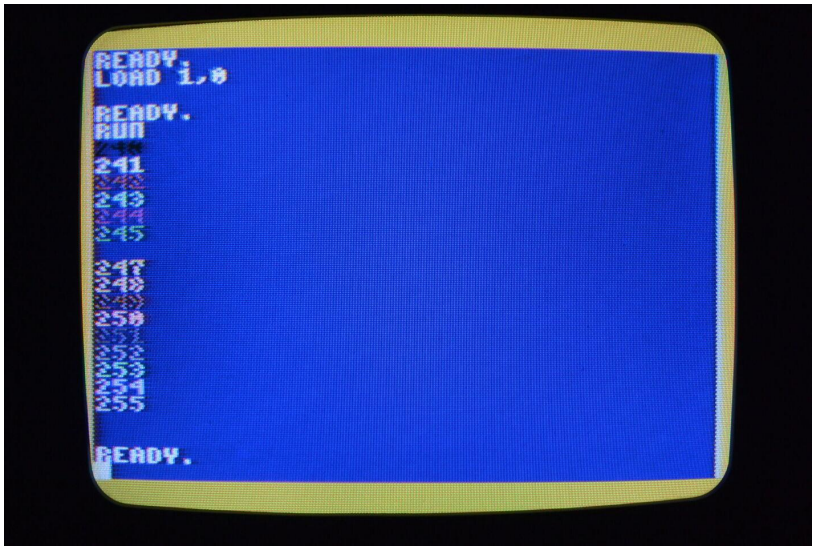
*Clearing the screen using the clear control code. When run in Cody BASIC the last **READY** statement will appear at the top of a new, blank screen.*

SETTING THE FOREGROUND COLOR

The foreground color can be changed using character codes between 240 and 255. Each code maps to one of the Cody Computer's 16 colors, each of which can be found in the reference in the back of the manual. To choose a specific foreground color, just take the color's number and add it to 240.

```
10 FOR I=0 TO 15
20 PRINT CHR$(240+I),240+I
30 NEXT
40 PRINT CHR$(241)

RUN
```

Printing out each foreground color using control codes.



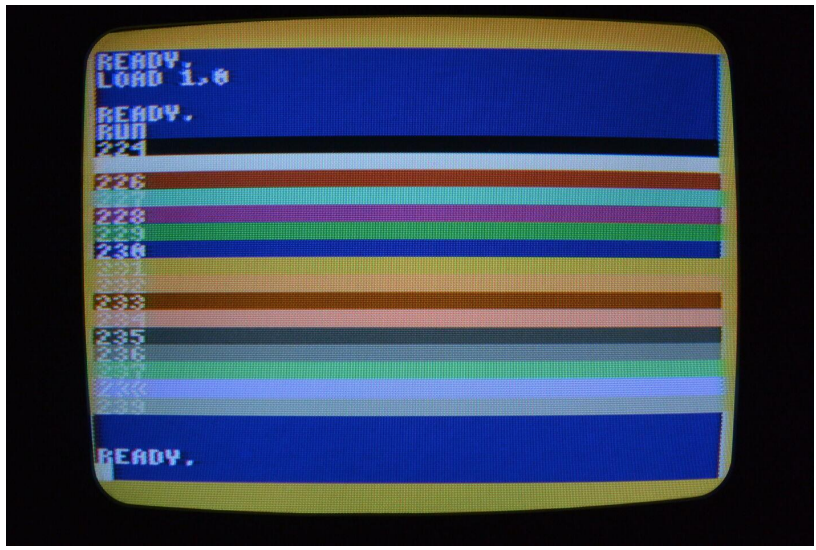
The Cody Computer's foreground colors displayed using control codes.

SETTING THE BACKGROUND COLOR

The background color can be changed using character codes between 224 and 239. This works in a very similar way to setting the foreground color except that the background is changed instead. Just add the color code to 224 to calculate the appropriate character code for the new background color.

```
10 FOR I=0 TO 15
20 PRINT CHR$(224+I),224+I
30 NEXT
40 PRINT CHR$(230)

RUN
```

Printing out each background color using control codes.



The Cody Computer's background colors displayed using control codes.

REVERSING FOREGROUND AND BACKGROUND

It'a also possible to reverse the current foreground and background colors. Character code 223 reverses the foreground and background colors. The current foreground color will be replaced with the current

background color, while the current background color is replaced with the current foreground color.

This is the Cody Computer's equivalent of the "reverse field" mode on Commodore computers. The Cody Computer has unique foreground and background attributes for each screen location and its character set doesn't contain inverted versions of each character. Instead it just swaps the attributes themselves.

```
10 INPUT S$
20 PRINT CHR$(223),S$,CHR$(223)
RUN
? HELLO, WORLD!
HELLO, WORLD!

READY.
■
```

Swapping foreground and background colors using the reverse control code.

PRINTING GRAPHICAL CHARACTERS

As mentioned elsewhere, the Cody Computer's CODSCII character set is just a customized, extended ASCII. The normal control codes, letters, digits, and punctuations are all the same as any other ASCII or ASCII-derived character set. As you've just learned, at the high end of the CODSCII range are control codes that can control various output attributes on the screen. However, there's one part of the CODSCII character set we haven't discussed yet.

Commodore computers used their own character set called PETSCII, named after the Commodore PET

computer it first appeared in. Because the Commodore PET had no graphics functionality of its own, the designers included graphical characters that could be used to make pictures and even games. This character set continued on for the rest of the Commodore 8-bit computer line.

The Cody Computer includes the graphical PETSCII subset in its own character set starting at character 128. You can use these characters in your own programs and games just like people did in the Commodore days, and all you need to do is include the appropriate character code for each one.

```
10 FOR I=0 TO 66
20 IF MOD(I,6)=0 THEN PRINT
30 PRINT 128+I," ",CHR$(128+I)," ";
40 NEXT
50 PRINT
```

Program that prints a table of the Cody Computer's PETSCII subset. In the actual output the ellipsis will be replaced by a table.

The program output showing the PETSCII subset in the Cody Computer's CODSCII character set.

FILE INPUT AND OUTPUT

Cody BASIC has the ability to read and write text files from within BASIC programs. Within a program, the **OPEN** and **CLOSE** statements can be used to redirect the program's input and output to one of the Cody Computer's two serial ports. From that point on, **PRINT** statements write to the serial port, while **INPUT** statements read from it. A **CLOSE** statement returns back to the screen and keyboard.

Note that this approach, while simple, also has its own challenges. Much like loading programs, the user must be careful that data lines aren't sent to the Cody Computer faster than the BASIC program can process them. Large per-line delays may be necessary. It also makes no provision for reading or writing binary data

as only text is supported. For binary data, dropping into machine language is recommended, and it may be advisable to write your entire program in assembly or another compiled language if speed is that critical.

> A similar strategy of reading and writing data files by input and output redirection was used in the OSI Challenger's version of Microsoft BASIC. In that system, LOAD and SAVE commands within a program directed output to the cassette, allowing INPUT and PRINT statements to read and write from the cassette port.

Note that when running programs that read and write files to the serial ports, the other device must be configured appropriately. The steps required are the same as those discussed in the previous chapter. The baud rate specified in Cody BASIC must match that configured for the external device, the external device must be configured for 8-N-1 (8 data bits, no parity bit, 1 stop bit), and a single ASCII linefeed should be set as the newline character. When reading from the device, line delays will be required on a per-program basis depending on the processing required.

WRITING TO A FILE

Writing to a file from within a Cody BASIC program requires you to open the correct I/O device, write your data to it, and then close the I/O device. For most purposes your I/O device will be device 1, the serial port wired to the Prop Plug connector at the back of

the computer. A second serial port is wired to pins on the expansion slot and can be used to interact with your own projects and custom peripherals.

Opening the I/O device is performed by the **OPEN** statement, which takes two arguments. The first is the I/O device number (1 or 2) and the second is a constant representing one of 15 different baud rates. This constant is the same as the value passed directly to the UART in the Propeller and can be any number between 1 (50 baud) and 15 (19200 baud). Once the port is opened, **PRINT** statements will print to the serial port until a **CLOSE** statement is encountered.

```
10 OPEN 1,15
20 PRINT "ENTERPRISE"
30 PRINT 5
40 PRINT "COLUMBIA"
50 PRINT 28
60 PRINT "CHALLENGER"
70 PRINT 10
80 PRINT "DISCOVERY"
90 PRINT 39
100 PRINT "ATLANTIS"
110 PRINT 33
120 PRINT "ENDEAVOUR"
130 PRINT 25
140 PRINT "EOF"
150 CLOSE
160 PRINT "DONE"
RUN
DONE

READY.
■
```

A program that writes the names of the space shuttles and number of flights to a text file.

Because the **INPUT** statement in Cody BASIC works on a per-line basis, it's important that the data you write also be readable on a per-line basis. One option, such as in this example, is to put each unique piece of data on its own line. The other option is to split up a line of data when read using the **STR$()** function, though this brings other complications with it.

```
ENTERPRISE
5
COLUMBIA
28
CHALLENGER
10
DISCOVERY
39
ATLANTIS
33
ENDEAVOUR
25
EOF
```

The data file generated by the above sample program. Note how each piece of data is on its own line.

READING FROM A FILE

Reading from a file is very similar to writing to one. The device must be opened using **OPEN** and closed using **CLOSE**. All the same caveats about baud rates and serial modes also apply. The main difference is that instead of writing data using **PRINT** you read data line by line using **INPUT**. Another difference is that, as your program is reading data, you may need to configure a line delay on the device sending you data so that your program can keep up.

As mentioned above, the **INPUT** statement in Cody BASIC works a little differently than in Commodore BASIC or similar. Each input variable reads an entire line, so each piece of data should also be on its own

line in the data file. The only way around this would be to read the line, then split out each part of it into its own substring, something we won't tackle here.

Remember that while a device is open, both input and output are redirected to it. That means that while you're reading from the external device, whatever you print will be sent to it, not to the screen. You will need a temporary storage area to keep whatever counts or tallies are needed until reading is done. In some cases this can be easy, while in other cases, designing your temporary storage can be difficult given the constraints of Cody BASIC.

```
10 OPEN 1,15
20 INPUT S$
30 IF S$="EOF" THEN GOTO 70
40 INPUT N
50 O$=O$+S$+" ("+STR$(N)+")"+CHR$(10)
60 GOTO 20
70 CLOSE
80 PRINT O$
RUN
ENTERPRISE (5)
COLUMBIA (28)
CHALLENGER (10)
DISCOVERY (39)
ATLANTIS (33)
ENDEAVOUR (25)


READY.
■
```

A program that reads the space shuttle data file from the previous example. As a simple example, a string is used to collect the output until processing is complete. Note the check for a special end token to determine the end of the file. (A blank line is another good option.)

Even when input and output have been redirected to a serial port, the **INPUT** statement still sends an ASCII question mark before waiting for the next line. Just like we discussed in the last chapter about loading programs, a terminal program or other application that recognizes this could send the next line as soon as it's asked for rather than waiting for a delay on each line. This would help speed up the loading of data files over serial connections.

INCLUDING DATA IN PROGRAMS

Another way to use data in a Cody BASIC program is hardcode it using **DATA** statements. Like Commodore BASIC and many other Microsoft BASIC dialects, Cody BASIC lets you add data in **DATA** statements and read it later using **READ** statements. Unlike other BASICs, however, Cody BASIC requires that all data be numeric in nature. Strings are not supported.

The data is read using **READ** statements. A **READ** statement takes one or more number variables as arguments and fetches the next entries from **DATA** statements, starting at the top of the program. If no more data exists, a logic error is raised to indicate an out of data condition.

DATA statements can be placed anywhere in the program. If one is encountered by the program, it is ignored. Only **READ** statements use **DATA** statements.

To reread data starting from the beginning of the program, the **RESTORE** statement can be used.

```
10 READ I
20 IF I<0 THEN GOTO 60
30 T=T+I
40 C=C+1
50 GOTO 10
60 PRINT "TOTAL ",T
70 PRINT "COUNT ",C
80 PRINT "AVERAGE ",T/C
90 DATA 3,10,12,7,6
100 DATA 3,15,8,2,-1
RUN
TOTAL 66
COUNT 9
AVERAGE 7

READY.
■
```

*Calculating totals and averages from numbers in **DATA** statements. A negative number is used as a sentinel value to stop processing.*

DATA and **READ** statements can be very helpful in programs that contain a lot of raw data or data tables. Games are a classic example as they contain sequences of bytes representing the game's sprites, tiles, backgrounds, and more. If a program needs to use portions of machine code to speed up operations or perform special operations, storing the assembled code in **DATA** statements is also common. Lastly, programs with mathematical computations can use **DATA** statements to store lookup tables for part of their calculations. Consider, for example, a program that estimates model rocket flights using tables of rocket engine data.

TIMEKEEPING

Cody BASIC has a limited form of timekeeping using the **TI** variable. More of a pseudovariable, **TI** stores the number of jiffies since the computer powered on. The value starts at zero, counts up through the positive numbers, wraps around through the negative numbers, and repeats. A single jiffy is 1/60th of a second, so the full range of **TI** is a little over 18 minutes. For longer time periods you can check in on the **TI** variable and update a seconds or minutes counter accordingly.

Using **TI** is preferable to hardcoded delays from loops in your Cody BASIC programs. However, direct comparisons between two values are not meaningful because **TI** will loop around through both positive and negative values. Instead, you must subtract the current value of **TI** from your previous value, then compare the difference. Because of the nature of signed arithmetic and modular arithmetic, this will calculate the correct difference in jiffies.

```
10 INPUT D
20 D=D*60
30 I=TI
40 IF TI-I<D THEN GOTO 40
RUN

READY.
■
```

*Sample program that waits for a given number of seconds before stopping. Note the conversion of the delay from seconds to jiffies (multiplying by 60), as well as the inline calculation subtracting the current **TI** from the initial value.*

READING AND WRITING MEMORY

While Cody BASIC is more high-level than assembly language, it's still very low-level compared to most modern languages. In the 8-bit era, interpreted BASICs commonly manipulated hardware directly, generally through reading and writing to memory. Communication with support chips and peripherals often occurred by direct reads and writes to registers, and passing data to machine language routines required similar access to reserved memory locations.

Cody BASIC, like most BASICs, provides the **POKE** statement to write to memory and the **PEEK** statement to read from it. It's important to be careful when using these parts of Cody BASIC as you can easily freeze up the Cody Computer or worse. However, once you understand how they work and learn the Cody Computer's memory map, most of the computer's

features will be open to you from BASIC alone. While many programs at this level are better written in assembly language, BASIC provides a solid foundation to begin from.

It's worth noting that the 65C02's address space ranges from 0 to 65535 because its address bus is 16 bits wide. Cody BASIC numbers are also 16 bits, but they are signed numbers, not unsigned, and they range from -32768 to 32767. Fortunately, Cody BASIC automatically parses unsigned number literals as the equivalent signed value, so you won't have a problem working with memory addresses in Cody BASIC. For example, you can type 50176 (the default start of screen memory) directly into your program and have it work. However, if you print the number out, Cody BASIC will print -15360, the signed number equivalent for the same bit pattern as 50176.

WRITING TO MEMORY

The **POKE** statement writes to memory. It takes two arguments, a memory address and a value to write to that address. The address can be anything within the 65C02's address space, ranging from 0 to 65535 (or the signed-number equivalent as discussed above). The value written to that address should be a byte from 0 to 255.

```
10 S=50176
20 C=55296
30 FOR I=0 TO 999
40 POKE S+I,128+MOD(RND(),32)
50 POKE C+I,RND()
60 NEXT
RUN

READY.
■
```

Program that directly writes to screen and color memory to draw graphical characters in a variety of colors. Exactly why this works is discussed in the chapter on graphics programming.

A **POKE** statement won't work correctly in memory areas that are read-only on the Cody Computer. The top 8 kilobytes of the Cody Computer's memory are essentially a ROM with Cody BASIC and the default character set, and these can't be modified by writing to them. Some registers are also read-only.

READING MEMORY

The **PEEK()** function reads a memory address. It takes one argument, a memory address just like those used in the **POKE** statement. It returns the byte at that address in memory as a number between 0 and 255.

```
10 PRINT "PRESS Q TO QUIT..."
20 IF AND(PEEK(16),1)=1 THEN GOTO 10
30 PRINT "Q PRESSED"
RUN
PRESS Q TO QUIT...
PRESS Q TO QUIT...
PRESS Q TO QUIT...
Q PRESSED.

READY.
■
```

Program that reads a memory location representing the first keyboard row. The memory location is automatically updated by a keyboard scanning routine in Cody BASIC. Your program can read the memory location and determine what keys are held down at the moment.

PEEK() functions aren't dangerous like **POKE** statements because they don't change the contents of memory. However, it's still important to understand the memory map and use the correct addresses. Otherwise your programs might not work correctly, and at such a low level, it can be difficult to debug them.

USING MACHINE CODE

High-performance programs for the Cody Computer should probably be written in assembly language and loaded as binary programs. However, it's possible to include some of the benefits of assembly language in your Cody BASIC programs. To do this, you write small portions of assembly language (either

using an assembler or by hand), then load the machine code into memory as part of your program.

When you want to call the machine code, you use Cody BASIC's **SYS** command, which temporarily passes control to a machine-language subroutine of your choosing. It even handles swapping the 65C02's accmulator, X, and Y registers in and out of special memory locations so you can use them in your code.

This topic is difficult enough that it's worth a detailed walkthrough. For a very simple example, imagine we want a machine code routine that takes the values in the accumulator, X register, and Y register, then increments each by one before returning to BASIC. First we need to write the assembly language routine that would do this for us. (Our example is simple enough to assemble by hand, but an assembler is recommended for more advanced ones.)

```
INC A       ; $1A (decimal 26)
INX         ; $E8 (decimal 232)
INY         ; $C8 (decimal 200)
RTS         ; $60 (decimal 96)
```

A snippet of 65C02 assembly that increments the accumulator, X, and Y registers.

Once we have the assembly language code, we need to load it into a memory location that's otherwise not in use. Somewhere very high in BASIC program memory or another free spot in the memory map are ideal. We include the numbers for our assembled machine code in one or more **DATA** statements, using **READ** to get each byte and **POKE** to load it into memory starting at that address.

To actually call the code, we would use the **SYS** statement. It takes only one argument, the address to

call. It calls that address using the 65C02's **JSR** instruction and returns back to your program once your machine code executes an **RTS** instruction.

You can pass parameters back and forth to your machine code from Cody BASIC using **POKE** and **PEEK** to addresses used by the machine code routine. However, **SYS** also has another way to do much of this for you. It copies the values at the first three memory locations, $00 through $02, into the accumulator, X register, and Y register before calling your machine code. When done, it copies the current values of those registers back to those same memory locations. Your BASIC program only needs to **POKE** values into those addresses before the call, then **PEEK** them to get the results after it's done.

```
10 P=25856
20 READ B
30 IF B<0 THEN GOTO 70
40 POKE P+I,B
50 I=I+1
60 GOTO 20
70 INPUT A
80 INPUT X
90 INPUT Y
100 POKE 0,A
110 POKE 1,X
120 POKE 2,Y
130 SYS P
140 PRINT "A=",PEEK(0)
150 PRINT "X=",PEEK(1)
160 PRINT "Y=",PEEK(2)
170 DATA 26,232,200,96,-1
RUN
? 1
? 4
? 9
A=2
X=5
Y=10

READY.
■
```

*Using the above machine code in a Cody BASIC program. The instructions are poked into memory, user-entered data is moved into designated memory locations, and the routine called using the **SYS** statement. When done the updated data is read back and displayed.*

Using machine code from within a Cody BASIC program isn't an easy thing to do, but in certain situations, it can be quite beneficial. Effectively doing so requires a good understanding not only of Cody

BASIC but of the Cody Computer's memory map and of 65C02 assembly language itself.

> If you find yourself using this approach, it might be worth asking yourself if you're better off just writing the entire program in assembly or a compiled language. On the other hand, some BASIC programs in the 8-bit era took advantage of similar features. The most critical parts of the code were written in assembly language, but most of the program was written in BASIC.

PROGRAMMING HINTS

Along with all the details involved in Cody BASIC programming, it's important to be aware of some of the other important aspects when writing your programs. Many of these are less technical, but no less important. You want your programs to be understandable both for yourself and for others. You also want your programs to be easily changeable as your requirements change, or if someone else uses one of your programs and needs to modify it. These skills are generally the same as in any programming language, but Cody BASIC's quirks add some additional things to consider.

DOCUMENTING YOUR PROGRAMS

In your program you should make use of **REM**, or remark, statements. These are the 8-bit BASIC equivalent of code comments and were used to

document programs. Programs often started with remarks about the name of the program, its author, and a description of what it did. In the program itself, remarks often marked different sections or routines within the program. They were also added to provide some additional information on particularly complicated parts.

Unlike comments in modern compiled languages, **REM** statements take up space in the interpreter, have to be loaded and saved, and also have to be skipped over at runtime. Therefore, while they're a no-op, they don't come without a cost. That said, it's good to document your programs.

Many programs were shared in books or magazine articles that provided the main documentation for both users and programmers (in that era, more often than not one and the same). In today's world it might be helpful to include a text file, a Markdown document, or even a simple HTML file with your programs.

USING LINE NUMBERS

Along with documenting your programs, it's important to structure them so that they're easy to read and modify. While that's harder in an environment like Cody BASIC, it's not impossible. Because Cody BASIC, like most retro basics, has a line-oriented editing system, much of your structure will relate to the line numbering you use.

One tactic for maintainable programs is to be generous with your use of line numbers. For example, numbering lines by multiples of 10 gives you additional room to go back and make changes without

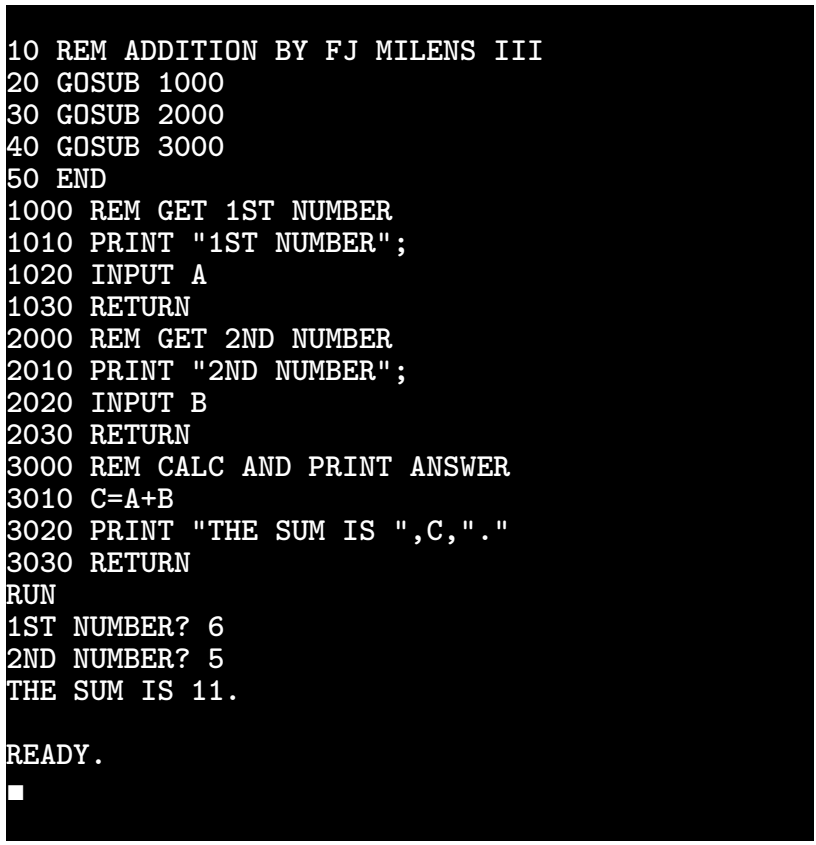
having to renumber an entire program. It also gives someone else the ability to experiment and make changes more easily.

It can also be helpful to have gaps between line numbering in unrelated parts of the program. Doing this along with **REM** statements at the beginning can help show where your subroutines begin and end, as well as what they do.

You also have the option to cheat and use a modern PC. Cody BASIC programs are stored as plain text, unlike Commodore BASIC programs that were kept in a tokenized format. They're also written in extended ASCII with the important non-graphical characters understood by any modern computer. This means you can load saved Cody BASIC files in any text editor that won't mangle the file's encoding or line endings, make changes, and send them along. You can also write programs from scratch in a text editor and then send them over to the Cody Computer just like any saved program. You just need to be careful about line endings. You also must ensure that your programs end with a blank line indicating the end of the file.

AN EXAMPLE PROGRAM

Below is an example program using some of the above advice. It's a very contrived example that only adds two numbers together, and in real life, such a simple program wouldn't need nearly so much boilerplate. The example is intentionally simple to demonstrate how the techniques above might be used in a larger program, without having to wade through the code of a larger and more complex program itself.



*An admittedly overengineered program demonstrating some of the techniques in this section. Note the **REM** statements, line numbering, and spacing of subroutines.*

7

Graphics Programming

INTRODUCTION

The Cody Computer is equipped with its own system for generating video graphics, the VID or Video Interface Device. Implemented as a software peripheral inside the Parallax Propeller chip, it presents itself as hardware on the 65C02's system bus. Writing to dedicated registers and memory regions allows you to construct 8-bit mulitcolor graphics.

The VID produces a character-based screen with a resolution of 160 pixels by 200 pixels. Each character is four pixels by eight pixels in size, using a fat-pixel ratio similar to that used by the Commodore 64's multicolor graphics mode. Up to 256 different characters can exist within a single character set, and multiple character sets can be used on different parts of the screen. A bitmapped mode is available that essentially configures all of screen memory to become addressible in character-like tiles. Screen contents can also be fine-scrolled in hardware by setting appropriate values.

Sprites are also supported by the VID, allowing you to have multicolor graphics that hover over the normal screen. These are 12 pixels across and 21 pixels tall, and each also has a fat-pixel ratio. The memory layout is very similar to the Commodore 64's multicolor sprites except that the colors are less constrained. The Cody Computer's VID doesn't support other sprite features like scaling or collision detection. It's there to move sprites around and draw them.

The VID supports 16 colors inspired by the Commodore 64's color palette. These colors can be

used on the screen and on sprites, though are there are some limitations in how many colors can be used together. Characters on the screen have two unique colors and two colors shared with the entire screen, while sprites have two unique colors and one color shared with other sprites.

Lastly, the VID allows you to change graphics on the fly using what are called row effects. Similar to old-school raster interrupts, where video options were switched out at specific character rows on the screen, you can program the VID to change sprite banks, character banks, scroll amounts, and even the colors on each character row as it draws a frame. Further intervention by the programmer is not required.

CHANGING THE BORDER COLOR

A good introduction to graphics programming is learning how to change the Cody Computer's border color. The border can be set to any of the sixteen colors supported by the Cody Computer. To change it, all you have to do is update the low four bits of the color register located at position **$D002** in memory.

The high four bits of the color register contain the position of color memory, something we don't want to change right now. Instead, what we have to do is read the current value of the color register, mask out the low four bits with an **AND**, and then **OR** them together with our desired color code.

This can be done from assembly language, but the Cody BASIC **PEEK** and **POKE** will let us directly read and write memory. We just need to use the correct address, 53250, the decimal equivalent of **$D002**.

```
10 PRINT "BORDER COLOR (0-15)";
20 INPUT C
30 IF C<0 THEN END
40 POKE 53250,OR(AND(PEEK(53250),240),C)
50 GOTO 10
RUN
0
1
2
-1

READY.
■
```

Simple program that changes the border color. The user types in a number which is put into the low four bits of the color register. Entering 7 will return the screen to its normal yellow border.

WORKING WITH SCREEN MEMORY

Now that you know how to change the border color using **PEEK**s and **POKE**s, we'll start using those same operations to change the screen contents themselves. The Cody Computer's screen is set up as a range of 1000 bytes, each of which represents a single character on a 40 column by 25 row screen. You can change the screen contents by changing the contents of memory in this region, and in fact that's what Cody BASIC does internally to display text.

UPDATING SCREEN MEMORY

As a simple example, we can fill the screen with data. By default the screen starts at memory address

$C400 or decimal 50176. If we populate the 1000 bytes starting at that location with numbers corresponding to CODSCII characters, we'll see them show up on the screen. Each number references a single character in the character set, so the number we **POKE** will be the character that we see.

```
10 FOR I=0 TO 999
20 POKE 50176+I,97+MOD(I,26)
30 NEXT
```

*Directly populating screen memory. Each **POKE** writes one of the lowercase characters in the CODSCII character set to a position in screen memory. When run, the program will overwrite the screen with lowercase letters.*

RELOCATING SCREEN MEMORY

The default screen starts at **$C400**, but it's possible to move the screen elsewhere, a capability often used in games and other graphics-intensive applications. Theoretically, screen memory can exist anywhere in a 16-kilobyte area of memory starting at memory address **$A000**, with the restriction that the memory must be on a 1-kilobyte boundary.

However, in practice we have to avoid certain parts of memory. The VID itself uses a page at **$D000** for its own register banks. The SID and UARTs take up a page at **$D400**. Memory must also be set aside for color memory and character memory, two video-related topics we'll get to in this chapter. When using such techniques in your own programs, begin with the Cody

Computer's memory map and sketch out where you want things to be placed.

Setting up another region to use as screen memory is just like the previous example. You just need to write the appropriate bytes to reference the characters that should be drawn. However, once you've done that, you still need to tell the Cody Computer where it lives. The base register at **$D003** sets the starting location of both character memory and screen memory, with screen memory stored in the high four bits.

To determine what value to plug into the high four bits, you need to do a simple math calculation. Four bits can contain one of 16 values, which is convenient because a 16-kilobyte area of memory can contain 16 regions aligned at 1-kilobyte boundaries (just what we have). Just subtract the start of your desired screen memory location from **$A000**, then divide by 1024 to get the result. If your screen memory started at **$A000** you would use a value of 0 because you're in the initial 1-kilobyte region. For the default screen memory location at **$C400**, you would use a value of 9.

```
10 A=41984
20 B=(A-40960)/1024
30 FOR I=0 TO 999
40 POKE A+I,97+MOD(I,26)
50 NEXT
60 POKE 53251,OR(AND(PEEK(53251),15),B*16)
70 T=TI
80 IF TI-T<300 THEN GOTO 80
90 POKE 53251,OR(AND(PEEK(53251),15),9*16)
```

Temporarily relocating screen memory. Another region in memory is specified and its base calculated. That same region is populated with lowercase letters. The base register is then updated with the new screen memory base, the program waits for five seconds, and then sets the screen memory base back to the default.

WORKING WITH COLOR MEMORY

Screen memory specifies what characters to draw on the screen, but color memory specifies what colors to draw them in. Characters on the Cody Computer can have up to four colors, two of which can be unique for each column-row position on the screen. These two colors are loaded from the corresponding color memory for the screen.

Much like screen memory, color memory is a contiguous array of 1000 bytes, and there is a one-to-one correspondence between screen memory and color memory locations. Cody BASIC updates color memory locations for you in **PRINT** statements, but you can also do so by yourself using **POKE**s.

UPDATING COLOR MEMORY

Color memory begins by default at address **$D800** or decimal 55296. Much like for screen memory, we need to **POKE** data into this region to see the contents of the screen change. In this case, instead of poking in numbers for characters, we poke in numbers that represent the foreground and background colors for each character. The foreground color code goes into the top four bits of the number and the background color code goes into the bottom four bits.

```
10 A=55296
20 PRINT "FOREGROUND COLOR (0-15)";
30 INPUT F
40 PRINT "BACKGROUND COLOR (0-15)";
50 INPUT B
60 C=F*16+B
70 FOR I=0 TO 999
80 POKE A+I,C
90 NEXT
RUN
FOREGROUND COLOR? 13
BACKGROUND COLOR? 0

READY.
■
```

Program that updates the default color memory with new foreground and background colors.

RELOCATING COLOR MEMORY

Just like screen memory, color memory can be moved to a different location. As with screen memory, the region of memory starting at **$A000** is divided

into 1-kilobyte blocks, and the same caveats and restrictions on their use apply here as well. To calculate the base for a particular color memory location, you can use the same formula that you used for screen memory in the prior section.

Once you've decided on a new location for color memory, you need to update the color register at **$D002**. You updated the low four bits of this register to change the border color at the beginning of this chapter, but now you'll update the high four bits to specify the base location for color memory.

```
10 PRINT "FOREGROUND COLOR (0-15)";
20 INPUT F
30 PRINT "BACKGROUND COLOR (0-15)";
40 INPUT B
50 C=F*16+B
60 A=41984
70 B=(A-40960)/1024
80 FOR I=0 TO 999
90 POKE A+I,C
100 NEXT
110 POKE 53250,OR(AND(PEEK(53250),15),B*16)
120 T=TI
130 IF TI-T<300 THEN GOTO 130
140 POKE 53250,OR(AND(PEEK(53250),15),14*16)
RUN
FOREGROUND COLOR? 15
BACKGROUND COLOR? 12

READY.
■
```

Program that temporarily relocates color memory to a different location. A second color memory region is set up with the colors selected by the user. The color register is then temporarily updated to point to the new region before returning back to the default location.

CHARACTERS AND CHARACTER MEMORY

Screen memory specifies what characters to draw and color memory specifies what colors to draw them in, but character memory specifies what the characters themselves look like. A character set on the Cody Computer consists of up to 255 unique characters, each of which is four pixels across and eight pixels tall.

CHARACTERS IN ROM

The Cody Computer contains the full default CODSCII character set in a 2-kilobyte area of memory starting at **$E000** or decimal 57344. When the computer starts up, the BASIC ROM copies this character set into memory at **$C800**, where it can be seen by the Video Interface Device and used to draw the screen. You can always access these characters yourself to see what data they contain in numeric format.

```
10 INPUT S$
20 C=ASC(S$)
30 A=57344+C*8
40 FOR I=0 TO 7
50 PRINT PEEK(A+I)
60 NEXT
RUN
? A
0
4
17
17
21
17
17
17

READY.
■
```

A Cody BASIC program that reads a character's bytes from the character ROM.

This means that in your own programs, you don't have to worry about clobbering the existing characters in video memory, or preserving them somewhere else.

You can just modify or overwrite them, and then copy the original characters from the ROM back to video memory to clean up.

```
10 S=51200+97*8
20 D=51200+65*8
30 FOR I=0 TO 207
40 POKE D+I,PEEK(S+I)
50 NEXT
60 T=TI
70 IF TI-T<300 THEN GOTO 70
80 S=57344+65*8
90 D=51200+65*8
100 FOR I=0 TO 207
110 POKE D+I,PEEK(S+I)
120 NEXT
```

A program that copies the lowercase characters over the uppercase characters, temporarily making everything on the screen lowercase. When done it copies the original uppercase characters from ROM back into video memory. Note that this isn't changing the screen memory contents at all. Instead, it's changing the characters themselves.

CUSTOM CHARACTERS

As mentioned, characters on the Cody Computer actually have four colors. Two of the colors, 0 and 1, are unique to each character position on the screen. Those colors are read from the color memory you learned about earlier. The other two colors, 2 and 3, are shared as common colors by every location on the screen.

The shared colors are kept in the screen colors register at location **$D005** or decimal 53253 and have a similar format to color memory. Color 2 is stored in

the low four bits and color 3 is stored in the high four bits of the register.

You don't notice these colors when the Cody Computer starts up because the default character set only uses colors 0 and 1, the two colors that are unique to a given screen position. This works nicely for the character set as we can specify the foreground and background colors independently for each position on the screen. However, in more graphical applications such as games, it helps to have more colors.

To use them, you have to define your own characters. Each character consists of eight bytes, with each pixel in a character represented by two bits. Bit combinations **00** and **01** reference the two screen colors at that location, while bit combinations **10** and **11** reference the common colors in the screen register. Each character is four pixels wide and eight pixels high, and the data in character memory is stored from the top of the character to the bottom. Within each byte, the pixel data goes from the leftmost pixel in the two highest bits to the rightmost pixel in the two lowest bits.

To design your own character you work out the bit combinations for your own 4-by-8 pixel pattern, then **POKE** that data somewhere in the current character set. Remember that characters don't actually have to be characters as such. They can be any kind of image, including tiles for games or portions of a background picture. You can even use different character sets on different screen rows if you need more unique characters (for example, using one character set for the user interface and another for the game world

itself). This can even be a substitute for bitmap graphics if used wisely.

```
10 FOR I=0 TO 7
20 READ M
30 POKE 51200+255*8+I,M
40 NEXT
50 FOR I=0 TO 999
60 POKE 50176+I,255
70 POKE 55296+I,MOD(RND(),16)*16+MOD(RND(),16)
80 NEXT
90 POKE 53253,1
100 DATA 80,80,80,80,250,250,250,250
```

Example program that defines a new character that consists of four colored blocks, then fills the screen with it. Two of the new character's colors are unique to the character itself and stored in the color memory. The other two are shared by all the characters on the screen and are stored in the screen colors register.

RELOCATING CHARACTER MEMORY

Like screen memory and color memory, character memory can be relocated. Like screen memory, the base location of character memory is specified in the base register at **$D003** or decimal 53251. The base for character memory is stored in the low four bits of the register, and the base can be calculated similar to that for screen and color memory: Subtract the base address from **$A000** or decimal 40960, then divide by 2048 in this case. Character sets take 2 kilobytes and must be aligned on a 2048-byte boundary, unlike screen and color memory that take 1000 bytes and must be aligned on a 1024-byte boundary.

```
10 A=40960
20 B=(A-40960)/2048
30 FOR I=0 TO 2047
40 POKE A+I,MOD(I,2)*85
50 NEXT
60 POKE 53251,OR(AND(PEEK(53251),240),B)
70 T=TI
80 IF TI-T<300 THEN GOTO 80
90 POKE 53251,OR(AND(PEEK(53251),240),5)
```

A program that fills another 2-kilobyte region of memory with test patterns, then temporarily points the base of character memory to it. In a real application the character data would be a new character set, game tiles, or similar. Note the 2-kilobyte alignment of the character set's start address and division by 2048 for calculating the base.

Relocating character memory becomes very important when used in combination with row effects, which we'll cover later in this chapter. Row effects let you specify a different base for the character set on each character row, allowing you to switch out character sets within a single video frame.

This technique can be used for video games, for example using different character sets for the main game area as opposed to the surrounding graphics and status displays. It's also how the Cody Computer can display fully-bitmapped graphics by breaking a bitmap into a series of tiles.

WAITING FOR BLANKING

In this section you've been making a lot of changes to the Cody Computer's video registers. One thing we

haven't discussed yet is what happens if you make changes when the video hardware is in the middle of drawing a frame. The answer is that while it won't break anything, there's the chance for screen tearing, jerky motion, and other weird visual glitches popping up in the middle of a frame.

One way to avoid those problems is to update the video registers and the active video memory only when the video device isn't generating a frame. The blanking register at **$D000**, or decimal 53248, indicates the current state. A zero indicates that the visible area of the screen is being drawn, while a 1 indicates that the blanking area or top and bottom borders are being drawn instead.

A common technique is to poll the blanking register until it transitions from a 0 to 1, then perform any required updates for the next frame. This usually works better in assembly language because of its increased speed, but we can still use the same approach in Cody BASIC as an example.

```
10 IF PEEK(53248)=0 THEN GOTO 10
20 PRINT "NEW FRAME"
30 IF PEEK(53248)=1 THEN GOTO 30
40 GOTO 10
RUN
NEW FRAME
NEW FRAME
NEW FRAME
NEW FRAME
BREAK IN 10


READY.
■
```

*A program that prints a message whenever a new frame begins, then waits for it to end before repeating. The program will run forever until you break using the **Cody** + **Arrow** key combination. Note that the program likely won't print on every frame in reality because of the time required for Cody BASIC to execute each line.*

SCROLLING THE SCREEN

The Cody Computer's Video Interface Device also has features to support vertical and horizontal scrolling with hardware assistance. Two types of scrolling exist with different levels of support. One type of scrolling, fine scrolling, allows you to adjust the vertical and horizontal position up to a full column or row. Once you've adjusted it up to that level, you need to use coarse scrolling, where scrolling occurs at a column or row basis. Fine scrolling is supported by the scroll register, while coarse scrolling is usually

implemented as a side effect of relocating screen memory.

FINE SCROLLING WITH REGISTERS

Two different registers are involved in fine scrolling. Fine scrolling is enabled using the control register at **$D001** or decimal 53249. When set to a 1, bit 1 enables vertical scrolling and bit 2 enables horizontal scrolling. Vertical and horizontal scrolling can be enabled individually or at the same time.

Enabling scrolling affects the screen dimensions. Vertical scrolling decreases the displayed vertical screen size by one row. Horizontal scrolling on decreases the displayed horizontal screen size by two columns. The actual screen and color memory layout are unaffected but the space on the screen is replaced by expanded borders.

Once scrolling has been enabled for a particular direction, the amount to scroll must be specified in the scroll register at **$D004** or decimal 53252. The horizontal scroll amount is stored in the higher four bits while the vertical scroll amount is stored in the lower four bits. Horizontal scrolling supports a value between 0 and 3 while vertical scrolling supports a value between 0 and 7. The difference occurs because pixels are wider than they are tall on the Cody Computer, much like how a character has 4 horizontal pixels but 8 vertical pixels.

```
10 PRINT "H SCROLL (0-3)";
20 INPUT H
30 IF H<0 THEN GOTO 100
40 PRINT "V SCROLL (0-7)";
50 INPUT V
60 IF V<0 THEN GOTO 100
70 POKE 53249,OR(PEEK(53249),6)
80 POKE 53252,H*16+V
90 GOTO 10
100 POKE 53249,AND(PEEK(53249),249)
110 POKE 53252,0
RUN
H SCROLL? 2
V SCROLL? 4
H SCROLL? -1

READY.
■
```

A program that lets you experiment with vertical and horizontal scrolling at the same time. The code accepts vertical and horizontal scroll values from the user, then turns on scrolling and pokes the values into the scroll register. At the end the normal settings are restored.

COMBINED SCROLLING

Fine scrolling works well for simple effects, but to make a scrolling game it's not enough by itself. For that you need to combine it with coarse scrolling, where you move the entire screen by a row or column. Unfortunately, much like its Commodore inspiration, the Cody Computer has no direct support for coarse scrolling. Instead, what you do is draw a second screen, then flip to it when you need to scroll, using the same

techniques you learned earlier in this chapter for relocating the screen and color memory.

That's a lot of memory to draw, and moving that much data around on a per-frame basis is typically reserved for assembly language applications. Even then, it's typically an optimized process where part of the screen and color memory is drawn behind the scenes during each fine-scrolled frame so that it's all ready to go. In some respects the Cody Computer makes this easier because the color memory can also be relocated, unlike its fixed position on the Commodore 64.

However, just because we can't do it fast enough in Cody BASIC doesn't mean we can't at least give a simple example of how it works. The following program demonstrates most of the techniques needed, but it keeps the screen design simple so that we only have to generate two example screens at the start. It also doesn't change the colors so we don't need to do anything about the color memory.

```
10 A(0)=40960
20 A(1)=41984
30 B(0)=(A(0)-40960)/1024
40 B(1)=(A(1)-40960)/1024
50 FOR I=0 TO 999
60 C(0)=20
70 C(1)=20
80 IF MOD(I,2)=1 THEN C(0)=194
90 IF MOD(I,2)=0 THEN C(1)=194
100 POKE A(0)+I,C(0)
110 POKE A(1)+I,C(1)
120 NEXT
130 S=0
140 POKE 53252,0
150 POKE 53249,OR(PEEK(53249),4)
160 M=S/4
```

```
170 IF M=0 THEN B(2)=B(0)
180 IF M=1 THEN B(2)=B(1)
190 IF PEEK(53248)=0 THEN GOTO 190
200 POKE 53252,MOD(S,4)*16
210 POKE 53251,OR(AND(PEEK(53251),15),B(2)*16)
220 S=MOD(S+1,8)
230 IF AND(PEEK(16),1)=0 THEN GOTO 260
240 IF PEEK(53248)=1 THEN GOTO 230
250 GOTO 160
260 POKE 53251,OR(AND(PEEK(53251),15),9*16)
270 POKE 53249,AND(PEEK(53249),251)
280 POKE 53252,0
```

A simple combined scrolling example in Cody BASIC. Two screens are generated with repeating patterns offset by one column. Horizontal scrolling is enabled and the screen is fine-scrolled one pixel on each frame. Every fourth frame the screen memory is toggled between the two screen regions we set up to handle the coarse scrolling. When the user presses the Q key, the program terminates and restores the normal video configuration.

MOVING GRAPHICS WITH SPRITES

The Cody Computer supports sprites, movable graphical objects on the screen often used in games. Sprites are independent of the screen background and hover over it. Each sprite is 12 pixels wide and 21 pixels tall with a total of three colors plus a transparent option. Two colors are unique to each sprite while one is shared by all the sprites on the screen. Sprites can be positoned anywhere on the screen as well as partially off the screen on both the vertical or horizontal axes.

Sprite data uses a total of 63 bytes of memory, with the amount being rounded up to 64 as a power-of-

two. Each byte contains four pixels in a multicolor format like those used by the character memory. Sprite memory is organized from left to right, with the top-left portion of the sprite beginning at the first location in memory. Within each byte, the left-most pixel data is stored in the higher bits and moves to the lower bits.

Each color is represented by two bits, with a value of 0 indicating a transparent pixel. Values of 1 and 2 represent the two unique sprite colors, while a value of 3 represents the common color shared by all sprites on the screen. Sprite memory is organized from left to right, with the top-left portion of the sprite beginning at the first location in memory.

Programming sprites is somewhat difficult in the beginning. In addition to the sprite data that defines the image of a sprite, registers must be programmed to set up the sprite, specify its location, unique colors, and base address of its image data. In order to support a large number of sprites on the screen, an entire page of memory is set aside as sprite register banks, and this must also be taken into account.

DISPLAYING A SPRITE

To display a single sprite we have to do a few things first. We need to copy the sprite's image data into a 64-byte-aligned location in the 16-kilobyte area beginning at **$A000**. As with similar operations, we also need to ensure that it won't collide with registers or data already there.

Once we have a location picked out, we need to use it to calculate the sprite's base pointer, which is

calculated in a similar way to the screen, color, and character memory base pointers. You subtract your sprite's starting address from the start of the region at **$A000**, then divide the result by 64 to determine the base pointer. Conveniently there are 256 possible locations aligned at 64-byte boundaries, so this value fits into a single byte.

Once the data is loaded for a sprite, you need to program the sprite registers to tell the computer how to display it. Sprite registers begin at location **$D080** or 53376 decimal, and each sprite takes up four consecutive bytes starting at the beginning. The first byte specifies the sprite's x-position, the second byte specifies the sprite's y-position, the third byte specifies the sprite's two unique colors, and the fourth and final byte specifies the base pointer for the sprite's image data. (Multiple sprites can reuse the same image data, such as in old games where the bad guys reused the same picture in different colors.)

The sprite's position on the screen, notably, does not start at (0,0) at the top-left corner. Sprites can slide in from the sides of the screen and be only partially displayed. To support this, a margin is added to the normal screen dimensions. Because sprites are 12 pixels wide, a 12 pixel margin is added to either side of the screen. Likewise, because sprites are 21 pixels tall, a 21 pixel margin is added to the top and bottom. This margin isn't displayed on the screen, but it allows the sprite to be partially positioned off the screen. This also means that the first screen location that would fully display the sprite is at (12,21).

A sprite's unique color data is stored in a format like the color memory. Two colors are stored in one byte,

with sprite color 1 stored in the lower half of the byte and sprite color 2 stored in the upper half. The common color, color 3, shared by all sprites is stored in the sprite register at **$D006** or decimal 53254, where it's kept in the low half of the byte. The color codes are the same as those used in color memory.

```
10 A=41984
20 B=(A-40960)/64
30 FOR I=0 TO 63
40 READ M
50 POKE A+I,M
60 NEXT
70 POKE 53254,0
80 POKE 53376+2,9*16+14
90 POKE 53376+3,B
100 P(0)=12
110 P(1)=21
120 D(0)=1
130 D(1)=1
140 IF PEEK(53248)=0 THEN GOTO 140
150 POKE 53376+0,P(0)
160 POKE 53376+1,P(1)
170 P(0)=P(0)+D(0)
180 P(1)=P(1)+D(1)
190 IF P(0)=12 THEN D(0)=-D(0)
200 IF P(0)=160 THEN D(0)=-D(0)
210 IF P(1)=21 THEN D(1)=-D(1)
220 IF P(1)=200 THEN D(1)=-D(1)
230 IF AND(PEEK(16),1)=0 THEN GOTO 260
240 IF PEEK(53248)=1 THEN GOTO 230
250 GOTO 140
260 POKE 53376+0,0
270 POKE 53376+1,0
280 DATA 0,20,0,1,85,64,5,85
290 DATA 80,5,85,80,21,125,84,21
300 DATA 215,84,21,213,84,21,213,84
310 DATA 21,215,84,5,125,80,5,85
320 DATA 80,5,85,80,13,85,112,12
330 DATA 93,48,12,93,48,3,28,192
340 DATA 3,12,192,3,12,192,0,142
```

```
350 DATA 0,0,170,0,0,170,0,131
```

A sprite demo that bounces a balloon sprite around on the screen. The sprite's data is kept in ***DATA*** *statements and* ***POKE****d into memory. The sprite's position and velocity are kept in arrays and updated on each frame. The code waits for the blanking interval and updates the sprite position using the numbers from the arrays. Pressing the Q key exits the program and restores the default settings.*

A single sprite in the form of a balloon.

DISPLAYING MULTIPLE SPRITES

Up to eight sprites can be displayed on the same part of the screen at any one time. You only need to set up the other sprite registers just as you did the first one in the previous example. As mentioned before, each sprite is more or less independent of the screen,

and in fact sprites are more or less independent of each other.

```
10 A=41984
20 B=(A-40960)/64
30 FOR I=0 TO 63
40 READ M
50 POKE A+I,M
60 NEXT
70 POKE 53254,0
80 FOR I=0 TO 7
90 POKE 53376+I*4+2,1*16+(I+7)
100 POKE 53376+I*4+3,B
110 X(I)=13+MOD(RND(),147)
120 Y(I)=22+MOD(RND(),177)
130 U(I)=1
140 V(I)=1
150 IF MOD(RND(),2)=0 THEN U(I)=-U(I)
160 IF MOD(RND(),2)=0 THEN V(I)=-V(I)
170 NEXT
180 IF PEEK(53248)=0 THEN GOTO 180
190 FOR I=0 TO 7
200 POKE 53376+I*4+0,X(I)
210 POKE 53376+I*4+1,Y(I)
220 X(I)=X(I)+U(I)
230 Y(I)=Y(I)+V(I)
240 IF X(I)=12 THEN U(I)=-U(I)
250 IF X(I)=160 THEN U(I)=-U(I)
260 IF Y(I)=21 THEN V(I)=-V(I)
270 IF Y(I)=200 THEN V(I)=-V(I)
280 NEXT
290 IF AND(PEEK(16),1)=0 THEN GOTO 320
300 IF PEEK(53248)=1 THEN GOTO 300
310 GOTO 180
320 FOR I=0 TO 7
330 POKE 53376+I*4+0,0
340 POKE 53376+I*4+1,0
350 NEXT
360 DATA 0,20,0,1,85,64,5,85
370 DATA 80,5,85,80,21,125,84,21
380 DATA 215,84,21,213,84,21,213,84
390 DATA 21,215,84,5,125,80,5,85
400 DATA 80,5,85,80,13,85,112,12
410 DATA 93,48,12,93,48,3,28,192
```

```
420 DATA 3,12,192,3,12,192,0,142
430 DATA 0,0,170,0,0,170,0,131
```

A program that bounces multiple balloons around the screen. The program is similar to the previous example except that all eight sprites in the first sprite bank are in use. Program flow is largely the same, though loops are added to iterate over each sprite, its coordinates, and its velocity. Pressing Q will exit the program.

All eight sprites in use with the same balloon image but different color values.

Here we only used 8 sprites that can move around the entire screen. So far we've only been using the first sprite bank that begins at **$D080** and continues for 32 bytes (4 bytes for each of 8 sprites). Up to 32 sprites can be displayed using sprite banks and row effects, something covered when we discuss row effects in more detail.

In those situations, multiple sprite banks with their own information are swapped in and out by the Video

Interface Device as it draws the frame. The top half of the sprite register at **$D006** is used to select one of the sprite banks, and this value can be overridden at the start of each subsequent character row by a row effects setting. However, there can still only be a maximum of 8 sprites on any row.

DISABLING VIDEO OUTPUT

The VID also allows you to turn off the video display entirely, for example if you don't want the user to see the screen slowly being drawn in Cody BASIC. One workaround would be to relocate the screen and color memory to another location, but a quicker way is to just shut off the video temporarily.

This can be done using the control register at **$D001** or decimal 53249. When bit 0 is set to 1, the display output is turned off and replaced with the current screen border color. When the bit is cleared back to a 0, screen output returns as expected.

```
10 POKE 53249,1
20 T=TI
30 IF TI-T<300 THEN GOTO 30
40 POKE 53249,0
```

A simple example that turns off the video output for 5 seconds.

Because the VID is implemented inside the Propeller and uses its internal memory, disabling video output doesn't speed up the 65C02. Many older computers turned off video generation to speed up computations as the video hardware no longer shared

the bus, but in the Cody Computer, our system just doesn't work like that.

ROW EFFECTS

One last feature of the Video Interface Device is its ability to switch out graphics while the screen is being drawn. Many 8-bit computers of the past had raster interrupts that notified the processor when a particular line was drawn on the screen, and if the computer could respond fast enough, it could actually swap out some of the data. The Cody Computer has a built-in way of doing this.

The Cody Computer supports a system of row effects, where the VID can be programmed to replace the contents of certain registers on specified character rows. The base register, scroll register, screen colors register, and sprite register can all be overridden at any character row boundary using this mechanism. Once applied, the change remains for the rest of the current frame or until another value is specified. On the next frame the process begins anew with the original register values.

Using the row effects unlocks the full capacity of the Cody Computer's graphics system. You can have multiple banks of sprites on the screen at the same time, so long as they are partitioned into different rows on the screen. You can change the shared screen and sprite colors to have a more colorful output and avoid color attribute clashes. You can have split scrolling so that a game screen can be scrolled while status bars remain fixed in place. You can dynamically

swap out character sets and create a very detailed, dynamic screen without resorting to bitmap mode.

ROW EFFECTS REGISTER BANKS

The mechanism works by having two dedicated row effects register banks of 32 bytes each. The first bank, starting at **$D040** or decimal 53312, contains the control values for each row effect. These values tell the VID where to perform the replacement and what register to replace. The second bank, starting at **$D060** or decimal 53344, specify the replacement values that should be used.

The control bytes consist of several pieces of information packed into a single byte. Bits 0 through 5 contain the row number to begin the replacement on. Bits 6 and 7 contain a two-bit value specifying the target register to override. The last bit, bit 8, is an enable bit that must be set to 1 for that specific row effect to be applied. The two-bit destination code is as follows:

- Destination **00** replaces the base register.
- Destination **01** replaces the scroll register.
- Destination **10** replaces the screen colors register.
- Destination **11** replaces the sprite register.

Row effects must also be enabled globally in the control register at **$D001** or decimal 53249. Bit 3 of the control register must be set to 1 to enable the effects regardless of the enable bit on each control byte in the row effect bank.

SCREEN COLORS AND ROW EFFECTS

One of the typical uses for row effects is increasing the number of colors on the screen. As you may recall, each location on the screen has two unique colors and two shared colors. With row effects, the shared colors can be swapped for other colors starting at any character row boundary.

Programs can use this ability to divide the screen into different shared color regions for different reasons. Games might use this to have different shared colors in different areas, for example, different shared colors for sky, ground, and ocean. Paint programs could use this to permit more colors on the screen for artwork. And for more detailed graphics, the same principle applies, allowing more colors to be used in detailed images or backgrounds than would normally be possible.

To do this, we need to select the screen colors register as our destination using code **10**, then ensure that the replacement value is loaded into the corresponding row effect data register. The format of the data in the row effect data register is the same as it would be if directly stored to the screen colors register.

```
10 FOR I=0 TO 7
20 READ M
30 POKE 51200+255*8+I,M
40 NEXT
50 FOR I=0 TO 999
60 POKE 50176+I,255
70 POKE 55296+I,MOD(RND(),16)*16+MOD(RND(),16)
80 NEXT
90 FOR I=0 TO 24
100 POKE 53312+I,OR(192,I)
110 POKE 53344+I,MOD(I,16)*16+MOD(I+8,16)
120 NEXT
130 POKE 53249,OR(PEEK(53249),8)
140 DATA 80,80,80,80,250,250,250,250
```

A modified version of the sample program for defining custom characters. As in that example a character pattern using four different colors is programmed in and filled to the entire screen. Unlike the earlier example, the two common colors on the characters will be different for each row. This is because we told the Cody Computer to change the shared screen colors on each row using row effects.

SPRITE COLORS AND ROW EFFECTS

While not as broadly useful, the shared sprite color can also be changed on a per-row basis using the sprite register row effect. The sprite register contains both the sprite bank base (in the high four bits) and the sprite shared color (in the low four bits).

By using **11** as our destination code to replace the sprite register, we can target the sprite register for a row effect. To change only the sprite color, our replacement value in the corresponding data register

would have the sprite bank register held constant but use a different color code in the low four bits.

```
10 A=41984
20 B=(A-40960)/64
30 FOR I=0 TO 63
40 READ M
50 POKE A+I,M
60 NEXT
70 POKE 53254,0
80 POKE 53376+2,9*16+14
90 POKE 53376+3,B
100 P(0)=12
110 P(1)=21
120 D(0)=1
130 D(1)=1
140 FOR I=0 TO 24
150 POKE 53312+I,OR(224,I)
160 POKE 53344+I,MOD(I,16)
170 NEXT
180 POKE 53249,OR(PEEK(53249),8)
190 IF PEEK(53248)=0 THEN GOTO 190
200 POKE 53376+0,P(0)
210 POKE 53376+1,P(1)
220 P(0)=P(0)+D(0)
230 P(1)=P(1)+D(1)
240 IF P(0)=12 THEN D(0)=-D(0)
250 IF P(0)=160 THEN D(0)=-D(0)
260 IF P(1)=21 THEN D(1)=-D(1)
270 IF P(1)=200 THEN D(1)=-D(1)
280 IF AND(PEEK(16),1)=0 THEN GOTO 310
290 IF PEEK(53248)=1 THEN GOTO 290
300 GOTO 190
310 POKE 53376+0,0
320 POKE 53376+1,0
330 DATA 0,20,0,1,85,64,5,85
340 DATA 80,5,85,80,21,125,84,21
350 DATA 215,84,21,213,84,21,213,84
360 DATA 21,215,84,5,125,80,5,85
370 DATA 80,5,85,80,13,85,112,12
380 DATA 93,48,12,93,48,3,28,192
390 DATA 3,12,192,3,12,192,0,142
```

```
400 DATA 0,0,170,0,0,170,0,131
```

A modified version of the balloon sprite example. In this program we have also added row effects to change the common sprite color on each row. As the balloon travels the screen the shared color will pulsate and change depending on the rows the balloon sprite hovers over. Press Q to quit.

SPRITE BANKS AND ROW EFFECTS

As you may have guessed during the above section on sprite color row effects, the sprite banks can also be changed when the sprite register is used in a row effect. Different sprite banks can contain different sprites and the row effects can change the bank at different rows on the screen. This approach is quite powerful as it allows more than eight sprites to be on the screen at the same time. The only limitation is that only one sprite bank can be used on any single row.

This technique is very useful in games so long as your game logic is designed to support it. An arcade game could have up to 8 airplanes in a sky region, up to 8 tanks on a ground region, and up to 8 ships in a water region, all on the same screen. A similar approach could be used for flying versus ground enemies in a sidescroller. A player sprite that needs to transit multiple regions can be programmed into multiple banks with the same information, so that regardless of its current location it's drawn appropriately on the screen.

```
10 A=41984
20 B=(A-40960)/64
```

```
30 FOR I=0 TO 63
40 READ M
50 POKE A+I,M
60 NEXT
70 FOR I=0 TO 31
80 POKE 53376+I*4+0,13+18*MOD(I,8)
90 POKE 53376+I*4+1,25+(I/8)*48
100 POKE 53376+I*4+2,9*16+MOD(I,16)
110 POKE 53376+I*4+3,B
120 NEXT
130 FOR I=0 TO 31
140 POKE 53312+I,0
150 NEXT
160 FOR I=0 TO 3
170 POKE 53312+I,OR(224,I*6)
180 POKE 53344+I,I*16
190 NEXT
200 POKE 53249,OR(PEEK(53249),8)
210 IF PEEK(53248)=0 THEN GOTO 210
220 FOR I=0 TO 31
230 T=PEEK(53376+I*4)+1
240 IF T>174 THEN T=0
250 POKE 53376+I*4,T
260 NEXT
270 IF AND(PEEK(16),1)=0 THEN GOTO 300
280 IF PEEK(53248)=1 THEN GOTO 280
290 GOTO 210
300 FOR I=0 TO 31
310 POKE 53376+I*4+0,0
320 POKE 53376+I*4+1,0
330 NEXT
340 DATA 0,20,0,1,85,64,5,85
350 DATA 80,5,85,80,21,125,84,21
360 DATA 215,84,21,213,84,21,213,84
370 DATA 21,215,84,5,125,80,5,85
380 DATA 80,5,85,80,13,85,112,12
390 DATA 93,48,12,93,48,3,28,192
400 DATA 3,12,192,3,12,192,0,142
410 DATA 0,0,170,0,0,170,0,131
```

A sprite example with multiple sprite banks in use. Based on the multiple sprite example earlier in the chapter, this program sets up a total of 32 sprites in four sprite banks. The sprites are split into four horizontal regions and the first four row effects

registers set up to switch out sprite banks at those screen-split locations. Pressing Q will quit.

A total of 32 sprites on the screen thanks to row effects. Note how each group of eight sprites exists in its own horizontal region on the screen.

SCROLLING WITH ROW EFFECTS

Row effects can also be used to set different fine-scroll amounts on different parts of the screen. The contents of the scroll register can be overridden using destination code **01** and the new value of the scroll register in the corresponding row effect data register. Horizontal or vertical scrolling must be enabled in the control register separately.

This approach can be useful for games that require a split-screen effect. Many games include a static status area with health/life, timer, inventory, or other information while the main game area scrolls along.

Splitting the screen into multiple scroll areas can help with this, and the split can even be combined with the double-buffering approach mentioned in the earlier section on fine and coarse scrolling.

```
10 FOR I=0 TO 999
20 POKE 50176+I,65
30 NEXT
40 FOR I=0 TO 31
50 POKE 53312+I,0
60 NEXT
70 POKE 53312+0,OR(160,0)
80 POKE 53344+0,0
90 POKE 53312+1,OR(160,3)
100 POKE 53249,12
110 S=0
120 IF PEEK(53248)=0 THEN GOTO 120
130 POKE 53344+1,S*16
140 S=MOD(S+1,4)
150 IF AND(PEEK(16),1)=0 THEN GOTO 180
160 IF PEEK(53248)=1 THEN GOTO 160
170 GOTO 120
180 POKE 53249,0
```

An example of split-screen scrolling. The row effects registers are cleared and then set up to have two different horizontal scrolling values, zero for the first three rows and a changing amount for the remainder of the screen. Horizontal scrolling and row effect are switched on and the main loop updates the scroll amount. Pressing the Q key ends the program and shuts off the extra effects.

RELOCATIONS USING ROW EFFECTS

The base register can be updated when the destination code is set to **00**. This can be used to update the base of screen memory on the fly, but in

general is going to be used to change the character set base portion of the register instead. Doing this allows more than 256 characters to be used on the screen at the same time.

The format used for the row effect's data register is the same as that used for the register itself. For example, to change the character set, keep the same screen memory base but use a different character set base.

This can be useful in games. For example, imagine a full character set used as tiles for the game world, and a separate character set used for the text and user interface at the top and bottom of the screen.

```
10 FOR I=0 TO 999
20 POKE 50176+I,65
30 NEXT
40 A=40960
50 B=(A-40960)/2048
60 FOR I=0 TO 2047
70 POKE A+I,MOD(I,2)*85
80 NEXT
90 FOR I=0 TO 31
100 POKE 53312+I,0
110 NEXT
120 POKE 53312,OR(128,12)
130 POKE 53344,9*16+B
140 POKE 53249,8
150 IF AND(PEEK(16),1)=1 THEN GOTO 150
160 POKE 53249,0
```

Using row effects to change the base address of the character set in the middle of a frame. A test pattern from a previous example is programmed into a second character set, then switched out in the middle of the frame using a row effect. The Q key will quit the program.

BITMAPPED GRAPHICS

The Cody Computer also supports a limited form of bitmap graphics. In this mode, each byte in screen memory is expanded to eight bytes containing the bit pattern to draw at the location. The layout of each eight-byte section is exactly the same as in character memory, and the same color limitations apply as in the normal character graphics mode. This also expands the size of video memory from 1000 bytes to 8000 bytes. Bitmap mode is enabled by bit 4 of the video control register at **$D001** or decimal 53249.

In many respects the bitmap mode is more of a hybrid mode between character graphics and a fully-bitmapped screen. The first eight bytes represent the first 4x8 tile, the next eight bytes represent the second 4x8 tile, and so on for the remainder of the screen. While this makes the implementation easier within the Cody Computer's firmware (and also more faithful to how things actually worked on the Commodore computers), it does make plotting pixels more difficult.

To find where to plot a pixel, it's necessary to begin with the (x,y) coordinate on the screen's 160x200 grid. First divide the y-coordinate by 8 (the number of lines in a character) rounding down, then multiply by 320 (the number of bytes in a row of 40 tiles). Then divide the x-coordinate by 4 (the number of columns in a character) rounding down, then multiply by 8 (the number of bytes in a character). This gets us to the beginning of the bytes for that section of the screen. We add the remainder from the earlier division of the y-coordinate to get the final byte we need to update.

To select the actual pixel within that byte, however, we still have a bit of work to do. We need to mask out the portion of the byte we want to change and replace it with the color we want to draw. Just like in character memory, each byte is represented by two bits, with the highest two bits representing the leftmost dot in the line of pixels. This means that we'll need a two-bit mask that we shift right the appropriate number of two-bit increments, and we'll need to do the same with the color value we'll insert.

It's not an easy operation, though once you've walked through the steps, it'll become clearer. It also means that it's a lot more time-consuming than just updating a single byte to change an entire tile on the screen. Bitmapped graphics have their place, but for things like video games, many of the most action-intense ones will need to rely on the character graphics mode over the bitmapped mode: A slow retro-style system like the Cody Computer just isn't going to push that many pixels.

Below we have a Cody BASIC program that demonstrates the bitmap mode by setting it up and randomly plotting some pixels. We have to relocate our screen memory so that we have enough space for the bigger memory, clear out the memory, set up our colors, and finally enter a loop where we randomly plot pixels into the screen area. The complicated calculation discussed above is implemented as a subroutine in Cody BASIC to make it a little easier to follow.

```
10 FOR I=40960 TO 48960
20 POKE I,255
```

```
30 NEXT
40 FOR I=55296 TO 56296
50 POKE I,RND()
60 NEXT
70 POKE 53253,1
80 POKE 53250,224
90 POKE 53251,5
100 POKE 53249,OR(PEEK(53249),16)
110 X=MOD(RND(),160)
120 Y=MOD(RND(),200)
130 C=MOD(RND(),4)
140 GOSUB 300
150 IF AND(PEEK(16),1)=0 THEN GOTO 200
160 GOTO 110
200 POKE 53253,22
210 POKE 53250,231
220 POKE 53251,149
230 POKE 53249,AND(PEEK(53249),15)
240 END
300 P=40960
310 P=P+Y/8*(40*8)
320 P=P+X/4*8
330 P=P+MOD(Y,8)
340 M=192
350 C=C*64
360 R=MOD(X,4)
370 IF R=0 THEN GOTO 420
380 M=M/4
390 C=C/4
400 R=R-1
410 GOTO 370
420 POKE P,OR(AND(PEEK(P),XOR(M,255)),C)
430 RETURN
```

Plotting random pixels in bitmap mode. It will take a little while to run as it clears out screen memory before beginning to plot pixels. When ready to exit, press the Q key and the screen will be restored to character graphics mode.

HIGH RESOLUTION GRAPHICS

High-resolution character and bitmap graphics modes are also available. These allow programs to increase the screen resolution to 320 pixels by 200 pixels at the cost of disabling many of the Cody Computer's more advanced video features. The horizontal resolution is doubled but only two colors may appear in any 8-by-8 pixel region. Features such as sprites and scrolling are disabled. Row effects are still possible but many of them have no effect because of the lack of sprites and other features. To enable the high-resolution mode you must turn on bit 5 of the video control register at **$D001** or decimal 53249.

Pixel data is similar to the low-resolution multicolor mode except that there are only two choices for colors. As a result, each pixel is represented by a single bit. A single byte of pixel data represents eight pixels, each of which can have only one of two colors. A 0 bit selects the low nibble in color memory for a particular location while a 1 bit selects the high nibble in color memory.

Character graphics in high resolution mode are identical to those in the normal graphics modes. The only difference is in the format of the data in character memory, which uses the one-bit-per-pixel layout described above. Otherwise it behaves exactly the same way as for the four-color mode you already learned about.

A simple example below will load some abstract high-resolution designs into unused spaces in character memory, then show them on the screen in a

high-resolution mode. You will also notice how the default characters look different (but are often nonetheless readable) even when switched into high-resolution mode. This occurs because the two-bit multicolor patterns are being interpreted as single-bit on-or-off values instead.

```
10 FOR I=0 TO 31
20 READ M
30 POKE 51200+252*8+I,M
40 NEXT
50 POKE 53249,OR(PEEK(53249),32)
60 FOR I=0 TO 999
70 POKE 50176+I,252+MOD(RND(),4)
80 POKE 55296+I,MOD(RND(),16)*16+MOD(RND(),16)
90 NEXT
100 PRINT "PRESS ENTER TO EXIT";
120 INPUT X$
130 POKE 53249,AND(PEEK(53249),15)
140 DATA 1,3,7,15,31,63,127,255
150 DATA 0,128,192,224,240,248,252,254,255
160 DATA 255,127,63,31,15,7,3,1
170 DATA 255,254,252,248,240,224,192,128,0
```

An example of the high-resolution character mode. Four characters at the top of character memory are reprogrammed as high-resolution characters. High-resolution mode is enabled and the color and screen memory updated with random colors and values. At the end you press enter to exit from the high-resolution mode.

A high-resolution bitmap mode is also possible. While most of the additional graphics features are not available in high-resolution mode, bit 4 of the video control register at **$D001** or decimal 53249 can be used together with bit 5 to enable the high-resolution bitmap mode. As with the character mode, the overall

memory layout is the same. The only difference is the pixel data, which must conform to the same one-bit-per-pixel layout described for character pixel data.

Plotting a specific pixel on the screen also follows the same algorithm as described for the normal multicolor bitmap mode. Again divide the y-coordinate by 8 and multiply by 320. However, you must divide the x-coordinate by 8 instead of 4 because there are eight pixels across per character rather than four, then multiply that result by 8. As before, add the remainder from dividing the y-coordinate to get the actual byte. The bit mask operation also works in a similar fashion except that the mask is only one bit and is shifted in one-bit increments.

The following example is very similar to the bitmap example shown previously. However, it has been adjusted to support the high-resolution two-color mode instead of the lower-resolution multicolor mode. Note the changes to portions of the code related to the x-coordinate and bit-shifting in particular.

```
10 FOR I=40960 TO 48960
20 POKE I,0
30 NEXT
40 FOR I=55296 TO 56296
50 POKE I,RND()*16
60 NEXT
70 POKE 53250,224
80 POKE 53251,5
90 POKE 53249,OR(PEEK(53249),48)
100 X=MOD(RND()+RND()*2,320)
110 Y=MOD(RND(),200)
120 C=MOD(RND(),2)
130 GOSUB 300
140 IF AND(PEEK(16),1)=0 THEN GOTO 200
150 GOTO 100
200 POKE 53250,231
```

```
210 POKE 53251,149
220 POKE 53249,AND(PEEK(53249),15)
230 END
300 P=40960
310 P=P+Y/8*(40*8)
320 P=P+X/8*8
330 P=P+MOD(Y,8)
340 M=128
350 C=C*128
360 R=MOD(X,8)
370 IF R=0 THEN GOTO 420
380 M=M/2
390 C=C/2
400 R=R-1
410 GOTO 370
420 POKE P,OR(AND(PEEK(P),XOR(M,255)),C)
430 RETURN
```

A modified bitmap example for the two-color high-resolution mode. As with the earlier bitmap example, pressing the Q key returns to the default graphics mode.

8

Sound and Music Programming

INTRODUCTION

The Cody Computer supports sound and music through the Sound Interface Device or "SID," a copy of the famous SID from the Commodore 64. The Cody Computer's SID supports many, but not all, of the same features as its predecessor. It's intended as a simplified sound generator suitable for the curious hobbyist or casual user, but with a significant degree of compatibility. Like the Cody Video Interface Device, the Cody SID is implemented as a software peripheral in the Propeller.

Like the original SID, the Cody SID relies on principles of digital audio synthesis to generate sounds. Unlike modern computers which essentially play back raw audio data (often after processing the signal in some way), the SID generates sound mathematically. Counters and mathematical formulas are used to produce sound-like waves and combine them together, with the exact characteristics of these waves under the control of the programmer.

The Cody SID supports up to three voices, or independent sounds, at the same time. Each voice can generate a sound at a different frequency, and each sound can consist of either a triangle wave, a sawtooth wave, a pulse wave, and white noise. These are combined with another wave called an envelope, which determines how loud the sound gets, how quickly, and how slowly it fades away when turned off.

The envelope is defined using attack (how fast the sound reaches a peak volume), decay (how long the sound drops to its normal value after the peak),

sustain (how loud the sound stays), and release (how long the sound takes to fade out). This ADSR envelope shapes the underlying sound for each voice and is capable of mimicking many instruments and sound effects.

The original SID chips in the Commodore 64 family had other features, including filters that let the programmer emphasize certain high-frequency, low-frequency, or middle portions of each sound. Filters could vary greatly between SID chips, and in order to keep the Cody Computer a fun learning tool, filters aren't supported by the Cody Computer's SID. Some sounds and songs, even if ported to work on the Cody Computer, won't sound quite right as a result, but most results are at least passable. Also unlike the Commodore SID, the Cody SID doesn't permit the user to select multiple waveforms for the same voice: you have to pick one, and only one, type of sound for each voice at any one time.

MAKING A SOUND

To program sounds, you poke values into memory registers. Each voice has seven registers, and there are a total of three voices, starting at memory location **$D400** (decimal 54272). Global settings for the SID, including volume, are controlled by a handful of other registers immediately following the voice registers.

For each voice, the registers are organized in the same order. The first two registers contain the low and high bytes for the voice's sound frequency as a number from 0 to 65535 (these map, more or less, to a range between 0 and 4 kilohertz as audio

frequencies). Following those are two registers only used for pulse waveforms, containing the low and high bytes of the pulse wave's duty cycle (how long it is on relative to how long it is off). The pulse value can range from 0 to 4095, with a zero being off all the time and 4095 being on all the time. (If you're curious, the more limited range of the pulse width occurs because the top half of the pulse wave's high byte is unused, just as it was on the C64.)

After that, the fifth register, the control register, allows you to select the type of sound you want to produce. The high four bits contain the type of sound while the lower four bits contain other control information, including turning the voice on and off. Bit 4 selects a triangle wave, bit 5 selects a sawtooth wave, bit 6 selects a pulse wave, and bit 7 selects a white noise wave. The lowest bit, bit 0, is the gate bit that turns the voice itself on and off. (The other bits are used for more advanced features that we'll cover later.)

The sixth and seventh registers define the ADSR (attack-decay-sustain-release) envelope that was mentioned in the introduction. The attack and decay are set by the sixth register. The attack value (how long the sound takes to reach maximum volume ater it starts) is stored in the top half of the sixth register. The decay value (how long the sound takes to decrease from its maximum to its sustain level) is stored in the bottom half. Both range from 0 to 15 but cover different time ranges. The attack range covers between 0 and 8 seconds while the decay range covers between 0 and 24 seconds. The relationship is not linear, so you need to consult the table below to find the exact value.

The seventh and final voice register contains the other part of the ADSR envelope, the sustain and release values. The sustain value (the volume the voice stays at after the decay phase) is stored in the top half of the register. The release value (the time it takes the sound to fade out after it's turned off) is stored in the bottom half. The sustain value ranges from 0 to 15 and represents a volume level. The release value also ranges from 0 to 15 but represents a time value, with its possibilities being the same as those for the decay value.

Value (dec)	Value (hex)	Attack (ms)	Decay/Release (ms)
0	$0	2	6
1	$1	4	24
2	$2	16	48
3	$3	24	72
4	$4	38	114
5	$5	58	168
6	$6	68	204
7	$7	80	240
8	$8	100	300
9	$9	250	750
10	$A	500	1500
11	$B	800	2400
12	$C	1000	3000
13	$D	3000	9000
14	$E	5000	15000
15	$F	8000	24000

The attack, decay, and release values and their rates. Note that sustain values are not included in the table

because the sustain setting is a volume, not a time constant.

In many respects, sound programming can be more difficult than video programming. While video programming has many complicating factors to get a picture on the screen, the overall concepts of pixels, characters, and sprites are usually somewhat familiar. Sound programming, absent any personal experience with musical instruments or signal processing, can take longer to understand.

For that reason, we'll start with a simple example. The following BASIC program will generate a triangle wave at 440 hertz, which is common in music as the A note above middle C. This particular frequency is used as a standard to tune instruments, and we'll use it here to get started.

```
10 FOR I=0 TO 29
20 POKE 54272+I,0
30 NEXT
40 POKE 54296,15
50 POKE 54272,AND(7382,255)
60 POKE 54273,7382/256
70 POKE 54277,2*16+4
80 POKE 54278,14*16+6
90 POKE 54276,16+1
100 T=TI
110 IF TI-T<120 THEN GOTO 110
120 POKE 54276,16
```

A program that plays an A note on voice 1. The SID registers are reset to 0, then the values for a note on voice 1 are poked into memory. A brief delay occurs before the sound is turned off.

The program begins by clearing out all the SID registers. This is very important in any case, as you may have noticed earlier in the book when running one program messes up the environment for a later one. For the SID it's particularly important so that any existing sounds or settings get cleared out.

After the SID is cleared out, the program sets the volume to maximum. The volume is poked into the lower half of the main volume control register at **$D418** or decimal 54296. The 440 Hz frequency is converted to its corresponding SID value, 7382, and then poked into the frequency registers at **$D400** (decimal 54272) and **$D401** (decimal 54273). (To calculate the frequency value to poke in, an old formula for the Commodore SID can be used, dividing the desired frequency by 0.0596. If you forget that, a reasonable approximation can be made by recalling that the range of frequencies goes from 0 to about 4000, and the register value goes from 0 to 65535; you won't get the exact value, but you can solve it like any other proportion.)

The attack and decay values are poked into register **$D405** or decimal 54277. Relatively small values are used for this example, with an attack value of 2 corresponding to a mere 16 milliseconds. The decay value of 4 isn't much bigger, corresponding to about 114 milliseconds. Sustain and release values are then poked into the following register at **$D406** or decimal 54278. A relatively high sustain volume of 14 is poked along with a relatively short decay value of 6 (corresponding to around 204 milliseconds).

To start the sound, the program pokes the voice 1 control register at **$D404**. Bit 4 is set to enable the

triangle wave sound, while bit 0 is also set to begin the sound. A timer loop waits for about two seconds, and then the control register is poked with bit 0 turned off to end the sound.

CREATING SOUNDS WITH NUMBERS

This may be the first time you're hearing of triangle waves, sawtooth waves, pulse waves, so we'll go over a brief example of each one. The exact values, including the frequencies and ADSR values, aren't the main focus here. The intent is to give you an idea of how the different sounds actually sound.

TRIANGLE WAVES

A triangle wave is basically what it sounds like. The wave goes up to a maximum in a straight line, peaks, goes down to a minimum in a straight line, and then repeats. Triangle waves are enabled by setting bit 4 in a voice's control register.

The triangle wave is also special in that it's the closest the SID can produce to an actual sine wave. Because of its audio characteristics, it can be described as sounding like something between a square wave (or pulse) and a sine wave.

```
10 FOR I=0 TO 29
20 POKE 54272+I,0
30 NEXT
40 POKE 54296,15
50 POKE 54272,196
60 POKE 54273,22
70 POKE 54277,105
80 POKE 54278,252
90 POKE 54276,17
```

A Cody BASIC program that produces a triangle wave. The exact SID register values were taken from an emudev.de article on the Commodore 64's sounds.

SAWTOOTH WAVES

A sawtooth wave is kind of like a triangle wave with special characteristics. Instead of going up and down in a linear fashion, it goes up to a maximum, then immediately drops to its minimum. This produces a waveform that looks a lot like the teeth on a saw blade. Sawtooth waves are enabled with bit 5 in a voice's control register.

Sawtooth waves tend to sound very harsh and sharp. They can be made to sound similar to a buzzer in many situations. Yet when set up with the appropriate characteristics, they can also be very useful for other sound effects and even music.

```
10 FOR I=0 TO 29
20 POKE 54272+I,0
30 NEXT
40 POKE 54296,15
50 POKE 54272,195
60 POKE 54273,10
70 POKE 54277,105
80 POKE 54278,252
90 POKE 54276,33
```

An example of a sawtooth wave. The exact SID register values were taken from an emudev.de article on the Commodore 64's sounds.

PULSE WAVES

A pulse wave may be what most people think of as an electronically-generated sound. It goes immediately to its maximum, stays there for a particular time, and then drops to its minimum, staying there for a while until the process repeats. A pulse wave has a duty cycle that indicates how long the wave is on compared to how long it is off: For example, a wave with a duty cycle of 75% is at its maximum three times longer than its minimum. A square wave is just a special case of the pulse wave with a duty cycle of 50%. Pulse waves are enabled using bit 6 in a voice's control register.

In addition to being useful to generate very electronic beeps and blips, different duty cycles for each wave can produce a variety of unique sounds. On the SID the pulse wave is unique in that in addition to the frequency value, the pulse is also programmable using some of the voice's registers.

```
10 FOR I=0 TO 29
20 POKE 54272+I,0
30 NEXT
40 POKE 54296,15
50 POKE 54272,196
60 POKE 54273,9
70 POKE 54274,15
80 POKE 54275,15
90 POKE 54277,105
100 POKE 54278,252
110 POKE 54276,65
```

An example of a pulse wave. The exact SID register values were taken from an emudev.de article on the Commodore 64's sounds.

NOISE

Noise is similar to the white noise that you may have heard from a white noise sound machine. Different techniques can be used to generate noise, but one of the most common is to use what is called a linear feedback shift register. It's similar to a normal shift register, but it has taps at different places along the shift register's path to obtain output or feed back into the circuit. Noise output is enabled using bit 7 of a voice's control register.

Noise is useful for a variety of sound effects, but it can also be used in various musical sounds. Nor should noise be considered as something to be used for static in sound effects. Consider that a white noise sound with the appropriate frequency, fade-in, and fade-out, could be used to mimic the sound of the ocean.

```
10 FOR I=0 TO 29
20 POKE 54272+I,0
30 NEXT
40 POKE 54296,15
50 POKE 54272,196
60 POKE 54273,9
70 POKE 54277,105
80 POKE 54278,252
90 POKE 54276,129
```

An example of noise output. The exact SID register values were taken from an emudev.de article on the Commodore 64's sounds.

EXPERIMENTING WITH DIFFERENT VALUES

Now that you've heard how the Cody Computer can generate sounds, try the following program to see what other kinds of sounds can be produced. Instead of writing many different programs with different settings, you can use the one below to enter different values and hear the results immediately. This won't work as an exhaustive example of every sound the Cody Computer can make using its SID, but it gives you a place to begin.

```
10 FOR I=0 TO 29
20 POKE 54272+I,0
30 NEXT
40 PRINT "AVAILABLE SOUNDS:"
50 PRINT "1. TRIANGLE"
60 PRINT "2. SAWTOOTH"
70 PRINT "4. PULSE"
80 PRINT "8. NOISE"
90 PRINT "SOUND (1, 2, 4, OR 8)";
100 INPUT C
110 PRINT "FREQUENCY (0-65535)";
```

```
120 INPUT F
130 W=0
140 IF C<>4 THEN GOTO 170
150 PRINT "PULSE WIDTH (0-4095)";
160 INPUT W
170 PRINT "ATTACK RATE (0-15)";
180 INPUT A
190 PRINT "DECAY RATE (0-15)";
200 INPUT D
210 PRINT "SUSTAIN LEVEL (0-15)";
220 INPUT S
230 PRINT "RELEASE RATE (0-15)";
240 INPUT R
250 PRINT "OVERALL VOLUME (0-15)";
260 INPUT V
270 POKE 54296,V
280 POKE 54272,AND(F,255)
290 POKE 54273,F/256
300 POKE 54274,AND(W,255)
310 POKE 54275,W/256
320 POKE 54277,A*16+D
330 POKE 54278,S*16+R
340 PRINT "PRESS ENTER TO PLAY";
350 INPUT X$
360 POKE 54276,C*16+1
370 PRINT "PRESS ENTER TO STOP";
380 INPUT X$
390 POKE 54276,C*16
400 PRINT "AGAIN (Y/N)";
410 INPUT X$
420 IF X$="N" THEN END
430 PRINT ""
440 GOTO 10
```

A tool for experimenting with simple SID sounds. On each loop the user is prompted for some SID values, and the program plugs them into the SID registers for voice 1.

To use the program you just need to load and run it. You specify the type of sound you want to generate by entering a number corresponding to the voice settings in the top half of the control register. After that you

enter the raw values for the frequency, attack, decay, sustain, and release, along with the overall volume. If you're trying out a pulse wave you'll also be prompted for the pulse's duty cycle. The program doesn't do any error checking, so if you enter an invalid value, you'll get some strange results.

You should experiment with different values to see how they sound, but below are some examples from a 1984 edition of the Commodore 64 User's Manual. One table contains the suggested values to resemble the sounds of different musical instruments. Another table shows a subset of the musical scale, giving you one octave's worth of constants to try out different notes.

Instrument	Sound	Pulse	Attack	Decay	Sustain	Release
Piano	4	225	0	9	0	0
Flute	1	0	0	6	0	0
Harpsichord	2	0	0	9	0	0
Xylophone	1	0	0	0	15	0
Accordion	1	0	6	6	0	0
Trumpet	2	0	6	0	0	0
Noise	4	0	0	0	0	0

A table of settings copied from a 1984 edition of the Commodore 64 User's Manual. Each is intended to be a rough first approximation of a musical instrument.

The exact sound values you use are largely the result of experimentation, and the above table is only a beginning. As Commodore's own data sheet for the SID noted long ago, the exact characteristics of an instrument are vital when determining what values to plug in. A violin often builds up somewhat slowly

when bowed and reaches an intermediate volume before fading out. As a first guess, one might try a somewhat slow attack, a middle-range sustain volume, and a shorter decay and longer release time. A percussion instrument, on the other hand, generally reaches a peak volume suddenly, then goes away entirely. In the end, the correct values to plug in are those that sound best for the song or effect that one is trying to achieve.

Along with the ADSR settings, however, is the frequency. We discussed before that you can calculate the frequency value by dividing the frequency in hertz by 0.0596, and it helps to use this formula when you need to. Below is a brief table of notes and their corresponding frequency register values for the fourth octave, including the 440 hertz A note you played earlier.

Note	Frequency (Hz)	Value (dec)	Value (hex)
C4	261.63	4389	$1125
D4	293.66	4927	$133F
E4	329.63	5530	$159A
F4	349.23	5859	$16E7
G4	392.00	6577	$1981
A4	440.00	7382	$1CD6
B4	493.88	8286	$205E

A subset of the musical note frequency values from the Commodore SID 6581 data sheet. Values for the fourth octave (excluding sharps) are included as an example.

Don't limit yourself to trying to play musical sounds. The SID can be used for a variety of sound

effects as well. Also try to familiarize yourself with how the different settings work in practice. Listen for a faster or slower buildup as you adjust the attack rate, and note how the decay and sustain portions of the sound change as you alter their values. Try different release values to learn how a sound can quickly or slowly fade off.

PLAYING A SIMPLE SONG

The same approach can be used to play simple songs in Cody BASIC. To play an entire song, however, the musical notes and their lengths need to be taken into account. A musical note is just a frequency, so the corresponding frequency register value can be used to represent each note at a low level. The time for each note can be represented as a time constant of some sort.

To play a note, a program would load the instrument data from the above table, load the frequency value for the note to play, and then start playing by setting the gate bit to 1. The program then waits for a time associated with the length of a note before turning the note off and moving on to the next one.

In music, a common standard for timing is 4/4 time, in which a whole note lasts for an entire portion of a song called a measure. The rest of the system is fractional, with a half-note lasting for half of a measure, a quarter note lasting for one-fourth of a measure, and so on. A corresponding symbol, the whole rest, indicates that no note should be played for the entire measure. These also have fractional

divisions such as the half-rest and quarter-rest. These concepts can easily be represented on a computer.

To see how this could work, we'll look at an introductory example from one edition of the Commodore 64 User's Manual as translated to the Cody Computer. In it, a simple program of POKEs, FOR/NEXT statements, and DATA statements is used to play a portion of the chorus from the American folk song "Tom Dooley."

```
10 S=54272
20 FOR Z=S TO S+24
30 POKE Z,0
40 NEXT
50 POKE S+24,15
60 POKE S+2,255
70 POKE S+3,0
80 POKE S+5,9
90 POKE S+6,0
100 READ H,L,D
110 PRINT H," ",L," ",D
120 IF H=0 THEN END
130 POKE S,L
140 POKE S+1,H
150 POKE S+4,65
160 FOR Z=1 TO D*4
170 NEXT
180 POKE S+4,64
190 FOR Z=1 TO 400
200 NEXT
210 GOTO 50
220 DATA 18,104,250,18,104,500,18,104,250
230 DATA 20,169,500,24,146,500,30,245,1000
240 DATA 30,245,1000,18,104,250,18,104,500
250 DATA 18,104,250,20,169,500,24,146,500
260 DATA 27,148,2000,18,104,250,18,104,500
270 DATA 18,104,250,20,169,500,24,146,500
280 DATA 27,148,1000,27,148,1000,27,148,250
290 DATA 27,148,500,30,245,250,24,146,500
```

```
300 DATA 20,169,500,24,146,1500,0,0,0
```

A modified program from the 1984 edition of the Commodore 64 User's Manual. It clears the SID registers and then plays a portion of the American folk song "Tom Dooley."

As in the earlier example, the SID registers are all reset to zero. The configuration data is then POKEd into voice 1 on the SID before the song is played. The song data is kept in DATA statements at the end of the program, with each set of three numbers representing a note: The first number is the high byte of the frequency value, the second number is the low byte of the frequency value, and the third number is the note's length. A value of 1000 represents a whole note, 500 represents half-note and 250 a quarter-note.

To play the song, the three pieces of data are read in a loop. Just as in the C64 example, an inner loop counts down for the length of the note. The note is then turned off and a brief delay occurs between notes for a folk-song feel. When a sequence of zero values is read at the end of the music data, the program stops.

There are, of course, many improvements that could be made to even a simple program such as this. Storing the notes and their delays as values for a loop worked well on the C64, but on the Cody Computer we have to make adjustments because the simpler Cody BASIC interpreter loops faster. The notes could instead be encoded using some other scheme, and the delays could be implemented by looking at the **TI** variable to determine elapsed time as in our graphics examples. However, the example serves its purpose, and it also

demonstrates the level of compatibility between the Cody SID and the real SID of the Commodore 64.

Keep in mind that this is a simple example that only uses one voice and doesn't show the best approach to playing music. On the Commodore 64, music was often written as self-contained programs called SID files, which were loaded into memory and called on a periodic basis to play a song.

Many of the simpler or earlier SIDs are playable on the Cody Computer, though there are also many incompatible ones because of differences in memory layouts and system features. Compute! magazine's SIDPLAYER, similar to a real MIDI-like computer music system, would likely be a better fit for the Cody Computer.

A simple SID player for PSID files, **CodySID**, is included as an assembly language example program later in the book. While not perfect, it does show how to load a SID file and play it in memory, and some recommended SID files that are known to work with it are mentioned. Writing a player for the MIDI-like SIDPLAYER system is left for the future or as an exercise for the reader.

SOUND EFFECTS

The SID can also be used for a variety of sound effects. In addition to the more obvious ones, it's also possible to update the values in the SID registers themselves to make even more interesting sounds. Many music players did exactly this, and games also

took advantage of the ability to control sound parameters on top of what the SID was already doing. (On the Cody SID, however, you'll want to be a bit more careful. If you change values in the Cody SID registers too quickly, the sound system may not pick up there was a change.)

The best way to come up with sound effects for your programs is to play around and come up with some yourself. There's no exact science to the process. Additionally, given that the C64 was at one point one of the most popular computers in the world, you'll find many resources on SID sounds that can be easily ported to the Cody Computer. A few examples are provided below to get you started.

AN EXPLOSION

The following program makes a quick explosion-like sound using the noise output from the Cody SID. The sound's attack and decay values are set to zero to produce an immediate effect, and the sustain level is set to a reasonably high value of 11. A release value of 10 ensures that the explosion sound takes a little while to fade away.

```
10 FOR I=0 TO 29
20 POKE 54272+I,0
30 NEXT
40 POKE 54296,14
50 POKE 54272,0
60 POKE 54273,20
70 POKE 54277,0
80 POKE 54278,186
90 POKE 54276,129
100 T=TI
110 POKE 54276,128
120 IF ABS(TI-T)<90 THEN GOTO 120
130 POKE 54276,0
```

A short Cody BASIC program that makes an explosion-like sound. Something like this could be used for a depth charge dropped on a submarine or a photon torpedo hit against a starship.

AN ALERT SIREN

This example produces a sound like an alert or siren. To get a sharp, Klaxon-like sound, a sawtooth wave is used as the basis for the sound generation. ADSR values suitable for a siren were also plugged in. Also, because sirens or alerts go from high to low and back again, the program contains a **FOR** loop that turns the sound on and off three times as it plays. Brief delays during each part of the sound guarantee that the user will hear both the attack and release stages.

```
10 FOR I=0 TO 29
20 POKE 54272+I,0
30 NEXT
40 POKE 54296,14
50 POKE 54272,0
60 POKE 54273,20
70 POKE 54277,176
80 POKE 54278,249
90 FOR I=1 TO 3
100 POKE 54276,33
110 T=TI
120 IF ABS(TI-T)<60 THEN GOTO 120
130 POKE 54276,32
140 T=TI
150 IF ABS(TI-T)<60 THEN GOTO 150
160 NEXT
```

This program produces an alert or siren-like sound. Something like this could call a ship's crew to general quarters, or perhaps set the mood aboard a distressed space station.

AN ENERGY BEAM

This program makes a sound suitable for use in games as an energy beam on a far-off spaceship defending the frontier, or perhaps a deranged robot trying to zap the player in a sidescrolling platformer. It uses a pulse wave for the sound but randomly changes the low byte of the frequency value while the sound is playing.

```
10 FOR I=0 TO 29
20 POKE 54272+I,0
30 NEXT
40 POKE 54296,14
50 POKE 54273,40
60 POKE 54275,8
70 POKE 54276,0
80 POKE 54277,0
90 POKE 54278,192
100 POKE 54276,65
110 T=TI
120 POKE 54272,RND()
130 IF ABS(TI-T)<60 THEN GOTO 120
140 POKE 54276,0
```

A short Cody BASIC program that makes a laser-beam or energy-beam sound effect.

A COMMODORE 64 EXAMPLE

Also remember that the Cody SID is essentially a simplified version of the SID chip used in the Commodore 64. Not everything will be completely compatible, but a lot of it will be, even if you have to make some minor changes to a program. To demonstrate that, let's take a look at the program below.

This program is a translation of another program from the Commodore 64, this one a sound effects program used to show off the C64 and SID's capabilities to new users. It will play one of six possible sounds in a loop, allowing you to select a new one when it's done. When done, break out of the program using the **Cody** and **Arrow** key combination.

```
10 PRINT "WHICH SOUND EFFECT:"
20 PRINT "1. WAILING"
30 PRINT "2. SHOOTING"
40 PRINT "3. SIREN"
50 PRINT "4. ROCKET"
60 PRINT "5. CRASH"
70 PRINT "6. MACHINE GUN"
80 INPUT X
90 S=54272
100 FOR I=S TO S+24
110 POKE I,0
120 NEXT
130 K=-1
140 T=TI
150 GOSUB 1000+X*100
160 POKE S+2,P(2)
170 POKE S+3,P(1)
180 POKE S+5,A(1)
190 POKE S+6,A(2)
200 POKE S+1,N(1)
210 POKE S,N(2)
220 IF Q=2 THEN Q=3
230 IF Q<>2 THEN GOTO 260
240 POKE S+1,64
250 POKE S,188
260 POKE S+4,W(1)
270 IF Q<>1 THEN GOTO 360
280 FOR I=1 TO 40
290 N(2)=200-I*5
300 POKE S,N(2)
310 NEXT
320 FOR I=1 TO 30
330 N(2)=150-I*5
340 POKE S,N(2)
350 NEXT
360 L=15
370 POKE S+24,L
380 IF L=V THEN GOTO 440
390 IF X=4 THEN GOTO 440
400 L=L+K
410 FOR I=1 TO D
420 NEXT
430 GOTO 370
440 POKE S+4,W(2)
```

```
450 IF ABS(TI-T)>300 THEN GOTO 10
460 IF Q<>3 THEN GOTO 200
470 Q=2
480 GOTO 230
1100 V=15
1105 N(1)=65
1110 N(2)=0
1115 W(1)=65
1120 W(2)=64
1125 P(1)=9
1130 P(2)=255
1135 A(1)=15
1140 A(2)=0
1145 D=1
1150 Q=1
1155 RETURN
1200 V=0
1205 N(1)=40
1210 N(2)=200
1215 W(1)=129
1220 W(2)=128
1225 P(1)=0
1230 P(2)=0
1235 A(1)=15
1240 A(2)=15
1245 D=1
1250 Q=0
1255 RETURN
1300 V=0
1305 N(1)=36
1310 N(2)=85
1315 W(1)=33
1320 W(2)=32
1325 P(1)=0
1330 P(2)=0
1335 A(1)=136
1340 A(2)=129
1345 D=350
1350 Q=2
1355 RETURN
1400 V=0
1405 N(1)=25
1410 N(2)=100
1415 W(1)=129
1420 W(2)=128
1425 P(1)=0
```

```
1430 P(2)=0
1435 A(1)=9
1440 A(2)=129
1445 D=50
1450 Q=0
1455 RETURN
1500 V=0
1505 N(1)=5
1510 N(2)=251
1515 W(1)=129
1520 W(2)=128
1525 P(1)=0
1530 P(2)=0
1535 A(1)=129
1540 A(2)=65
1545 D=50
1550 Q=0
1555 RETURN
1600 V=15
1605 N(1)=34
1610 N(2)=75
1615 W(1)=129
1620 W(2)=128
1625 P(1)=0
1630 P(2)=0
1635 A(1)=8
1640 A(2)=1
1645 D=50
1650 Q=0
1655 RETURN
```

The sound effects example from the Commodore 64 manual, updated to run on Cody Basic. While not the easiest program to follow, even in its original C64 version, it demonstrates the variety of sound effects possible even in simple BASIC programs.

The vast majority of the program consists of the values to plug in for different sounds. You can look at the initial register values by reading the appropriate lines in the program (a **GOSUB** branches to the setup code for a particular sound). A collection of **POKE**,

FOR, and **IF** statements take the values and use them to generate the selected sound.

The code for playing a sound is actually quite complicated, mostly because like the original program it uses the same code for playing all six sounds. Some values are changed on different loops, which adds to the complexity. For a particular sound in the example, it's best to just follow the code path to understand what it does. You can then use a similar approach in your own programs.

A PRACTICAL SOUND PROGRAM

Sound effects aren't just for games. In addition to creating music, sound effects can be used in a variety of more serious applications. Sounds can provide cues in a program, tell the user when something happened, or even be the main output of a program. Below is a simple Morse code generator that takes an input string and generates the corresponding dots and dashes.

The program uses many of the things you've learned in previous chapters on Cody BASIC. It accepts input from the user, processes each character in the input string, and uses **IF** statements to look up the corresponding sequence of dots and dashes for each character. In addition to printing out the dots and dashes, it uses sound effects to play short and long tones corresponding to each part of the translated Morse code output.

```
100 REM MORSE CODE GENERATOR
110 U=10
120 GOSUB 700
```

```
130 PRINT "MESSAGE";
140 INPUT M$
150 PRINT
160 GOSUB 200
170 PRINT
180 GOTO 110
200 REM SEND MESSAGE
210 IF M$="" THEN RETURN
220 A=ASC(M$)
230 M$=SUB$(M$,1,LEN(M$))
240 REM CHECK DELAY BETWEEN WORDS
250 IF A<>32 THEN GOTO 300
260 PRINT "<SPACE>"
270 D=7
280 GOSUB 800
290 GOTO 200
300 REM PROCESS NEXT LETTER
310 PRINT CHR$(A),TAB(20);
320 GOSUB 600
330 IF C$<>"" THEN GOTO 360
340 PRINT "NO CODE"
350 GOTO 520
360 REM SEND DOTS AND DASHES
370 B=ASC(C$)
380 C$=SUB$(C$,1,LEN(C$))
390 PRINT CHR$(B);
400 POKE 54276,65
410 IF B=45 THEN D=3
420 IF B=46 THEN D=1
430 GOSUB 800
440 POKE 54276,0
450 REM DELAY BETWEEN BEEPS
460 D=1
470 GOSUB 800
480 IF C$<>"" THEN GOTO 360
490 REM DELAY BETWEEN LETTERS
500 D=3
510 GOSUB 800
520 PRINT
530 GOTO 200
600 REM GET MORSE
601 IF A>=97 THEN A=A-32
602 C$=""
603 IF A=65 THEN C$=".-"
604 IF A=66 THEN C$="-..."
605 IF A=67 THEN C$="-.-."
```

```
606 IF A=68 THEN C$="-.."
607 IF A=69 THEN C$="."
608 IF A=70 THEN C$="..-."
609 IF A=71 THEN C$="--."
610 IF A=72 THEN C$="...."
611 IF A=73 THEN C$=".."
612 IF A=74 THEN C$=".---"
613 IF A=75 THEN C$="-.-"
614 IF A=76 THEN C$=".-.."
615 IF A=77 THEN C$="--"
616 IF A=78 THEN C$="-."
617 IF A=79 THEN C$="---"
618 IF A=80 THEN C$=".--."
619 IF A=81 THEN C$="--.-"
620 IF A=82 THEN C$=".-."
621 IF A=83 THEN C$="..."
622 IF A=84 THEN C$="-"
623 IF A=85 THEN C$="..-"
624 IF A=86 THEN C$="...-"
625 IF A=87 THEN C$=".--"
626 IF A=88 THEN C$="-..-"
627 IF A=89 THEN C$="-.--"
628 IF A=90 THEN C$="--.."
629 IF A=48 THEN C$="-----"
630 IF A=49 THEN C$=".----"
631 IF A=50 THEN C$="..---"
632 IF A=51 THEN C$="...--"
633 IF A=52 THEN C$="....-"
634 IF A=53 THEN C$="-...."
635 IF A=54 THEN C$="-...."
636 IF A=55 THEN C$="---.."
637 IF A=56 THEN C$="----."
638 IF A=57 THEN C$="....."
639 RETURN
700 REM SET UP SOUND
705 FOR I=0 TO 6
710 POKE 54272+I,0
715 NEXT
720 POKE 54296,14
725 POKE 54272,0
730 POKE 54273,30
735 POKE 54275,8
740 POKE 54276,0
745 POKE 54277,0
750 POKE 54278,192
755 RETURN
```

```
800 REM DELAY
810 T=TI
820 L=D*U
830 IF ABS(TI-T)<L THEN GOTO 830
840 RETURN
```

This program generates Morse code for an input string, displaying the dots and dashes on the screen as the corresponding sounds are played.

The provided Morse code example printing the codes for the word 'RADIOACTIVITY'. Note that when run you'll also hear the dots and dashes.

RING MODULATION

Ring modulation modifies one voice using the output of another voice, allowing the programmer to construct a variety of interesting sounds. In addition to producing sound effects, bell-like or gong-like sounds can also be generated using this approach.

Ring modulation on the Cody SID, like the original SID, requires two voices and has some important limitations. Only triangle waves are supported, so the primary voice must be set to output a triangle wave along with the ring modulation bit (bit 2) in the control register. Also unlike real ring modulation, ring modulation for the SID only relies on multiplying the signs of the signals, rather than a full multiplication as in true ring modulation.

The secondary voice that supplies the other input for ring modulation must also be set up with a frequency for any of this to work. Other settings on the secondary voice are ignored and otherwise has no effect on the ring modulation. The corresponding voice used for the secondary voice in ring modulation is hardwired: Voice 1 uses voice 3, voice 2 uses voice 1, and voice 3 uses voice 2.

For an example of ring modulation, see the following Cody BASIC example that generates a somewhat-technological humming sound. In addition to the typical ADSR envelope, it uses voice 1 and voice 3 together. Voice 1 is set up as a triangle wave with ring modulation turned on, and voice 3 is set up with a separate frequency to modulate voice 1's output.

```
10 FOR I=0 TO 29
20 POKE 54272+I,0
30 NEXT
40 POKE 54296,14
50 POKE 54272,0
60 POKE 54273,40
70 POKE 54277,160
80 POKE 54278,251
90 POKE 54286,0
100 POKE 54287,10
110 POKE 54276,21
120 T=TI
130 IF ABS(TI-T)<120 THEN GOTO 130
140 POKE 54276,20
150 T=TI
160 IF ABS(TI-T)<120 THEN GOTO 160
```

A program that produces a low, fading hum. A sound like this could be used for some kind of futuristic machinery or perhaps a teleport between game levels.

9

Input and Output Programming

INTRODUCTION

The Cody Computer has multiple input and output devices built into it. Using the Propeller it has two UARTs for serial communication, one connected to the Prop Plug port and the other to the expansion port on the back. Another chip, the 65C22 Versatile Interface Adapter, implements two 8-bit I/O ports along with some miscellaneous signals and a programmable shift register.

Some of these capabilities are already in use by the Cody Computer. For example, Port A on the 65C22 I/O chip is used to read the keyboard matrix and joystick ports, while port A's control signals are used to check if a cartridge is plugged into the expansion port. Port B, on the other hand, is connected directly to the expansion port for use in your own programs and projects.

Being able to connect your own circuits and peripherals to the Cody Computer opens up many new options and projects. You could write your own machine-language games and store them on a cartridge, effectively turning the Cody Computer into an 8-bit game machine. You could implement modern protocols for communicating with other chips, such as I2C or SPI, and use them to interface with the outside world. Projects requiring simple serial communications (such as reading NMEA sentences from a GPS) could be built with either of the Cody Computer's UARTs, provided the external devices can support the Cody Computer's slower (by modern standards) speeds. And for projects that require extra capabilities, you

could even wire another microcontroller to the expansion port to extend the base system.

> Wiring stuff into the expansion slot or ports is one of the few ways that you could easily destroy your Cody Computer. While modern electronics aren't as brittle or likely to fry as they once were, incorrect connections or voltages could still result in doom. Also be aware that while the Cody Computer's chips can drive 3.3-volt digital signals, you'll want to follow good design practices when connecting up motors, relays, and higher voltages or currents. Think through what you're doing and refer to the 65C22 and Propeller data sheets as well as the Cody Computer's schematics.

KEYBOARD AND JOYSTICK INPUT

We covered the Cody Computer's keyboard in chapter 2, including a discussion of the keyboard matrix and how the joystick ports are actually treated as the last two rows of the keyboard. The keyboard is wired to the 65C22 I/O chip's Port A, which scans the keyboard and joystick using three of its pins. The three pins are decoded into one of eight rows by a 1-of-8 decoder chip, with the five pins for that row or joystick port read back into the 65C22.

In assembly language programs you will have to scan the keyboard and joystick by communicating with the 65C22's Port A directly. However, in your Cody BASIC programs this is handled automatically by the BASIC interpreter. It has an interrupt in the

background that scans the keyboard and joystick matrix many times per second, updating a portion of memory with the data. You can access the values with a **PEEK** statement.

Memory locations **$10** (decimal 16) through **$15** (decimal 21) are populated with the scanned key rows. Memory locations **$16** (decimal 22) and **$17** (decimal 23) store the scans for joystick ports 1 and 2. Because of the Cody Computer's keyboard wiring, the bits are actually inverted, meaning that a 0 indicates a key or button that is pressed, while a 1 indicates that it's not pressed.

To see this in action, try the below Cody BASIC program. It loops over the values in the memory region we just mentioned, then prints out each bit as well as the entire number. You can press keys on your keyboard or use your joystick, then watch as the bits change. The program isn't particularly fast, particular as it has a nested loop that calculates each bit and prints it to the screen.

```
10 PRINT CHR$(222)
20 PRINT AT(0,0);
30 FOR A=0 TO 7
40 D=PEEK(16+A)
50 M=128
60 FOR B=0 TO 7
70 N=0
80 IF AND(D,M)>0 THEN N=1
90 PRINT N;
100 M=M/2
110 NEXT
120 PRINT " (",D,")"
130 NEXT
140 GOTO 20
```

A Cody BASIC program that prints out the current state of the keyboard and joystick matrix.

Once you've played around with the program, try comparing the results you get to the Cody Computer's keyboard schematic (available online or in Chapter 2 of this book). You should be able to match up the key you're pressing with a position in the keyboard matrix, then see the corresponding bits for that row on the screen.

Your own programs don't need to perform the per-bit calculations or display anything at all. The most common use case for reading the keyboard or joystick like this is in a game where you want to determine particular keypresses or joystick actions. For that, you will want to just check the relevant memory locations and bits.

This is particularly relevant for reading the joystick. Even in a BASIC game you may want to read the joystick to move a player around on the screen, and the following example will help get you started. It

reads from the last of the memory locations, **$16** and **$17**, then examines each bit to determine what position the joystick has and whether the fire button is being pushed.

```
10 PRINT CHR$(222)
20 PRINT AT(0,0);
30 FOR I=1 TO 2
40 PRINT "JOY ",I,": ";
50 D=PEEK(16+5+I)
60 PRINT TAB(10);
70 IF AND(D,16)=0 THEN PRINT "FIRE";
80 PRINT TAB(16);
90 IF AND(D,8)=0 THEN PRINT "RIGHT";
100 PRINT TAB(22);
110 IF AND(D,4)=0 THEN PRINT "LEFT";
120 PRINT TAB(28);
130 IF AND(D,2)=0 THEN PRINT "DOWN";
140 PRINT TAB(34);
150 IF AND(D,1)=0 THEN PRINT "UP";
160 PRINT
170 NEXT
180 GOTO 20
```

A Cody BASIC program that reads the joysticks and prints out the current joystick position and fire button status.

In an assembly language program, however, you'll have to scan the keyboard and joystick yourself. Cody BASIC won't be able to help you. However, the techniques you learn in Cody BASIC can make it easier. For example, learning how to map the keyboard and joystick values to the keyboard matrix and computer schematic will give you a head start on understanding how to program them. You can also rely on the existing code within the Cody BASIC interpreter as a place to start writing your own.

SERIAL INPUT AND OUTPUT

The Cody Computer also has two UART (Universal Asychronous Receiver Transmitter) peripherals implemented using the Propeller. These allow the Cody Computer to communicate with other systems over a serial port, with some restrictions. In most respects the Cody Computer UARTs serve a similar function to the 6551 Asynchronous Communications Interface Adapter (ACIA) used in many 6502-based computers, but in reality they're quite different to program.

The Cody Computer UARTs are specific to the needs of the Cody Computer, so they only support a standard 8-N-1 serial configuration with 8 data bits, no parity bit, and one stop bit. It's also entirely polling-based, which means you have to check them on a regular basis from within your program. On the other hand, they have ring buffers for transmitting and receiving bytes, which means you don't have to check them as often. Each UART has a total of 23 registers, almost all of them related to the ring buffer.

A ring buffer is a data structure commonly used for communications, and it consists of a range of memory devoted to storing data. Along with the data are two values indicating the start and the end of the data in the buffer, the head and the tail. When data enters the buffer it's stored at the head position, which is then moved forward. When data is removed from the buffer it's taken from the tail position, which is then moved forward as well. However, the positions actually roll around from the end of the buffer back to the start,

hence the term "ring buffer." (This also means that to determine when the buffer is full, we have to either store a count or look at the distance between the head and tail.)

To actually program a UART, you'll need to **POKE** and **PEEK** its registers just like you have for the other peripherals. UART 1, connected to the Propeller Plug port, resides at **$D480** (decimal 54400). UART 2 is part of the expansion port on the back and resides at **$D4A0** (decimal 54432). From either of those positions, the offsets to a particular register are the same, just shifted by the base address for the UART you're talking to.

The first UART register, register **$0**, is the control register. It sets the baud rate to use when sending or receiving data. The baud rate goes into the lower half of the register, with the current half of the register currently being unused. Similar to the Cody SID, you'll need to look up the matching baud rate for each number in the following table. The values are actually taken from the 6551's baud rate options and do not follow any standard progression.

Value (dec)	Value (hex)	Bit Rate
0	$0	Invalid
1	$1	50
2	$2	75
3	$3	110
4	$4	135
6	$6	300
7	$7	600
8	$8	1200

Value (dec)	Value (hex)	Bit Rate
9	$9	1800
10	$A	2400
11	$B	3600
12	$C	4800
13	$D	7200
14	$E	9600
15	$F	19200

The Cody Computer's UART baud rate table. Inspired by the 6551's baud rate options, these values cover the common baud rates for systems of a particular vintage.

The second UART register, register **$1**, is the command register. It consists of a single bit at bit 0 that turns the UART on and off. Setting it to 1 turns the UART on, while setting it to 0 resets the UART. After you turn the UART on or off, you need to check the UART's status register to ensure it has processed the command. (We'll cover that in a minute.)

The third UART register, at **$2**, is the status register. It provides a window into what the UART is currently doing. Bit 0 is unused. Bit 1 is set to 1 if a framing error has occurred, indicating that a stop bit wasn't received as expected. Bit 2 is set to 1 if an overrun has occurred, meaning that more data was coming into a receive buffer than there was room to store it. Bits 3 and 4 indicate if data is currently received or transmitted, respectively. Bit 6 indicates whether or not the UART is running and should be polled when the UART is turned on or off to wait until the UART is in the proper mode.

The fourth register at **$3** is reserved. The next two registers, **$4** and **$5**, contain the head and tail positions for the UART's receive buffer. The UART will update the head position as data is received, while you must update the tail position as you read from it.

A similar situation exists for registers **$6** and **$7**, the transmit ring buffer head and tail positions. Because you are putting data to be sent into the buffer, you will be the one to update the head position. The UART will update the tail position as it sends the data.

The remaining registers consist of the receive and transmit ring buffers. The receive buffer starts at **$8** and goes on for 8 bytes. The transmit buffer starts immediately after at **$10** and goes on for an additional 8 bytes. Because of the nature of the ring buffer implementation used by the Cody Computer, only seven bytes can be in use at any one time. This is because to store a full eight bytes, the head and tail positions would be equal, a case indistinguishable from an empty buffer without additional information (such as a count). Rather than make the implementation more complicated, to keep things simple the maximum capacity is limited by one byte.

TRANSMITTING DATA

Now that we've had a bit of theory on the UART, consider the following example Cody BASIC program. It will collect some information from you, including a string to send over the serial port. It then turns the UART on, waits for it to start up, configures it and sends the string as ASCII values. It also has to poll the

ring buffer as it empties to fill it up with the rest of the data you're trying to send.

To run the program you should be able to use the same serial program you've been using to communicate with the Cody Computer until now. You'll just need to set it up to receive with the baud rate you select, and then begin sending data to it using this program.

```
10 REM UART TRANSMIT EXAMPLE
20 PRINT "UART (1-2)";
30 INPUT U
40 IF U=1 THEN A=54400
50 IF U=2 THEN A=54432
60 PRINT "BAUD RATE (1-15)";
70 INPUT B
80 PRINT "TEXT";
90 INPUT S$
100 REM STRING TO BYTES
110 L=LEN(S$)
120 I=0
130 IF I=L THEN GOTO 180
140 S(I)=ASC(S$)
150 S$=SUB$(S$,1,LEN(S$)-1)
160 I=I+1
170 GOTO 130
180 REM CONFIGURE UART
190 POKE A+1,0
200 IF AND(PEEK(A+2),64)>0 THEN GOTO 200
210 POKE A+0,B
220 POKE A+6,0
230 POKE A+1,1
240 IF AND(PEEK(A+2),64)=0 THEN GOTO 240
250 REM TRANSMIT LOOP
260 FOR I=0 TO L-1
270 H=PEEK(A+6)
280 T=PEEK(A+7)
290 IF ABS(H-T)>6 THEN GOTO 270
300 POKE A+16+H,S(I)
310 POKE A+6,MOD(H+1,8)
320 PRINT "SENDING CHR '",CHR$(S(I)),"' (",S(I),
```

```
330 NEXT
```

A short example in Cody BASIC that shows how to send data by low-level programming of a UART. In practice these operations would be done either by the BASIC interpreter itself or from within an assembly language program.

There are a few key parts of this program. Note how the UART base address is selectable. Also note how the program breaks the string you enter into a series of numbers to send via the UART. Regarding the actual UART programming, the program turns the UART off and waits for the status register to update. It then sets up the baud rate and configures the UART before turning it back on, again waiting for the status register.

For the main loop, it uses an approach common to working with a ring buffer. It checks the head and tail positions, then performs a quick subtraction to see if the buffer is full. If not, it adds another character to send, then increments the head position so that the UART knows to pick it up. Because the values wrap around, there are some additional things the program does, such as using modular arithmetic when incrementing a value or an absolute value when performing a subtraction.

In a real program, it would be a good idea to shut the UART off when it's done. To keep this example as minimal as possible, that's not done here. In a lower level program written in assembly language, constantly polling and busy-waiting would also leave much to be desired. In that situation, it's better to perform the polling on a periodic basis, or to

interleave a quick check of the UART into the main loop of your program.

RECEIVING DATA

The UART also receives data when turned on. The baud rate option set into the control register is used for receive and transmit and both operations occur simultaneously (the UART is "full duplex" rather than "half duplex"). The receive ring buffer is populated with the incoming data and the UART automatically updates the receive buffer head register as new data arrives. The programmer is responsible for reading data from the buffer and updating the tail register, exactly the opposite as what happens when transmitting via the UART.

The following Cody BASIC program sets up the UART to receive data. You can run it in the same manner as the transmit example above but using your serial program to send characters to the Cody Computer instead. Note that because the entire program is written in Cody BASIC, it runs very slowly compared to assembly language, and there's significant overhead. While it can support even the highest available baud rates for the UART, you will likely need to insert a per-character delay inside your serial program to communicate without overrunning the buffer. Otherwise this little program just won't be able to keep up.

```
10 REM UART RECEIVE EXAMPLE
20 PRINT "UART (1-2)";
30 INPUT U
40 IF U=1 THEN A=54400
```

```
50 IF U=2 THEN A=54432
60 PRINT "BAUD RATE (1-15)";
70 INPUT B
80 REM CONFIGURE UART
90 POKE A+1,0
100 IF AND(PEEK(A+2),64)>0 THEN GOTO 100
110 POKE A+0,B
120 POKE A+5,0
130 POKE A+1,1
140 IF AND(PEEK(A+2),64)=0 THEN GOTO 140
150 REM RECEIVE LOOP
160 E=PEEK(A+2)
170 IF AND(E,2)>0 THEN GOTO 260
180 IF AND(E,4)>0 THEN GOTO 280
190 H=PEEK(A+4)
200 T=PEEK(A+5)
210 IF H=T THEN GOTO 160
220 C=PEEK(A+8+T)
230 POKE A+5,MOD(T+1,8)
240 PRINT "RECEIVED CHR '",CHR$(C),"' (",C,")"
250 GOTO 160
260 PRINT "FRAMING ERROR"
270 END
280 PRINT "OVERRUN ERROR"
290 END
```

A Cody BASIC example of receiving data from a UART at a low level. This is only an example that unfortunately runs quite slowly. In actual usage the program would likely be written in assembly language if the existing Cody BASIC input routine was insufficient.

The overall program flow is very similar to the transmit example. It obtains the configuration data from the user, turns the UART off to reset it, turns it back on and waits for it to come up, sets the UART up, and begins listening. Each time a new character is found in the buffer, it's removed from the buffer and an update message is printed to the screen.

Unlike the transmit example, this example checks the status register for the UART's two error modes, both of which only show up when receiving. A framing error (bit 1 in the status register) indicates that the UART didn't read a stop bit when expected, meaning that something was out of whack (perhaps different baud rates between sender and receiver, or perhaps the sender wasn't sending 8-N-1). An overrun error (bit 2 in the status register) means that the program couldn't read data out of the buffer as fast as it was coming in, and the UART ran out of room to store more data.

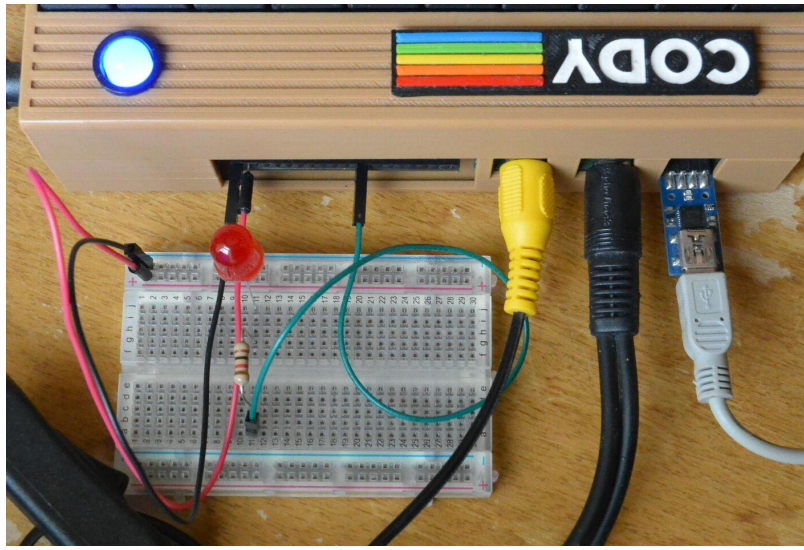
The examples show transmit and receive separately, but keep in mind that the Cody UARTs can do both at the same time. Setting up the UARTs is exactly the same, but both the receive and transmit buffers would need to be checked and updated to support simultaneous transmit and receive.

It's not a particularly difficult task, but it's one best left to low-level programs in assembly language. For high-speed communication using the UARTs in Cody BASIC, you're best off using the **OPEN** statement to redirect **INPUT** and **PRINT** statements to the serial port. This topic is covered in Chapter 6 while discussing how to read and write text files over a serial link, but the same technique can be used for general text-based serial input and output. (Even binary data could be sent across if a hex or other encoding is used, albeit with some additional overhead.)

GENERAL-PURPOSE INPUT AND OUTPUT

Aside from the UART and some of the special 65C22 pins (such as its built-in shift register), most of the pins on the Cody Computer's expansion port are not dedicated to any particular use. These can be configured either as inputs or outputs by setting the 65C22's Data Direction Register B at address **$9F02** (decimal 40706). By default, each bit is zero and configured as an input, but setting the bit to 1 makes it an output instead. Output values for each pin can be specified by writing to IO Data Register B at address **$9F00** (decimal 40704), while reading the same register will return the input values for the input pins.

As a simple example we'll use one of the pins to blink an LED. To build this circuit you will need a small breadboard. Expansion port pin 1 (counting from the rightmost side when looking down on the Cody Computer) should be connected to the ground row, pin 2 should be connected to the positive voltage row, and pin 12 should be connected to an LED through a current-limiting resistor. The LED's anode (long lead) should be connected to the resistor's other terminal, with its cathode (the short lead) connected to ground. The Cody Computer's expansion port is not designed to be hot-plugged, so turn the computer off when wiring to it, then turn it back on when you're finished.



The simple breadboard circuit at left blinks an LED under the Cody Computer's control.

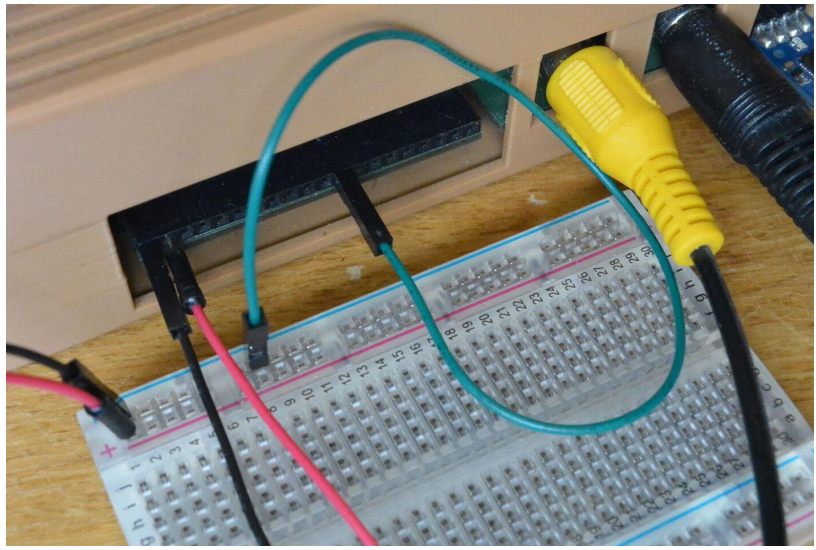
Once wired up, the following Cody BASIC program can be used to blink the LED on and off for a few cycles. It clears the data register then sets up output pin 1 as an output by writing to the data direction register. After that, bit 1 of the data register is toggled off and on in a loop with a brief delay, blinking the LED.

```
10 POKE 40704,0
20 POKE 40706,1
30 FOR I=0 TO 9
40 POKE 40704,1
50 T=TI
60 IF TI-T<60 THEN GOTO 60
70 POKE 40704,0
80 T=TI
90 IF TI-T<60 THEN GOTO 90
100 NEXT
110 POKE 40706,0
```

A program to blink an LED.

Each pin can also be used as an input when the corresponding bit in the data direction register is turned off. In this case, the input bits can be read by reading from the port B data register as mentioned above.

A simple circuit based on the LED circuit can be used to show this. The LED and resistor are no longer needed, and the wire connected to pin 12 of the expansion port can instead be plugged into the 3.3 volt or ground buses for an input value of 1 or 0 respectively. However, you should be careful when rewiring the circuit and running the program below, as you don't want to plug the pin into one of the buses when set up in output mode. Instead, as before, wire up the circuit when the computer is off, then turn the computer on.



An even more simple circuit can be used to drive an input pin using either the 3.3V and ground lines.

The following Cody BASIC program will read the input pin and display its current value. The data direction register is set to zero, then the data register itself is read in a loop. The value for pin 1 is selected using an **AND** function (unconnected input pins can flap between 0 and 1 so bit-masking the value we want makes the output clearer to read). When the program is running, you can move the input wire back and forth between the 3.3 volt and ground lines to produce a 1 or 0 input.

```
10 POKE 40706,0
20 I=PEEK(40704)
30 PRINT AND(I,1)
40 GOTO 20
```

A program to read and display a single input bit.

SPECIAL PINS AND SHIFT REGISTERS

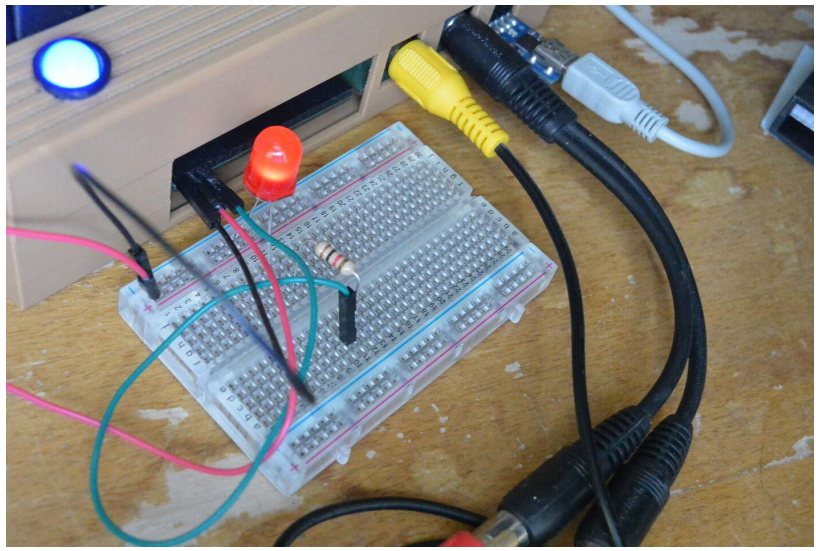
The 65C22 also has two handshaking ports consisting of two pins each. The pins for port A CA1 and CA2, are already in use as a cartridge-detect mechanism for the Cody Computer. The others, CB1 and CB2, are free for use in your own projects. While these pins can be used to implement a handshaking mechanism for 8-bit data transfer across port B as discussed in the 65C22's data sheet, there are also other possibilities.

One possibility is to use the pins as an interrupt input. This would allow external devices to signal that something has occurred and have an interrupt handler run in an assembly language program. Another interesting option is to configure the pins as a shift register, letting you clock data in or out on a periodic basis.

None of these scenarios are trivial, so if you intend to do something like this in your own projects, you'll want to refer to the 65C22 data sheet. It's also difficult to come up with good examples of more advanced features without having some other parts around that can use them, so by necessity this section is somewhat limited. We can demonstrate the shift register function using an LED, but to follow along, it would be helpful to have access to an oscilloscope or other means of seeing the actual signal.

First you'll need a circuit. For those without any kind of oscilloscope or logic analyzer tool, you'll want a circuit very similar to the LED circuit earlier in this

chapter. However, in this case, instead of connecting the LED's resistor to expansion port pin 12, you'll connect it to expansion port pin 3. Expansion port pin 3 is wired to the 65C22's CB2 pin, which has the actual data coming out of the shift register.



An LED circuit connected to the expansion port's CB2 pin. The LED brightness changes depending on the data sent out of the shift register. Here it glows a dull red because few of the bits in the data sequence are ones.

The 65C22 supports various shift register modes for both input and output using different clock signal sources. Most of the configuration happens through the 65C22's Auxiliary Control Register at address **$9F08** (decimal 40715). For this example, we're going to be setting it up as a simple output controlled by the 65C22's Timer 2 internal clock. This means that bits through 2 through 4 of that register need to be set to binary **100** according to the data sheet.

We also need to set up 65C22 timer 2 to generate the clock signal. Each time the Cody Computer's system clock ticks, Timer 2 will decrement by one. We give the timer a value to count down from, and the time it takes to count to zero ends up being the time for one phase of the clock. The timer 2 counter is a 16-bit value with the low byte at address **$9F08** (decimal 40712) and the high byte at address **$9F09** (decimal 40713). We write the low byte followed by the high byte, with the writing of the high byte triggering the timer's clock to restart with the new timer value.

The shift register's output is kept in a register at address **$9F0A** (decimal 40714). The value written there continues to be reused until a new value is programmed in. Other registers or interrupts can be used to determine when the shift register needs to be fed new data, but for our simple example, we're fine with the value wrapping around.

You can see all this put together in a small Cody BASIC program. It prompts you for a value to write to the shift register, then sets up the shift register and timer 2 with the longest possible delay in this mode. Counting down from 65535 with a 1-megahertz system clock means that the shift register sends out a new bit about every .07 seconds, which is too fast to see without some way to capture the actual signal.

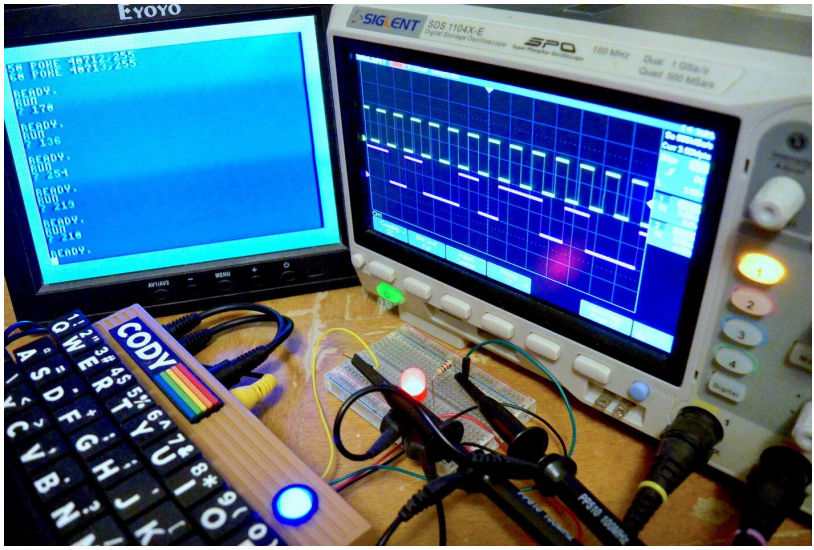
```
10 INPUT I
20 POKE 40714,I
30 C=OR(AND(PEEK(40715),227),16)
40 POKE 40715,C
50 POKE 40712,255
60 POKE 40713,255
```

A program to send a pattern out of the shift register.

However, different patterns will change the brightness of the connected LED because it will be on or off for different periods of time. For example, a value of 255 is all ones, which means the LED will be at maximum brightness, while a value of 0 is all zeroes, so the LED will be off. A decimal value of 170 corresponds to a binary 10101010, while a decimal value of 136 corresponds to 10001000. Try different values and see their results.

If you do have an oscilloscope around, you can actually see the individual zeroes and ones. The 65C22's CB1 pin is connected to expansion port pin 4 and acts as the shift register's clock. The 65C22's CB2 pin is connected to expansion port pin 3 and actually sends (or receives) the data. Connect your first oscilloscope probe to expansion port pin 4, your second oscilloscope probe to expansion port pin 3, and set up your oscilloscope to trigger on the first probe.

You should see a square wave for the clock signal and a sequence of highs and lows for the data signal corresponding to whatever number you typed in. This isn't purely an academic exercise, as you might end up having to do pretty much the same thing to track down bugs when bit-banging various protocols out of the expansion port. A logic analyzer would also suffice.



Watching the shift register's clock and data pins using an oscilloscope. The yellow trace shows the shift register's clock and the purple trace shows the shift register's data output. The clock will always be the same but the data will change based on what's being shifted out.

Remember that the shift register isn't just used for output. It can also be used for input from an external device. It's just a matter of wiring it up and then writing the appropriate software in Cody BASIC or assembly language to talk to it.

Note that the 65C22 shift register is not compatible with SPI communications, though there are some hacks to work around it for one particular SPI mode (the Steckschwein retrocomputer actually does this to implement an SPI master). For this reason the Cody Computer implements SPI in software, as you'll learn in the next section. However, the 65C22's CB pins can do a lot, and you should refer to the 65C22 data sheet to learn more about them. And for your own Cody Computer peripherals, you can do it your way.

SPI COMMUNICATION AND CARTRIDGES

The Cody Computer's expansion port is a relatively general-purpose device. With the few exceptions noted above, every pin is programmable as an input or an output and can be directly controlled from either BASIC or assembly language. By themselves or with minimal additional hardware they can even implement more modern data protocols such as Inter-Integrated Circuit (I2C) or the Serial Peripheral Interface (SPI).

In fact, some of the general-purpose pins also have a designated special use to load programs from cartridges. Like many computers of the 8-bit era, the Cody Computer supports program cartridges that can be plugged directly into the expansion port. If one is detected using the CA lines, the Cody Computer's

ROM will load the program from the cartridge over SPI and run that program instead of Cody BASIC.

This topic is complex enough to warrant a separate discussion. More details are provided in Chapter 11, Cartridges and SPI.

10

Assembly Language Progamming

INTRODUCTION

In this chapter we'll provide some examples of programming the Cody Computer in 65C02 assembly language. The chapter isn't an introduction to the 65C02's assembly language in itself. If you haven't worked with it before, you're better off learning the basics using an online emulator before digging into these examples. The 6502 family, while decades old, was one of the most popular microprocessor families in existence. Documentation, both historical and modern, is plentiful online.

Regarding the chip itself, the 65C02 is essentially an updated 6502 with some additional instructions added and invalid ones removed. It has a very small number of registers—an accumulator (A), two index registers (X and Y), and some additional registers for stack and CPU flag management. It supports most of the addressing modes typical for a chip of its era, including direct addressing, indexed addressing, and some forms of indirect addressing. It also uses a range of 256 "zero-page" addresses that, while stored in main RAM rather than the processor, can be viewed as being a huge bank of low-cost registers.

In its day it was the affordable alternative to more expensive microprocessor or microcontroller families. Many of the most popular 8-bit computers utilized the 6502 family for their main processor, and 16-bit variants of the family went on to be used in later computers, add-ons, and game consoles. The same efficiency and elegance that made the chip so popular

in prior decades is also put to good use by the Cody Computer.

This chapter introduces two small assembly language programs. The first is a SID player that can play many, but not all, Commodore 64 SID music files. The second is a simple game demo inspired by 1980s platformers to show some of the Cody Computer's sound and graphics capabilities. The programs are not too complicated, but without a basic grasp of 65C02 assembly programming, they can be a bit much to digest. If you've programmed in another assembly language but haven't worked with the 65C02, you'll probably be able to at least follow along. Having a 65C02 reference will be handy.

Just as with Cody BASIC, the assembly language programs are written using *64tass*, a 6502-family assembler for the Commodore computers that can also generate generic 65C02 code. This assembler is both open-source and freely downloadable, so installing or building a copy should not be difficult on any of today's major computing platforms.

THE CODYSID MUSIC PLAYER

A simple SID player is a good project for assembly language. It requires low-level programming, including reading a SID file over the UART, loading it into memory, and calling its functions on a regular basis to play the song. SID files have some unique characteristics that make it easier to write a player, yet these same characteristics also make it less likely that any particular SID file will play on the Cody Computer.

At its core a SID file is just a program with a load address and some functions to call. One of the functions is the **INIT** routine that sets up the SID file. Another is a **PLAY** routine that plays the current portion of the song when called on a regular basis by the player. Everything else, including the way the music data is stored, is under the control of the person who wrote the SID.

This is very different from more traditional music formats such as MIDI that contain structured data about the song. Because a SID file is a program, each SID has its own unique expectations about where it will be loaded, how it will be called, the memory layout of the system, and what peripherals (including interrupts and timers) are present.

While the Cody Computer has a rudimentary SID built in, it's not a Commodore 64. As a result many perfectly valid SID files will fail to play on it. However, many of them will, particularly if we constrain ourselves to a certain subset of SID file types and carefully look at their sizes and load addresses. For now, we'll limit ourselves to PSID files of version 2, then prepare ourselves for a certain amount of disappointment.

Even some incompatible SIDs might work after running them through a relocator tool such as Linus Akesson's **sidreloc**. Another option would be to write a player for Compute! Magazine's MUS file format, which is more MIDI-like and has fewer hardware dependencies. We won't be covering any of that in this book.

THE PSID FILE FORMAT

There are several versions of the SID file format. PSID files are less platform-specific and more amenable to playing them without full C64 compatibility. RSID files, on the other hand, generally require a full emulator or real C64. We'll limit ourselves to PSID files, and within that category, we'll only support version 2 of the format. This still leaves us with many songs to try out.

The file begins with a header containing some information about the song. Much of this we don't care about at all. A few parts of it, such as the song name, author, and other related information, are nice to know but not necessary for playing it. A few pieces of information related to function addresses within the SID file are required, so we'll have to get those from the header. We'll also need to take into account that the header is in a big-endian format but the 65C02 works as a little-endian system.

After the header comes the actual SID data. Because of the assumptions we've made, we can expect the SID data will begin with the load address for the SID itself. This tells us where to copy it into memory, and we hope that it won't conflict with our own unique memory layout. (There's actually a field for this in the header, but it's usually not populated and we ignore it for our purposes.)

Once the SID is loaded starting at its load address, we have to set up a periodic timer interrupt to call the song's code and play it. The SID itself needs us to call its **INIT** function before each time we play, then call its

PLAY routine on each timer interrupt to keep the song playing. (It's actually possible for a SID to contain multiple songs, something we handle when calling the **INIT** function.)

As far as the actual music data, it's just contained somewhere within the SID code and data we loaded. We don't know how it's stored, what it does, or much of anything about it without reverse-engineering the file itself. In many respects writing a SID player is more like writing a program loader, and it's one of the reasons this project is relatively straightforward.

You can find many references online to the SID file format if you're interested in the details. For what we're going to write, this is sufficient to begin going through the code. Any little details we haven't covered here will be mentioned as we go through the **CodySID** program.

THE CODYSID PROGRAM

The CodySID source code starts with constant definitions referring to various memory addresses that will be used by the program. Many of these you've already heard of in earlier chapters, such as the UART 1 and 65C22 VIA register addresses. We'll need the UART to load the SID files, while we need the VIA to scan the keyboard and run a timer. Other addresses include the base addresses of the current screen memory and the SID.

```
ADDR      = $0300               ; The actual loading address of the program

SCRRAM    = $C400               ; Screen memory base address
SIDBASE   = $D400               ; SID register base address

UART1_BASE  = $D480             ; Register addresses for UART 1
UART1_CNTL  = UART1_BASE+0
UART1_CMND  = UART1_BASE+1
```

```
UART1_STAT  = UART1_BASE+2
UART1_RXHD  = UART1_BASE+4
UART1_RXTL  = UART1_BASE+5
UART1_TXHD  = UART1_BASE+6
UART1_TXTL  = UART1_BASE+7
UART1_RXBF  = UART1_BASE+8
UART1_TXBF  = UART1_BASE+16

VIA_BASE  = $9F00              ; VIA base address and register locations
VIA_IORB  = VIA_BASE+$0
VIA_IORA  = VIA_BASE+$1
VIA_DDRB  = VIA_BASE+$2
VIA_DDRA  = VIA_BASE+$3
VIA_T1CL  = VIA_BASE+$4
VIA_T1CH  = VIA_BASE+$5
VIA_SR    = VIA_BASE+$A
VIA_ACR   = VIA_BASE+$B
VIA_PCR   = VIA_BASE+$C
VIA_IFR   = VIA_BASE+$D
VIA_IER   = VIA_BASE+$E
```

Constants for many of the peripherals' register locations.

The program will also need some places to put its data. These include **STRPTR** to loop through text strings, **SCRPTR** for the current location in screen memory, and **SIDPTR** to point to the beginning of the loaded SID data. Other data includes **SONGNUM** for the current SID song, a **PLAYBIT** flag indicating if a song is playing, and several **KEYROW** variables containing the current keyboard matrix as of the last scan. (Because we need to register our own interrupt service routine on top of the one built into Cody BASIC, we also define **ISRPTR** to know where the ISR address needs to go.)

```
ISRPTR    = $08                ; Pointer to the ISR address zero page variable

STRPTR    = $D0                ; Pointer to string (2 bytes)
SCRPTR    = $D2                ; Pointer to screen (2 bytes)
SIDPTR    = $D4                ; Pointer to SID load address (2 bytes)
SONGNUM   = $D8                ; Song number
PLAYBIT   = $D9                ; Play bit (are we playing a song?)
KEYROW0   = $DA                ; Keyboard row 0
KEYROW1   = $DB                ; Keyboard row 1
KEYROW2   = $DC                ; Keyboard row 2
KEYROW3   = $DD                ; Keyboard row 3
KEYROW4   = $DE                ; Keyboard row 4
KEYROW5   = $DF                ; Keyboard row 5
```

Assorted zero-page variables for memory locations, song status, and keyboard matrix status.

Many of the constants are dedicated to the SID header. Our program will load the header into a fixed address at **$0200** as denoted by the **SIDHEAD** constant. From there we have offsets into the header portions our program might actually need, such as the init routine address (**SIDINIT**), play routine address (**SIDPLAY**), and song information (**SIDNAME** for the name, **SIDAUTH** for the author, **SIDRELE** for the release/copyright info, and **SIDSNUM** for the number of songs).

```
SIDHEAD   = $0200              ; Page to store the SID file header
SIDLOAD   = SIDHEAD+$08
SIDINIT   = SIDHEAD+$0A
SIDPLAY   = SIDHEAD+$0C
SIDNAME   = SIDHEAD+$16
SIDAUTH   = SIDHEAD+$36
SIDRELE   = SIDHEAD+$56
SIDSNUM   = SIDHEAD+$0E
```

Offsets within the SID header.

Two 16-bit values define the program header for the Cody Computer. When Cody BASIC tries to load a machine language program, it needs to know where to put it and how long it is. This means that each program begins with a load address and an ending address. We can calculate these using the **ADDR** constant and **LAST**

label we define. We also tell the *64tass* assembler to start generating code starting at our load address using the **.LOGICAL** directive.

```
; Program header for Cody Basic's loader (needs to be first)

.WORD ADDR                      ; Starting address (just like KIM-1, Commodore, etc.)
.WORD (ADDR + LAST - MAIN - 1)  ; Ending address (so we know when we're done loading)

; The actual program goes below here

.LOGICAL    ADDR                ; The actual program gets loaded at ADDR
```

Creating the program header and telling the assembler where our program will start.

On startup, control begins in the **MAIN** routine right at the load address. In our case it performs all the initial setup, such as enabling our interrupt service routine, turning on the timer, and preparing to scan the keyboard. After that it tries to load a SID file, then enters the program's main loop. User input from the keyboard is mapped to the menu options, and as the user makes selections, the program branches to the corresponding code.

```
;
; MAIN
;
; Main loop of the SID player. Responsible for initialization, information display,
; and menu selection.
;
MAIN        SEI
            STZ PLAYBIT         ; Not playing by default

            LDA #%00000111      ; Set VIA data direction register A to 00000111 (pins 0-2 outp
            STA VIA_DDRA

            LDA #<TIMERISR      ; Set up timer ISR location
            STA ISRPTR+0
            LDA #>TIMERISR
            STA ISRPTR+1

            LDA #<20000         ; Set up VIA timer 1 to emit ticks for timing purposes
            STA VIA_T1CL
            LDA #>20000
            STA VIA_T1CH

            LDA #%01000000      ; Set up VIA timer 1 continuous interrupts, no outputs
            STA VIA_ACR

            LDA #%11000000      ; Enable VIA timer 1 interrupt
            STA VIA_IER
```

```
            CLI                 ; Turn on interrupts

            JSR CMDLOAD         ; Always start by loading and playing a song

_MENU       JSR SHOWMENU        ; Always print the menu just in case

_SCAN       JSR SHOWREGS

            LDA KEYROW0         ; Pressed Q for quit?
            AND #%00001
            BNE _QUIT

            LDA KEYROW1         ; Pressed L for load?
            AND #%10000
            BNE _LOAD

            LDA KEYROW2         ; Pressed N for next?
            AND #%01000
            BNE _NEXT

            LDA KEYROW5         ; Pressed P for previous?
            AND #%10000
            BNE _PREV

            BRA _SCAN           ; Repeat main loop

_QUIT       JSR STOPSID         ; Shut off SID

            SEI                 ; Disable interrupts

            RTS                 ; Return to BASIC and hope it works

_LOAD       JSR CMDLOAD         ; Run the load command
            BRA _MENU

_NEXT       LDA KEYROW2         ; Wait for N key to be released
            BNE _NEXT

            JSR STOPSID         ; Stop playing music

            LDA SONGNUM         ; Increment song number if within range, else play
            INC A
            CMP SIDSNUM
            BEQ _PLAY

            STA SONGNUM         ; Update song number and play
            BRA _PLAY

_PREV       LDA KEYROW5         ; Wait for P key to be released
            BNE _PREV

            JSR STOPSID         ; Stop playing music

            LDA SONGNUM         ; If song number at zero, just play the song
            BEQ _PLAY

            DEC SONGNUM         ; Otherwise decrement song number and then play
            BRA _PLAY

_PLAY       JSR SHOWINFO
            JSR STARTSID
            BRA _MENU
```

CodySID's main routine. It begins by setting up the Cody Computer, loading the first SID, and then entering the main loop to handle menu selections.

Two routines act as a bridge between the CodySID program and the SID's own routines. **STARTSID** starts

the SID using the current song number and calling its init address. **STOPSID** stops playing of the SID by clearing the play flag and resets the SID's registers. Note how interrupts are disabled during certain parts as we don't want the SID to play in the middle of making these kinds of changes.

```
;
; STARTSID
;
; Begins playing the SID by calling its INIT function.
;
STARTSID    SEI                 ; Initialize and start playing the SID file
            LDA SONGNUM
            JSR _CALLINIT
            LDA #1
            STA PLAYBIT
            CLI
            RTS
_CALLINIT   JMP (SIDINIT)

;
; STOPSID
;
; Stops the currently playing SID.
;
STOPSID     SEI
            STZ PLAYBIT
            CLI

            LDA #0
            LDX #0
_LOOP       STA SIDBASE,X
            INX
            CPX #25
            BNE _LOOP
            RTS
```

*Routines for starting and stopping SID file playback. The **PLAYBIT** variable is a flag indicating the current play status.*

We need a routine to load a SID when the user requests it. The **CMDLOAD** routine handles this by displaying an appropriate message on the screen, then loading a SID using the **LOADHEAD** and **LOADDATA** routines. After the file is loaded some quick byte-swaps are done to convert certain addresses from big-endian to little-endian. Before returning, the load routine starts playing the SID.

```
;
; CMDLOAD
;
; Implements the menu option to load a SID file over the UART connection.
;
CMDLOAD     JSR STOPSID         ; Stop the current song and clear the SID registers

            JSR SHOWSCRN        ; Clear screen

            LDX #Ø              ; Display message about waiting to receive SID file
            LDY #3
            JSR MOVESCRN

            LDX #MSG_RECEIVE
            JSR PUTMSG

            JSR UARTON          ; Receive the SID file
            JSR LOADHEAD
            JSR LOADDATA
            JSR UARTOFF

            LDA SIDINIT+Ø       ; Swap INIT address bytes (big-endian in PSID header)
            PHA
            LDA SIDINIT+1
            STA SIDINIT+Ø
            PLA
            STA SIDINIT+1

            LDA SIDPLAY+Ø       ; Swap PLAY address bytes (big endian in PSID header)
            PHA
            LDA SIDPLAY+1
            STA SIDPLAY+Ø
            PLA
            STA SIDPLAY+1

            LDA SIDSNUM+Ø       ; Swap song count address bytes (big endian in PSID header)
            PHA
            LDA SIDSNUM+1
            STA SIDSNUM+Ø
            PLA
            STA SIDSNUM+1

            STZ SONGNUM         ; Always start at first song

            JSR SHOWSCRN        ; Clear screen

            JSR SHOWINFO        ; Display the info of the SID file we read

            JSR STARTSID        ; Start playing the current SID and song

            RTS                 ; All done
```

*The **CMDLOAD** routine handles SID file loading at a high level.*

Support routines include the **KEYSCAN** routine for scanning the keyboard matrix and the **TIMERISR** routine for handling timer interrupts. Both of these are very similar to routines in the Cody BASIC interpreter except for the SID specific behavior. **TIMERISR** calls **KEYSCAN** to update the keyboard variables scanned

by the main routine, and it also calls the SID's play routine when a song is playing.

```
;
; KEYSCAN
;
; Scans the keyboard matrix (so that key selections for menu options can be detected).
;
KEYSCAN     PHA                 ; Preserve registers
            PHX

            STZ VIA_IORA        ; Start at the first row and first key of the keyboard
            LDX #0

_LOOP       LDA VIA_IORA        ; Read the keys for the current row from the VIA port
            EOR #$FF
            LSR A
            LSR A
            LSR A
            STA KEYROW0,X

            INC VIA_IORA        ; Move on to the next keyboard row
            INX

            CPX #6              ; Do we have any rows remaining to scan?
            BNE _LOOP

            PLX                 ; Restore registers
            PLA

            RTS
```

*A simple routine for scanning the keyboard matrix and storing the results into the **KEYROW** zero-page variables.*

```
;
; TIMERISR
;
; A timer interrupt handler that scans the keyboard and calls the SID's play routine.
;
TIMERISR    BIT VIA_T1CL        ; Clear 65C22 interrupt by reading

            PHA                 ; Preserve registers
            PHX
            PHY

            JSR KEYSCAN         ; Scan the keyboard

            LDA PLAYBIT         ; Are we playing?
            BEQ _DONE

            JSR _CALLPLAY       ; Call the play routine

_DONE       PLY                 ; Restore registers
            PLX
            PLA

            RTI                 ; All done
```

```
_CALLPLAY   JMP (SIDPLAY)
```

*The SID player's **TIMERISR** updates the keyboard variables and plays the next part of the song if playing.*

Loading of the SID data is handled by the **LOADHEAD** and **LOADDATA** routines. These are called once the UART is turned on and rely on various UART helper routines to read incoming bytes. Because we have no specific end-of-file for the incoming SID data, we rely on a timeout instead. This could be a problem over an unreliable serial link, but relatively low baud rates over modern communications are generally reliable. If you find yourself having intermittent problems, check your connections and cables.

```
;
; LOADHEAD
;
; Loads a SID file header into the SIDHEAD page. Assumes PSID version 2.
;
LOADHEAD  LDX #0

_READ     JSR UARTGET
          BCC _READ

          STA SIDHEAD,X
          INX

          CPX #$7C
          BNE _READ

          RTS

;
; LOADDATA
;
; Loads the SID file data into memory. The routine assumes the load address
; must be read from the file (not included in the SID header).
;
LOADDATA

_READ1    JSR UARTGET
          BCC _READ1
          STA SIDPTR+0

_READ2    JSR UARTGET
          BCC _READ2
          STA SIDPTR+1

          LDX #$FF

_READ3    DEX
          BEQ _DONE

          JSR UARTGET
          BCC _READ3
```

```
        LDX #$FF            ; Reset counter

        STA (SIDPTR)        ; Store data

        INC SIDPTR+0        ; Increment load address
        BNE _READ3
        INC SIDPTR+1
        BRA _READ3

_DONE   RTS
```

***LOADHEAD** and **LOADDATA** copy the SID's contents from the UART into the Cody Computer's memory.*

Important information in the SID header is shown to the user when the file is playing. In CodySID this is handled in the **SHOWINFO** routine, which moves to certain positions on the screen and prints the SID's name, author, copyright information, song numbers, and code addresses.

```
;
; SHOWINFO
;
; Displays SID information on the screen. This includes the song name,
; author, release/copyright, load/init/play addresses, and song number.
;
SHOWINFO  LDX #0              ; Move to song name position
          LDY #3
          JSR MOVESCRN

          LDX #0              ; Print song name from header
_NAME     LDA SIDNAME,X
          JSR PUTCHR
          INX
          CPX #32
          BNE _NAME

          LDX #0              ; Move to song author position
          LDY #4
          JSR MOVESCRN

          LDX #0              ; Print song author from header
_AUTH     LDA SIDAUTH,X
          JSR PUTCHR
          INX
          CPX #32
          BNE _AUTH

          LDX #0              ; Move to song release/copyright position
          LDY #5
          JSR MOVESCRN

          LDX #0              ; Print song release/copyright information
_RELE     LDA SIDRELE,X
          JSR PUTCHR
          INX
          CPX #32
          BNE _RELE

          LDX #0              ; Print song load address from header
          LDY #7
          JSR MOVESCRN
```

```
        LDX #MSG_LOAD
        JSR PUTMSG

        LDA SIDLOAD+1
        JSR PUTHEX
        LDA SIDLOAD+Ø
        JSR PUTHEX

        LDX #Ø                ; Print song init address from header
        LDY #8
        JSR MOVESCRN

        LDX #MSG_INIT
        JSR PUTMSG

        LDA SIDINIT+1
        JSR PUTHEX
        LDA SIDINIT+Ø
        JSR PUTHEX

        LDX #Ø                ; Print song play address from header
        LDY #9
        JSR MOVESCRN

        LDX #MSG_PLAY
        JSR PUTMSG

        LDA SIDPLAY+1
        JSR PUTHEX
        LDA SIDPLAY+Ø
        JSR PUTHEX

        LDX #Ø                ; Print song number in SID
        LDY #1Ø
        JSR MOVESCRN

        LDX #MSG_SONGNUM
        JSR PUTMSG

        LDA SONGNUM
        INC A
        JSR PUTHEX

        LDX #MSG_SONGOF
        JSR PUTMSG

        LDA SIDSNUM+Ø
        JSR PUTHEX

        RTS                   ; All done
```

*The **SHOWINFO** routine displays the song's header information.*

While the song is playing, the SID's registers are being updated constantly by the code in the SID file itself. To show the user what's going on, we periodically display the current contents of the SID registers. This is handled by the **SHOWREGS** routine, which displays the registers broken down by voice register bank and filter/volume register. This routine

is itself called from within the main loop to keep the screen up to date.

```
;
; SHOWREGS
;
; Displays the SID register values as hex numbers on the screen.
;
SHOWREGS  LDX #3                ; Print register column headings
          LDY #12
          JSR MOVESCRN

          LDX #MSG_REGS
          JSR PUTMSG

          LDX #0                ; Print voice 1 registers
          LDY #13
          JSR MOVESCRN

          LDX #MSG_V1
          JSR PUTMSG

          LDX #0
_V1       LDA SIDBASE+0,X
          JSR PUTHEX
          LDA #20
          JSR PUTCHR
          INX
          CPX #7
          BNE _V1

          LDX #0                ; Print voice 2 registers
          LDY #14
          JSR MOVESCRN

          LDX #MSG_V2
          JSR PUTMSG

          LDX #0
_V2       LDA SIDBASE+7,X
          JSR PUTHEX
          LDA #20
          JSR PUTCHR
          INX
          CPX #7
          BNE _V2

          LDX #0                ; Print voice 3 registers
          LDY #15
          JSR MOVESCRN

          LDX #MSG_V3
          JSR PUTMSG

          LDX #0
_V3       LDA SIDBASE+14,X
          JSR PUTHEX
          LDA #20
          JSR PUTCHR
          INX
          CPX #7
          BNE _V3

          LDX #27               ; Print filter and volume registers
          LDY #13
          JSR MOVESCRN

          LDX #0
_FV       LDA SIDBASE+21,X
          JSR PUTHEX
          LDA #20
          JSR PUTCHR
          INX
```

```
        CPX #4
        BNE _FV

        RTS
```

SHOWREGS *is responsible for displaying the current SID register values on the screen. This is a common feature in many SID players.*

Small helper routines are used to display other parts of the user interface. **SHOWMENU** displays the menu at the bottom of the main screen while **SHOWSCRN** clears the screen and prints the CodySID banner at the top.

```
;
; SHOWMENU
;
; Shows the menu text at the bottom of the screen.
;
SHOWMENU  LDX #0
          LDY #20
          JSR MOVESCRN

          LDX #MSG_MENU
          JSR PUTMSG
          RTS

;
; SHOWSCRN
;
; Shows the CodySID banner at the top of the screen.
;
SHOWSCRN  JSR CLRSCRN

          LDX #16
          LDY #0
          JSR MOVESCRN

          LDX #MSG_CODYSID
          JSR PUTMSG

          LDX #6
          LDY #1
          JSR MOVESCRN

          LDX #MSG_SUBTITLE
          JSR PUTMSG

          RTS
```

Helper routines for displaying a new CodySID player screen and the menu.

A total of three routines exist to handle communications over the UART. **UARTON** turns UART 1 on with a baud rate of 19200. **UARTGET** checks to see

if any data is in the receive buffer, and if so, removes it. If not, the routine returns immediately so that the program doesn't block. (Code using the routine can check if anything was read by looking at the 65C02's carry flag.) When the program is done reading a SID file, it calls **UARTOFF** to turn off UART 1. This code is conceptually similar to the UART code in the Cody BASIC interpreter as well as the UART examples written in BASIC in the previous chapter.

```
;
; UARTON
;
; Turns on UART 1.
;
UARTON    PHA
          PHY

_INIT     STZ UART1_RXTL          ; Clear out buffer registers
          STZ UART1_TXHD

          LDA #$0F                ; Set baud rate to 19200
          STA UART1_CNTL

          LDA #01                 ; Enable UART
          STA UART1_CMND

_WAIT     LDA UART1_STAT          ; Wait for UART to start up
          AND #$40
          BEQ _WAIT

          PLY
          PLA

          RTS                     ; All done

;
; UARTOFF
;
; Turns off UART 1.
;
UARTOFF   PHA

          STZ UART1_CMND          ; Clear bit to stop UART

_WAIT     LDA UART1_STAT          ; Wait for UART to stop
          AND #$40
          BNE _WAIT

          PLA

          RTS

;
; UARTGET
;
; Attempts to read a byte from the UART 1 buffer.
;
UARTGET   PHY

          LDA UART1_STAT          ; Test no error bits set in the status register
          BIT #$06
          BNE _ERR
```

```
        LDA UART1_RXTL          ; Compare current tail to current head position
        CMP UART1_RXHD
        BEQ _EMPTY

        TAY                     ; Read the next character from the buffer
        LDA UART1_RXBF,Y

        PHA                     ; Increment the receiver tail position
        INY
        TYA
        AND #$07
        STA UART1_RXTL
        PLA

        PLY
        SEC                     ; Set carry to indicate a character was read
        RTS

_EMPTY  PLY
        CLC                     ; Clear carry to indicate no character read
        RTS

_ERR    LDX #MSG_ERROR
        JSR PUTMSG

_DONE   JMP _DONE
```

UART routines used when a SID file is being loaded over the serial port.

Some additional utility routines are present to help with displaying content on the screen. **MOVESCRN** moves the current output location to a particular x and y coordinate on the screen, while **CLRSCRN** clears the screen entirely by filling the memory with whitespace characters.

```
;
; MOVESCRN
;
; Moves the SCRPTR to the position for the column/row in the X and Y
; registers. All registers are clobbered by the routine.
;
MOVESCRN  LDA #<SCRRAM          ; Move screen pointer to beginning
          STA SCRPTR+0
          LDA #>SCRRAM
          STA SCRPTR+1

          INY                   ; Increment pointer for each row
_LOOPY    CLC
          LDA SCRPTR+0
          ADC #40
          STA SCRPTR+0
          LDA SCRPTR+1
          ADC #0
          STA SCRPTR+1
          DEY
          BNE _LOOPY

          CLC                   ; Add position on column
          TXA
          ADC SCRPTR+0
          STA SCRPTR+0
```

```
          LDA SCRPTR+1
          ADC #Ø
          STA SCRPTR+1

          RTS

;
; CLRSCRN
;
; Clear the entire screen by filling it with whitespace (ASCII 2Ø decimal).
;
CLRSCRN   LDA #<SCRRAM            ; Move screen pointer to beginning
          STA SCRPTR+Ø
          LDA #>SCRRAM
          STA SCRPTR+1

          LDA #2Ø                 ; Clear screen by filling with whitespaces

          LDY #25                 ; Loop 25 times on Y

_LOOPY    LDX #4Ø                 ; Loop 4Ø times on X for each Y

_LOOPX    STA (SCRPTR)            ; Store zero

          INC SCRPTR+Ø            ; Increment screen position
          BNE _NEXT
          INC SCRPTR+1

_NEXT     DEX                     ; Next X
          BNE _LOOPX

          DEY                     ; Next Y
          BNE _LOOPY

          RTS
```

*The **MOVESCRN** and **CLRSCRN** routines set the current screen location or clear the screen entirely.*

Other utility routines include those for displaying content on the screen. **PUTMSG** prints a message string (defined by one of the **MSG_** constants) at the current location. **PUTCHR** puts a single character at the current location. **PUTHEX** is similar to **PUTCHR** but displays the current value as a two-digit hex number. All advance the screen location while printing.

```
;
; PUTMSG
;
; Puts a message string (one of the MSG_XXX constants) on the screen.
;
PUTMSG      PHA
            PHY

            LDA MSGS_L,X        ; Load the pointer for the string to print
            STA STRPTR+Ø
            LDA MSGS_H,X
            STA STRPTR+1

            LDY #Ø

_LOOP       LDA (STRPTR),Y      ; Read the next character (check for null)
```

```
            BEQ _DONE

            JSR PUTCHR          ; Copy the character and move to next
            INY

            BRA _LOOP           ; Next loop

_DONE       PLY
            PLA

            RTS

;
; PUTCHR
;
; Puts an individual ASCII character on the screen.
;
PUTCHR      STA (SCRPTR)        ; Copy the character

            INC SCRPTR+Ø        ; Increment screen position
            BNE _DONE
            INC SCRPTR+1

_DONE       RTS

;
; PUTHEX
;
; Puts a byte's hex value on the screen as two hex digits.
;
PUTHEX      PHA
            PHX
            TAX
            JSR HEXTOASCII
            PHA
            TXA
            LSR A
            LSR A
            LSR A
            LSR A
            JSR HEXTOASCII
            PHA
            PLA
            JSR PUTCHR
            PLA
            JSR PUTCHR
            PLX
            PLA
            RTS
HEXTOASCII  AND #$F
            CLC
            ADC #48
            CMP #58
            BCC _DONE
            ADC #6
_DONE       RTS
```

Utility routines for putting strings and hex numbers on the screen.

The messages that can be displayed on the screen are defined by set of constants. Each is prefixed with **MSG_** and relates to a particular location in the program's message table.

```
;
; IDs for the message strings that can be displayed in the program.
;
MSG_CODYSID   = 0
MSG_SUBTITLE  = 1
MSG_LOAD      = 2
MSG_INIT      = 3
MSG_PLAY      = 4
MSG_REGS      = 5
MSG_V1        = 6
MSG_V2        = 7
MSG_V3        = 8
MSG_MENU      = 9
MSG_RECEIVE   = 10
MSG_SONGNUM   = 11
MSG_SONGOF    = 12
MSG_ERROR     = 13
```

The messages that may be displayed by the CodySID program.

The string themselves are defined just below as null-terminated C strings.

```
;
; The strings displayed by the program.
;
STR_CODYSID   .NULL "CodySID!"
STR_SUBTITLE  .NULL "The Cody Computer SID Player"
STR_LOAD      .NULL "Load $"
STR_INIT      .NULL "Init $"
STR_PLAY      .NULL "Play $"
STR_REGS      .NULL "FL FH PL PH CL AD SR    CL CH FR MV"
STR_V1        .NULL "V1 "
STR_V2        .NULL "V2 "
STR_V3        .NULL "V3 "
STR_MENU      .NULL "(L)oad (Q)uit (P)rev (N)ext"
STR_RECEIVE   .NULL "Send PSID V2 file and wait for end..."
STR_SONGNUM   .NULL "Song $"
STR_SONGOF    .NULL " of $"
STR_ERROR     .NULL "ERROR!"
```

The actual strings corresponding to each message ID.

To map the constants to the strings, the strings' addresses are kept in a table of low bytes and high bytes. Each constant represents an index into the table. When a particular string is needed it's easy for the **PUTMSG** routine to find the string pointer based on the index within the table.

Splitting the table into low and high bytes is a common trick in 8-bit code. The program can use the

same index register value to look up both bytes without any other incrementing.

```
;
; Low bytes of the string table addresses.
;
MSGS_L
  .BYTE <STR_CODYSID
  .BYTE <STR_SUBTITLE
  .BYTE <STR_LOAD
  .BYTE <STR_INIT
  .BYTE <STR_PLAY
  .BYTE <STR_REGS
  .BYTE <STR_V1
  .BYTE <STR_V2
  .BYTE <STR_V3
  .BYTE <STR_MENU
  .BYTE <STR_RECEIVE
  .BYTE <STR_SONGNUM
  .BYTE <STR_SONGOF
  .BYTE <STR_ERROR

;
; High bytes of the string table addresses.
;
MSGS_H
  .BYTE >STR_CODYSID
  .BYTE >STR_SUBTITLE
  .BYTE >STR_LOAD
  .BYTE >STR_INIT
  .BYTE >STR_PLAY
  .BYTE >STR_REGS
  .BYTE >STR_V1
  .BYTE >STR_V2
  .BYTE >STR_V3
  .BYTE >STR_MENU
  .BYTE >STR_RECEIVE
  .BYTE >STR_SONGNUM
  .BYTE >STR_SONGOF
  .BYTE >STR_ERROR
```

The low-byte and high-byte portions of the message table.

The program's source code is ended with some boilerplate. The **LAST** label is used to indicate the end of the program. This is used when calculating the program length and end address for the program header, as you may remember from the beginning of the walkthrough. The **.ENDLOGICAL** assembly directive ends the **.LOGICAL** directive used at the beginning of the program to emit code for a particular load address.

```
LAST                            ; End of the entire program
.ENDLOGICAL
```

Boilerplate at the end of the program.

BUILDING AND RUNNING CODYSID

Building CodySID with *tass64* is straightforward. You only need the **codysid.asm** file and your installed *tass64* assembler. Just run the same command as in the previous example, but for CodySID: **64tass --mw65c02 --nostart -o codysid.bin codysid.asm**.

```
% 64tass --mw65c02 --nostart -o codysid.bin cody

64tass Turbo Assembler Macro V1.59.3120
64TASS comes with ABSOLUTELY NO WARRANTY; This i
are welcome to redistribute it under certain con

Assembling file:   codysid.asm
Output file:       codysid.bin
Data:       1126   $0000-$0465   $0466
Passes:            2
```

Assembling CodySID into a binary file.

Once you have the binary, you can load it from the Cody Computer like any other. Run **LOAD 1,1** to begin a load operation from the Prop Plug, then send the newly-generated binary over as you did in the previous example.

Once the program has started, it will prompt you to send a SID file over. You can send this from your terminal program just like you did the program itself. When the SID file has been received, the player will

automatically begin playing the first song in the SID. The screen contents will update with the current song and SID register information as the song is played. (If the SID is incompatible, however, anything could happen and you may have to restart the Cody Computer.)

You can use the on-screen options to load a different file, quit the program, or go back and forth to the previous or next song in the file (if any). Just press the key on your keyboard corresponding to the menu option.

The CodySID program playing a SID file of AC/DC's Highway to Hell. Note how the current SID register values are updated as the song plays.

SUGGESTED SID FILES

The *High-Voltage Sid Collection* contains the largest single repository of SID files. Many, but not all, of these can be used on the Cody Computer. During

development a subset of these were found to work reasonably well and were used for testing. A list of many of these high-quality known working files is given below.

- *Agent USA* by Tom Snyder (1984).
- *Axel F* by Barry Leitch (1986).
- The *Blackadder Theme* by Joachim Wijnhoven (2002).
- The *Blues Brothers* soundtrack by Paul Tankard (1991) contains multiple songs. It clobbers the screen memory but is otherwise playable.
- *Ducktales* by Vincent Voois (1990).
- *Electricity* by Pawel Wieczorek (1994).
- *Ghostbusters* by Etienne Muson (1985).
- *Highway to Hell* by Benjamin Dibbert (2022).
- *Jingle Bells* by Richard Bayliss (2002).
- The *Mayhem in Monsterland* soundtrack by Steve Rowlands (1993) contains multiple songs and sound effects.
- *The Mohican in the Gael* by Zack Maxis (2024).
- The *Murder on the Mississippi* soundtrack by Ed Bogas (1986) contains over a dozen brief songs.
- *Popcorn* by Sami Sepp (1980).
- *Radioactivity* by Sami Louko (2022).
- *The Railroad Works* by John Wentworth (1984) plays correctly but clobbers the default character set. Restart the computer after playing.
- *Seahorses* by Ed Bogas (1984) contains multiple songs and sound effects from *Sea Horse Hide'n Seek*.
- *Starman* by Sami Sepp (2015).

- *Star Trek - The Rebel Universe* by David Dunn (1989) is a rendition of the TV theme for the game of the same name.
- *Summer Games* (1984) from Epyx contains the national anthems and event songs from the game.
- *Take My Breath Away* by Steven Diemer (1991).

THE "CODY BROS." DEMO

Games are often written in assembly language because of its better performance. This is particularly the case for any kind of game with fast action such as arcade games. We won't be writing an entire game in this section, but we are going to write a simple demo reminiscent of *Super Mario Brothers*, *Great Giana Sisters*, and other platform games. It's a good oppportunity to show how some of the Cody Computer's features can be used together to make a game in assembly language.

We'll keep the game and its graphics simple so we don't need other tools to make it, instead just writing the relevant data as constants and tables in a simple assembly language program. To keep things very simple, our game will have a game world that is 64 tiles wide by 25 tiles high. We'll also only have a handful of tile types and only a single sprite.

All control will occur by reading the joystick periodically. When moving around in the game, the world willscroll horizontally from side to side. The player will have a single sprite under their control, and we'll be able to move the sprite left and right. Moving up on the joystick will produce a simple animation and

sound effect, while pulling down on the joystick will change the sprite's color. The fire button will exit the game and return to Cody BASIC.

Because it's a computer named after a dog, our sprite will be a stylized Pomeranian. And because the demo is inspired by a particular Nintendo classic, we'll have his outfits be red or green. Lastly, for an animation and sound effect, we'll make him bark rather than jump or shoot fireballs. Once you've mastered the basics, there's no reason you can't use what you learn here to make a real game.

THE CODYBROS PROGRAM

As with the CodySID player, the program starts with a variety of constant definitions and memory locations that we'll be using throughout the program. Some of these relate to the memory locations used for double-buffering of graphics. Because it's not possible to redraw an entire screen during the interval between frames, we have to render the next screen to another buffer. When the drawing is done, we switch them out between frames. This means that unlike many programs, we have two different screen memory and color memory locations.

```
ADDR      = $0300               ; The actual loading address of the program

SCRRAM1   = $A000               ; Screen memory locations for double-buffering
SCRRAM2   = $A400

COLRAM1   = $A800               ; Color memory locations for double-buffering
COLRAM2   = $AC00

SPRITES   = $B000               ; Sprite memory locations
```

Some of the most important memory locations we'll be using. This includes the double-buffers for the screen and color memory.

We'll be reading from the joystick, so the constants for the 65C22 VIA addresses are also included.

```
VIA_BASE  = $9F00               ; VIA base address and register locations
VIA_IORB  = VIA_BASE+$0
VIA_IORA  = VIA_BASE+$1
VIA_DDRB  = VIA_BASE+$2
VIA_DDRA  = VIA_BASE+$3
VIA_T1CL  = VIA_BASE+$4
VIA_T1CH  = VIA_BASE+$5
VIA_SR    = VIA_BASE+$A
VIA_ACR   = VIA_BASE+$B
VIA_PCR   = VIA_BASE+$C
VIA_IFR   = VIA_BASE+$D
VIA_IER   = VIA_BASE+$E
```

The memory locations for the 65C22 VIA's registers.

The program will need to read and update several video register locations, so those also need to be included somewhere in the program. Just like for the others, we'll define constants instead of using magic numbers.

```
VID_BLNK  = $D000               ; Video blanking status register
VID_CNTL  = $D001               ; Video control register
VID_COLR  = $D002               ; Video color register
VID_BPTR  = $D003               ; Video base pointer register
VID_SCRL  = $D004               ; Video scroll register
VID_SCRC  = $D005               ; Video screen common colors register
VID_SPRC  = $D006               ; Video sprite control register
```

Memory locations for the registers in the Cody Computer's video interface device.

We'll only have a single sprite in our program, and we'll place it at the beginning of the first sprite bank. This keeps the number of constants we need to define to a minimum.

```
SPR0_X    = $D000               ; Sprite X coordinate
SPR0_Y    = $D001               ; Sprite Y coordinate
SPR0_COL  = $D002               ; Sprite color
SPR0_PTR  = $D003               ; Sprite base pointer
```

The sprite registers used in the demo. There are many more for other sprites, but we're only using the first sprite in the first sprite bank.

The game won't have music, but it will have a sound effect. That means we'll need to know where the SID registers are in memory. In particular, we'll be using voice 1 for our sound effect, so we'll need those registers, along with a control register for setting the global volume. The SID, of course, has two other voices that we won't be using.

```
SID_BASE  = $D400               ; SID registers (mostly for voice 1)
SID_V1FL  = SID_BASE+0
SID_V1FH  = SID_BASE+1
SID_V1PL  = SID_BASE+2
SID_V1PH  = SID_BASE+3
SID_V1CT  = SID_BASE+4
SID_V1AD  = SID_BASE+5
SID_V1SR  = SID_BASE+6
SID_FVOL  = SID_BASE+24
```

The SID registers we'll be using in the program. The focus is on voice 1, which we'll use for a bark-like sound effect.

We'll also need to track the player's x and y coordinates along with the corner x and y position on the map. The player's y coordinate won't be used much for our demo, but the x coordinate is needed to determine where the player is on the screen. Because the player can move in per-pixel increments but the

tile map is along character boundaries, we'll have to convert back and forth at times in the program.

In our simple demo, the player can move up to 256 pixels because the x-coordinate is stored in a single byte. This is also the reason our game world is limited to 64 horizontal tiles (recall that each character on the screen is four pixels wide). In a real game you would probably want to have a larger game world, so you would either need to use a 16-bit number or keep track of per-character offsets in a separate variable.

```
PLAYERX   = $D0               ; Player coordinates
PLAYERY   = $D1

CORNERX   = $D2               ; Screen top-left corner coordinates
CORNERY   = $D3
```

Variables in zero-page used for the player's location and corners.

When we draw the game screen we'll need pointers to the game map and to the video device's screen and color memory. These will be typical 16-bit variables like you've already seen in other assembly programs.

```
MAPPTR    = $D4               ; Memory pointers for drawing the screen
SCRPTR    = $D6
COLPTR    = $D8
```

Pointer variables used when drawing the game screen.

We also have a few remaining flag variables. One tells us which of the two screen and color memory buffers to use, as we'll need to toggle between them on each frame. Another tells us whether the game sprite is moving forward or backward in the game world. We'll also need a temporary variable for some of our calculations, so it's declared here as well.

```
BUFFLAG   = $DA                ; Flag indicating what buffer is being used
FWDREV    = $DB                ; Flag indicating player direction (forward or reverse)

TEMP      = $DC                ; Temporary variable
```

Miscellaneous zero-page variables used by the program.

After our definitions are in place, we start with the beginning of the program. This program header is the same as in the other assembly language example. We also use the same assembly directive as before to generate our code relative to the program's load address.

```
; Program header for Cody Basic's loader (needs to be first)

.WORD ADDR                     ; Starting address (just like KIM-1, Commodore, etc.)
.WORD (ADDR + LAST - MAIN - 1)  ; Ending address (so we know when we're done loading)

; The actual program goes below here

.LOGICAL    ADDR               ; The actual program gets loaded at ADDR
```

The program header containing the start and end addresses of the program. Cody BASIC's program loader needs this information to be able to load and run the program.

Immediately after the program header is the start of the program, in our case a **MAIN** routine. It begins by setting up some of the variables in the game world, along with configuring the SID, VID, and VIA peripherals.

```
;
; MAIN
;
; The starting point of the demo. Performs the necessary setup before the demo runs.
;
MAIN        STZ PLAYERX         ; Reset player position
            LDA #183
            STA PLAYERY

            STZ FWDREV          ; Player moving forward by default

            STZ BUFFLAG         ; Clear double buffer flag

            LDA #%00000111      ; Set VIA data direction register A to 00000111 (pins 0-2 outp
            STA VIA_DDRA

            LDA #%00000110      ; Set VIA to read joystick 1
            STA VIA_IORA

            LDA #%00000001      ; Sprite bank 0, white as common color
            STA VID_SPRC

            LDA VID_COLR        ; Set border color to black
            AND #%11110000
            STA VID_COLR

            LDA #%11100000      ; Store shared colors (light blue and black)
            STA VID_SCRC

            LDA #%00000100      ; Enable horizontal scrolling
            STA VID_CNTL
```

*Initial setup in the **MAIN** routine.*

After the initial setup is done the program needs to populate the game world. Part of that involves copying the sprite data for our sprite into locations in sprite memory. It also has to copy a set of characters into character memory, as these characters are the custom tiles that make up the game world itself. (For our example we'll just copy them into the beginning of the normal character memory location, but in your own games, you could even move the character memory itself to a different location.)

```
            LDX #0              ; Copy game map tiles into character memory
_COPYCHAR   LDA CHARDATA,X
            STA $C800,X
            INX
            CPX #80
            BNE _COPYCHAR

            LDX #0              ; Copy sprite data into video memory
_COPYSPRT   LDA SPRITEDATA,X
            STA SPRITES,X
            INX
            CPX #255
            BNE _COPYSPRT

            LDA #$D8            ; Initial sprite color
            STA SPR0_COL
```

Setting up the characters (game tiles) and sprites for the demo.

At this point the program enters the game loop. On each loop we have to convert the player's location to the screen coordinates, draw the screen, and then handle any user input via the joystick. Some of the details are handled by subroutines, but the main loop organizes most of it.

The first part of the main loop calculates the screen location, taking into account the bounds of the game world. Ordinarly we want the game world centered on the player's current location, but at the beginning and end, we need to do a special check instead. We don't want the player to be able to move outside of the game world.

Once that's taken care of, the program calls **DRAWSCRN** to draw the screen for this frame. As part of drawing the screen, the program waits for a vertical blank to update the video registers before returning. As soon as it returns, the program calls **DRAWSPRT** to update the sprite in its correct location while the vertical blank is still occurring.

```
LOOP        LDA PLAYERX         ; Calculate coarse scroll position
            LSR A
            LSR A

            CMP #21
            BCC _TOOLO

            CMP #46
            BCS _TOOHI

            SEC
            SBC #21
            STA CORNERX

            BRA _DRAW

_TOOLO      STZ CORNERX
            BRA _DRAW

_TOOHI      LDA #25
            STA CORNERX
            BRA _DRAW

_DRAW       JSR DRAWSCRN        ; Draw the screen and sprite
            JSR DRAWSPRT
```

Code for calculating the current frame's coordinates before drawing it.

The rest of the main loop processes the joystick input. It reads VIA port A and then checks the bits to see if any buttons or switches are pressed. The fire button will exit the program, while right and left joystick movements move the player one pixel for that frame. Pushing the joystick up calls **BARK**, which displays a simple animation and sound effect. Pushing the joystick down calls **SWAPCOLOR**, which toggles the sprite's clothing color between green and red.

```
            LDA VIA_IORA        ; Read joystick
            LSR A
            LSR A
            LSR A

            BIT #16             ; Fire button?
            BEQ _FIRE

            BIT #8              ; Joystick right?
            BEQ _RIGHT

            BIT #4              ; Joystick left?
            BEQ _LEFT

            BIT #2              ; Joystick down to swap colors?
            BEQ SWAPCOLOR

            BIT #1              ; Joystick up to bark?
```

```
            BEQ BARK

            BRA LOOP

_FIRE       RTS                 ; Exit on fire button

_LEFT       LDA #1              ; Move left
            STA FWDREV

            LDA PLAYERX
            BEQ _NEXT

            DEC PLAYERX
            BRA _NEXT

_RIGHT      STZ FWDREV          ; Move right

            LDA PLAYERX
            CMP #232
            BEQ _NEXT

            INC PLAYERX

_NEXT       JMP LOOP
```

*The final portion of the **MAIN** routine. This code handles the user input from the joystick and fire button.*

The **BARK** routine handles the sound and animation when the player moves the joystick up. It starts by configuring the SID to play a sawtooth wave, then enters an inner loop, **_WOOF**. In the **_WOOF** loop, the program increases the frequency of the sound slightly while moving the sprite upward on the screen. At the end the sound is shut off and the sprite moved back to its normal y-coordinate.

```
;
; BARK
;
; Handles a barking sound/animation for the sprite, then jumps back to the
; main loop.
;
BARK        LDA #$0F            ; Set main volume
            STA SID_FVOL

            LDA #<2400          ; Set starting frequency
            STA SID_V1FL
            LDA #>2400
            STA SID_V1FH

            LDA #$50            ; Attack/decay
            STA SID_V1AD

            LDA #$F0            ; Sustain/release
            STA SID_V1SR

            LDA #$21            ; Begin playing
            STA SID_V1CT
```

```
            LDX #Ø              ; Loop counter

_WOOF       JSR WAITBLANK       ; Wait for the next frame

            DEC SPRØ_Y          ; Decrement sprite Y for dog hop

            CLC                 ; Increment frequency for next loop
            LDA SID_V1FL
            ADC #1ØØ
            STA SID_V1FL

            LDA SID_V1FH
            ADC #Ø
            STA SID_V1FH

            INX                 ; Increment for next loop
            CPX #1Ø
            BNE _WOOF

            LDA #Ø              ; Stop playing
            STA SID_V1CT

            LDA PLAYERY         ; Move sprite back to original y
            STA SPRØ_Y

            JMP LOOP
```

*The **BARK** routine makes a bark-like sound while moving the game sprite up and down quickly. As a first approximation, it simulates a barky agitated or excited Pomeranian.*

The other player action (other than movement) is handled by **SWAPCOLOR**. Those of you who have played the original *Super Mario Brothers* may have noted that Mario and Luigi were basically the same sprite, just with red or green colors. Our demo does a similar thing, with the player sprite starting out green. When toggled, we switch out the sprite's color register so that the green color is red. And when toggled again, it switches back to green, and so on.

```
;
; SWAPCOLOR
;
; Swaps the sprite color (red/green or green/red) and jumps back to the main
; loop.
;
SWAPCOLOR   LDA SPRØ_COL        ; Check current sprite colors
            CMP #$D8
            BEQ _RED

_GRN        LDA #$D8            ; Make sprite wear green
            STA SPRØ_COL
            BRA _WAITJOY

_RED        LDA #$28            ; Make sprite wear red
            STA SPRØ_COL
            BRA _WAITJOY

_WAITJOY    LDA VIA_IORA        ; Read joystick
            LSR A
            LSR A
            LSR A

            BIT #2              ; Wait for joystick release
            BEQ _WAITJOY

            JMP LOOP            ; All done
```

***SWAPCOLOR** toggles the player sprite between green and red.*

Drawing the screen is handled by the **DRAWSCRN** routine. It sets up a pointer into the map data, then iterates over the data to populate the screen and color memory for the next frame. Because it takes so long to draw a screen, all the drawing is done offscreen in a technique known as double-buffering. At the end, the routine waits for a vertical blank, then switches the video registers to point to the new screen and color memory areas. We flip back and forth between them on each call to **DRAWSCRN** so one is being shown while the other is being drawn.

This isn't quite how the drawing would be done in a real game. In a real game, the screen would only be fully updated every fourth frame. The scroll registers would be used to slowly slide the current screen across while the new screen is being drawn (roughly one-quarter of it on each frame). When the scroll

wraps around, the new screen would be ready and swapped in.

That approach is more complex but it allows a better frame rate than our demo. What we have here is intended to be an example of double-buffering without additional complications. It does mean that we're doing extra work redrawing the entire screen on each call, but the result is suitable to show the basics. Just be aware that there are better ways of doing this in real life.

Much of the drawing (or more accurately, copying) is done in the **COPYROWS** routine. It takes a single parameter in the X register, the number of rows to copy. This is because, again, in a real application only a subset of screen rows may be copied between frames (rather than slowing down the whole application to draw the whole thing each time). We just use a value of 25 to draw all the rows.

```
;
; DRAWSCRN
;
; Draws the current visible of the screen. This routine uses double-buffering
; so that the new screen and colors are drawn to a different location, and the
; screens/colors are switched out during the vertical blanking interval.
;
; In a real application the screen may need to be drawn (offscreen) in sections
; to keep up with a high game frame rate. For an example this works well enough
; to avoid glitches or tearing during scrolling.
;
DRAWSCRN    LDA #<MAPDATA       ; Start map pointer at beginning of map
            STA MAPPTR+0
            LDA #>MAPDATA
            STA MAPPTR+1

            CLC                 ; Adjust map position based on player position
            LDA MAPPTR+0
            ADC CORNERX
            STA MAPPTR+0
            LDA MAPPTR+1
            ADC #0
            STA MAPPTR+1

            LDA BUFFLAG         ; Determine what buffer to draw to
            TAX

            LDA SCRRAMS_L,X     ; Start screen pointer at beginning of buffer
            STA SCRPTR+0
            LDA SCRRAMS_H,X
            STA SCRPTR+1
```

```
            LDA COLRAMS_L,X     ; Start color pointer at beginning of buffer
            STA COLPTR+Ø
            LDA COLRAMS_H,X
            STA COLPTR+1

            LDX #25             ; For now, try drawing everything
            JSR COPYROWS

            JSR WAITBLANK       ; Wait for the blanking interval to make changes

            LDA BUFFLAG         ; Determine what buffer to flip to
            TAX

            LDA BASEREGS,X      ; Update base register for screen memory
            STA VID_BPTR

            LDA COLREGS,X       ; Update color register for color memory
            STA VID_COLR

            LDA BUFFLAG         ; Toggle buffer flag
            EOR #$Ø1
            STA BUFFLAG

            LDA PLAYERX         ; Update fine scroll position if needed

            CMP #(4*21)
            BCC _DONE

            CMP #(4*46)
            BCS _DONE

            AND #$Ø3
            ASL A
            ASL A
            ASL A
            ASL A
            STA VID_SCRL

_DONE       RTS                 ; All done
```

DRAWSCRN *handles most of the high-level operations involved in rendering a new screen and color memory area based on the current map location.*

The screen and color memory is updated by the **COPYROWS** routine. As mentioned, it will update a variable number of rows on each call, specified by the value in the X register. It also assumes that the **MAPPTR** is pointed to the current source row in the map data, while **SCRPTR** and **COLPTR** point to the current destination rows in scren and color memory.

Screen data is copied directly from the map data. Color data is obtained by using the tile value as an index into a lookup table, **COLORDATA**, that has the character-specific colors for each tile. (For many games this technique is actually not that optimal, as

tiles may be drawn in a variety of colors, but for this example it works nicely.)

Each row consists of 40 characters written to the screen and color memory locations. Index registers are used to reference particular memory locations relative to the pointers, but after each row, they need to be updated to move to the next row. For **COLPTR** and **SCRPTR** they need to be incremented by 40 because screen and color memory are 40 characters wide. For **MAPDATA** the pointer needs to be incremented by 64 because the game world is 64 tiles wide.

```
;
; COPYROWS
;
; Copies a number of rows from the game map into the screen and color memory. The
; number of rows to copy is stored in the X register.
;
COPYROWS

_XLOOP      PHX
            LDY #0

_YLOOP      LDA (MAPPTR),Y      ; Copy the character (game tile) into screen memory
            STA (SCRPTR),Y

            TAX                 ; Copy the color into color memory
            LDA COLORDATA,X
            STA (COLPTR),Y

            INY                 ; Next loop for Y
            CPY #40
            BNE _YLOOP

            CLC                 ; Increment map pointer to next row
            LDA MAPPTR+0
            ADC #64
            STA MAPPTR+0
            LDA MAPPTR+1
            ADC #0
            STA MAPPTR+1

            CLC                 ; Increment screen pointer to next row
            LDA SCRPTR+0
            ADC #40
            STA SCRPTR+0
            LDA SCRPTR+1
            ADC #0
            STA SCRPTR+1

            CLC                 ; Increment color pointer to next row
            LDA COLPTR+0
            ADC #40
            STA COLPTR+0
            LDA COLPTR+1
            ADC #0
            STA COLPTR+1

            PLX                 ; Next loop for X
            DEX
            BNE _XLOOP
```

```
          RTS               ; All done
```

*The **COPYROWS** routine updates a certain number of rows in a screen and color memory location with the data from the game map.*

The sprite also needs to be updated on each frame. This is handled by the **DRAWSPRT** routine. It looks at the current player position in the game world and determines where the sprite should be drawn on the screen. In most situations the sprite should be drawn in the middle of the screen, but at the beginning and end of the game world the behavior is different. In those cases, scrolling stops, so the sprite has to move instead.

Our sprite also has a total of four frames, two walking forward and two walking backward. To specify the correct sprite image, the program examines the value in **FWDREV** set by the main loop to determine whether the player's moving forward (right) or backward (left). Once that's decided, the current player X coordinate is used to pick one of the two walk frames for each direction. Even values use one sprite and odd ones the other.

This routine gets called immediately after **DRAWSCRN** because we want to make the sprite register updates during the vertical blank as well. When drawing the screen the program waits until a vertical blank to update the video registers, and so calling this immediately after means the code can run in the same vertical blank.

```
;
; DRAWSPRT
;
; Draws the sprite in the correct location for this frame. Note that the sprite
; isn't "drawn" so much as its registers updated so that it appears correctly.
```

```
; This should be called after drawing the screen because we want to sneak in
; during the vertical blank.
;
DRAWSPRT    LDA PLAYERX         ; Calculate new sprite location
            CMP #(21*4)
            BCC _LO

            CMP #(46*4)
            BCS _HI

            LDA #(21*4)
            BRA _SPRX

_LO         BRA _SPRX

_HI         SEC
            SBC #((46*4)-84)
            BRA _SPRX

_SPRX       ADC #12             ; Update sprite X
            STA SPR0_X

            LDA PLAYERY         ; Update sprite Y
            STA SPR0_Y

            LDA FWDREV          ; Update sprite base pointer (different frames)
            ASL A
            STA TEMP
            CLC
            LDA PLAYERX
            AND #$02
            LSR A
            ADC TEMP
            ADC #(4096/64)
            STA SPR0_PTR

            RTS
```

DRAWSPRT *updates the sprite on the screen based on the current game state.*

WAITBLANK handles the actual waiting for a vertical blank. First it waits for the blanking register to have a zero value, indicating that the screen is actively being displayed by the video hardware. After detecting a zero, it waits for a transition to a 1, meaning that we went from drawing to the blanking interval. Just checking for a 1 won't do as we might be in the middle or at the end of the interval, which isn't necessarily what we want.

The Commodore 64, like many computers of its day, had an interrupt that would fire on particular screen lines. That could be used to handle this in an interrupt rather than having to poll for a changed value. Many other computers, including the Commodore VIC-20, didn't have such an interrupt, so polling was the only option. The Cody Computer falls into this latter category.

```
;
; WAITBLANK
;
; Waits for the vertical blank signal to transition from drawing to not drawing, then
; returns. Used to sync up screen/register updates so they don't occur in the middle
; of the screen.
;
WAITBLANK

_WAITVIS    LDA VID_BLNK        ; Wait until the blanking is zero (drawing the screen)
            BNE _WAITVIS

_WAITBLANK  LDA VID_BLNK        ; Wait until the blanking is one (not drawing the screen)
            BEQ _WAITBLANK

            RTS
```

*The **WAITBLANK** routine waits for a transition between drawing the visible screen (0) and blanking (1). Code that updates video registers should run in the blanking interval if possible.*

The game map is defined in **MAPDATA**, a sequence of 25 rows of 64 bytes. This is the source for drawing the screen, and each byte represents a particular tile type. In real games, some kind of map editor is usually used to make the game map. The data is exported to an assembly file to include in your program. In earlier times, the game map may have actually been designed on graph paper before such tools were

common. For a simple example like this, we can just pop numbers into the program as follows.

```
;
; The game map.
;
; 0 = Sky
; 1 = Brick
; 2 = Cloud left
; 3 = Cloud middle
; 4 = Cloud right
; 5 = Hills left
; 6 = Hills middle
; 7 = Hills right
; 8 = ?
; 9 = ?
;
MAPDATA

  .BYTE 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,3,4,0,0,0,0,0,0,0
  .BYTE 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,3,3,3,3,3,4,0,0,0
  .BYTE 0,0,0,2,3,3,4,0,0,0,0,0,0,0,0,0,0,0,2,3,3,3,3,3,4,0,0,0,0,0,2,3,3,4,0,0,0,0,0,0
  .BYTE 0,0,2,3,3,3,3,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
  .BYTE 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,3,
  .BYTE 0,0,0,0,0,0,0,0,0,0,0,0,2,3,3,3,3,3,4,0,0,0,0,0,0,0,0,0,2,3,4,0,0,0,0,2,3,3,3,3
  .BYTE 0,0,0,0,0,0,0,0,0,0,0,0,0,2,3,3,3,3,3,3,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,3,3
  .BYTE 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,3,4,0,0,0,0,0,0,0
  .BYTE 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
  .BYTE 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
  .BYTE 0,0,0,0,2,3,3,4,0,0,0,0,0,0,0,0,0,0,0,2,3,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
  .BYTE 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
  .BYTE 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,1,1,0,0,0,0
  .BYTE 0,0,0,0,0,0,0,0,0,0,0,0,2,3,4,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,1,1,0,0,0,0,0,0
  .BYTE 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,1,1,0,0,0,0,0,0
  .BYTE 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,1,1,0,0,0,0,0,0
  .BYTE 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1
  .BYTE 0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1
  .BYTE 0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,0,5,6,7,0,0,0,0,0,0,0,0,0
  .BYTE 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,5,6,6,6,7,0,0,0,0,0,0,0,0,0,0,0
  .BYTE 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,5,6,6,6,6,6,7,0,0,0,0,0,0,0,0,0,0
  .BYTE 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,5,6,6,6,6,6,6,6,7,0,0,0,0,0,0,0,0,5
  .BYTE 0,0,0,0,0,5,7,0,0,0,0,0,0,0,0,0,0,5,6,6,6,7,5,6,6,6,6,6,6,6,6,7,0,0,0,0,0,0,5,6
  .BYTE 0,0,0,0,5,6,6,7,0,0,0,0,0,0,0,0,5,6,6,6,6,6,6,6,6,6,6,6,6,6,6,7,0,0,0,0,0,5,6,6
  .BYTE 0,0,0,5,6,6,6,6,7,0,0,0,0,0,0,0,5,6,6,6,6,6,6,6,6,6,6,6,6,6,6,7,0,0,0,0,5,6,6,6
  .BYTE 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
  .BYTE 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
```

MAPDATA *is a sequence of bytes that represent the game world.*

The tiles themselves are represented as characters. When the video hardware draws the screen, the "characters" it draws will actually be the game world's tiles. The **MAIN** routine copies these characters over the first 10 characters in the default character memory at startup. We can use them in the game just by putting the matching number into screen memory.

```
;
; The game's character tiles (used to draw the map).
```

```
;
CHARDATA

  .BYTE %11111111   ; Sky
  .BYTE %11111111
  .BYTE %11111111
  .BYTE %11111111
  .BYTE %11111111
  .BYTE %11111111
  .BYTE %11111111
  .BYTE %11111111

  .BYTE %01010101   ; Brick
  .BYTE %01000000
  .BYTE %01000000
  .BYTE %01000000
  .BYTE %01010101
  .BYTE %00000001
  .BYTE %00000001
  .BYTE %00000001

  .BYTE %11111100   ; Cloud left
  .BYTE %11000000
  .BYTE %00000000
  .BYTE %00000000
  .BYTE %00000000
  .BYTE %00000000
  .BYTE %11000000
  .BYTE %11111100

  .BYTE %00000000   ; Cloud middle
  .BYTE %00000000
  .BYTE %00000000
  .BYTE %00000000
  .BYTE %00000000
  .BYTE %00000000
  .BYTE %00000000
  .BYTE %00000000

  .BYTE %00111111   ; Cloud right
  .BYTE %00000011
  .BYTE %00000000
  .BYTE %00000000
  .BYTE %00000000
  .BYTE %00000000
  .BYTE %00000011
  .BYTE %00111111

  .BYTE %11111100   ; Hills left
  .BYTE %11111100
  .BYTE %11110001
  .BYTE %11110000
  .BYTE %11000100
  .BYTE %11000000
  .BYTE %00010000
  .BYTE %00000001

  .BYTE %00000000   ; Hills middle
  .BYTE %00010000
  .BYTE %00000000
  .BYTE %01000000
  .BYTE %00000100
  .BYTE %00000000
  .BYTE %01000000
  .BYTE %00000001

  .BYTE %00111111   ; Hills right
  .BYTE %00111111
  .BYTE %00001111
  .BYTE %01001111
  .BYTE %00000011
  .BYTE %00010011
  .BYTE %00000000
  .BYTE %01000100

  .BYTE %00000000   ; Unused
  .BYTE %00000000
```

```
  .BYTE %00000000
  .BYTE %00000000
  .BYTE %00000000
  .BYTE %00000000
  .BYTE %00000000
  .BYTE %00000000

  .BYTE %00000000   ; Unused
  .BYTE %00000000
  .BYTE %00000000
  .BYTE %00000000
  .BYTE %00000000
  .BYTE %00000000
  .BYTE %00000000
  .BYTE %00000000
```

*The **CHARDATA** for the game tiles. This is copied into the first 10 entries in character memory on startup.*

There is no connection between tiles and their colors. Color memory is separate from screen memory, and each tile could in theory be drawn in a variety of colors. For our demo, however, each tile only needs one particular set of colors. Rather than have an entire map just for colors, we can make a small lookup table to find the color memory value for each game tile. **COLORDATA** is exactly such a lookup table.

```
;
; The color date to copy for each tile type.
;
COLORDATA

  .BYTE   $00       ; Sky (no colors)
  .BYTE   $09       ; Brick (black and brown)
  .BYTE   $F1       ; Clouds (gray and white)
  .BYTE   $F1       ; Clouds (gray and white)
  .BYTE   $F1       ; Clouds (gray and white)
  .BYTE   $D5       ; Hills (light green and green)
  .BYTE   $D5       ; Hills (light green and green)
  .BYTE   $D5       ; Hills (light green and green)
  .BYTE   $00       ; Unused
  .BYTE   $00       ; Unused
```

***COLORDATA** contains the color memory value for each game tile.*

The last portion of data needed for the program is the data for the Pomeranian sprite the player can control on the screen. As mentioned earlier in the book, sprites are 12 pixels by 21 pixels in size and have a layout very similar to C64 multicolor sprites. Each

sprite fits in 63 bytes with one blank byte rounding up to an even 64 bytes.

For the demo we have a total of four sprites, two of the Pomeranian walking forward to the right and two of the Pomeranian walking backward to the left. This is a total of 256 bytes, all of which are copied to video memory and used as sprite graphics during the game. The actual copying is done by the **MAIN** routine with the sprite registers being updated on each call to **DRAWSPRT**.

```
;
; The sprite data for the Pomeranian sprite on the screen.
;
SPRITEDATA

  .BYTE %00000000,%00000001,%01000000   ; Pomeranian forward 0
  .BYTE %00010000,%00001101,%11110000
  .BYTE %00010000,%00001101,%01111111
  .BYTE %01010100,%00000101,%01010000
  .BYTE %01010100,%00110101,%01110000
  .BYTE %01010100,%10110101,%01010101
  .BYTE %01010100,%10111001,%01010111
  .BYTE %01010111,%10101110,%01010100
  .BYTE %01010111,%10101110,%01010000
  .BYTE %01010111,%10101110,%10100000
  .BYTE %00010110,%11101110,%10100000
  .BYTE %00011010,%11101110,%10100000
  .BYTE %00001010,%11101110,%10000000
  .BYTE %00001010,%10111010,%10000000
  .BYTE %00010110,%10111001,%01010000
  .BYTE %00010101,%01000001,%01010000
  .BYTE %01010101,%00000000,%01010000
  .BYTE %01010000,%00000000,%01010000
  .BYTE %01010000,%00000000,%01010000
  .BYTE %00010100,%00000000,%00010100
  .BYTE %00010100,%00000000,%00010100
  .BYTE %00000000

  .BYTE %00000000,%00000001,%01000000 ; Pomeranian forward 1
  .BYTE %00010000,%00001101,%11110000
  .BYTE %00010000,%00001101,%01111111
  .BYTE %01010100,%00000101,%01010000
  .BYTE %01010100,%00110101,%01110000
  .BYTE %01010100,%10110101,%01010101
  .BYTE %01010100,%10111001,%01010111
  .BYTE %01010111,%10101110,%01010100
  .BYTE %01010111,%10101110,%01010000
  .BYTE %01010111,%10101110,%10100000
  .BYTE %00010110,%11101110,%10100000
  .BYTE %00011010,%11101110,%10100000
  .BYTE %00001010,%11101110,%10000000
  .BYTE %00001010,%10111010,%10000000
  .BYTE %00000110,%10111001,%01000000
  .BYTE %00010101,%01000001,%01000000
  .BYTE %00010101,%00000101,%00000000
  .BYTE %00000101,%00000101,%00000000
  .BYTE %00010101,%00000101,%00000000
  .BYTE %01010100,%00000001,%01000000
  .BYTE %01010000,%00000001,%01000000
  .BYTE %00000000
```

```
 .BYTE %00000001,%01000000,%00000000   ; Pomeranian reverse 0
 .BYTE %00001111,%01110000,%00000100
 .BYTE %11111101,%01110000,%00000100
 .BYTE %00000101,%01010000,%00010101
 .BYTE %00001101,%01011100,%00010101
 .BYTE %01010101,%01011110,%00010101
 .BYTE %11010101,%01101110,%00010101
 .BYTE %00010101,%10111010,%11010101
 .BYTE %00000101,%10111010,%11010101
 .BYTE %00001010,%10111010,%11010101
 .BYTE %00001010,%10111011,%10010100
 .BYTE %00001010,%10111011,%10100100
 .BYTE %00000010,%10111011,%10100000
 .BYTE %00000010,%10101110,%10100000
 .BYTE %00000101,%01101110,%10010100
 .BYTE %00000101,%01000001,%01010100
 .BYTE %00000101,%00000000,%01010101
 .BYTE %00000101,%00000000,%00000101
 .BYTE %00000101,%00000000,%00000101
 .BYTE %00010100,%00000000,%00010100
 .BYTE %00010100,%00000000,%00010100
 .BYTE %00000000

 .BYTE %00000001,%01000000,%00000000   ; Pomeranian reverse 1
 .BYTE %00001111,%01110000,%00000100
 .BYTE %11111101,%01110000,%00000100
 .BYTE %00000101,%01010000,%00010101
 .BYTE %00001101,%01011100,%00010101
 .BYTE %01010101,%01011110,%00010101
 .BYTE %11010101,%01101110,%00010101
 .BYTE %00010101,%10111010,%11010101
 .BYTE %00000101,%10111010,%11010101
 .BYTE %00001010,%10111010,%11010101
 .BYTE %00001010,%10111011,%10010100
 .BYTE %00001010,%10111011,%10100100
 .BYTE %00000010,%10111011,%10100000
 .BYTE %00000010,%10101110,%10100000
 .BYTE %00000001,%01101110,%10010000
 .BYTE %00000001,%01000001,%01010100
 .BYTE %00000000,%01010000,%01010100
 .BYTE %00000000,%01010000,%01010000
 .BYTE %00000000,%01010000,%01010100
 .BYTE %00000001,%01000000,%00010101
 .BYTE %00000001,%01000000,%00000101
 .BYTE %00000000
```

SPRITEDATA *consists of four sprite graphics, two of a Pomeranian walking to the right and two of a Pomeranian walking to the left.*

The program ends with some lookup table used as part of double-buffering. We have two different screen/color memory buffers that need to be swapped in and out. To make it easy to do that, lookup tables contain the base addresses of each along with the corresponding register values needed to update them. When swapping, we can just read a value in the table corresponding to the **BUFFLAG** variable.

```
;
; Lookup tables for screen and color memory locations. Used to quickly
```

```
; switch between the double buffer during an update.
;
SCRRAMS_L

  .BYTE <SCRRAM1
  .BYTE <SCRRAM2

SCRRAMS_H

  .BYTE >SCRRAM1
  .BYTE >SCRRAM2

COLRAMS_L

  .BYTE <COLRAM1
  .BYTE <COLRAM2

COLRAMS_H

  .BYTE >COLRAM1
  .BYTE >COLRAM2

BASEREGS

  .BYTE $05
  .BYTE $15

COLREGS

  .BYTE $20
  .BYTE $30
```

Lookup tables used to simplify double-buffering operations.

The program itself ends as our CodySID music player example. We have a **LAST** label used to calculate the end address of the program. This is followed by an assembler directive closing the one our program started with.

```
LAST                            ; End of the entire program

.ENDLOGICAL
```

The same boilerplate at the end of the program.

BUILDING AND RUNNING CODY BROS.

You build and run the demo the same way as you did the CodySID music player. First you'll need to run the code through the *64tass* assembler on your PC.

Just run **64tass --mw65c02 --nostart -o codybros.bin codybros.asm** and check the output:

```
% 64tass --mw65c02 --nostart -o codybros.bin cod

64tass Turbo Assembler Macro V1.59.3120
64TASS comes with ABSOLUTELY NO WARRANTY; This i
are welcome to redistribute it under certain con

Assembling file:   codybros.asm
Output file:       codybros.bin
Data:       2448   $0000-$098f   $0990
Passes:            2
```

Building the codybros demo using the 64tass assembler.

Once you have the binary, you run **LOAD 1,1** on the Cody Computer and send the file over a serial link. The program will start up automatically. To use the program you'll need to have an Atari-compatible joystick to plug into joystick port 1. Moving the joystick left and right will move the player on the screen, moving the joystick up runs the "bark" animation, and moving the joystick down changes the sprite color. To return to Cody BASIC just press the fire button.

If you don't have an Atari-compatible joystick available, cheap ones are available online or at many retro electronics or video game stores in larger cities. The design is quite simple, so you can even find plans online to make your own: Unlike Nintendo controllers that required at least some logic chips, an Atari joystick is literally just switches wired to a connector.

If all else fails, you can also change the program to accept keyboard input rather than joystick input. In the main loop where the joystick row is read, change the

row to one of the rows on the keyboard matrix, then check for pressed keys instead of pressed switches on the joystick. Look up the keys you would need to press for that row and use those for the controls instead. (You'll need the keyboard schematic and perhaps the CodySID or input-output examples to help you in doing that.)

A Pomeranian sprite moving around in a very Mario-like or Giana-like game world. You can use something like this as a starting point for a full game.

MEMORY-RESIDENT PROGRAMS

The Cody Computer also allows you to write programs in assembly language that can be left in memory and called by Cody BASIC programs. This can be a powerful way to allow external devices and peripherals to copy some code into memory that can later be called from a **SYS** statement.

In this situation, you'll be writing an assembly language program that can be loaded into memory but later returns back to BASIC. The program can use the Cody BASIC binary loader to load it into the expected location in memory to start with, or you can have it load the binary and then copy the code yourself.

As part of the initialization of such a program, you might adjust the value of **PROGEND**, a zero-page variable at address **$004B** that specifies the page boundary of BASIC program memory. You can move the page down to steal memory for your program from BASIC, then load your BASIC program after the binary is finished loading. This is a particularly useful approach for writing assembly language programs that work like drivers or extensions of BASIC itself.

To show how this works, we'll go through a very short program that will let you change the color of the screen border. Once assembled, you will load the program using the **LOAD** command. The program will return to Cody BASIC after initialization, and you'll be able to call it using **SYS 25600** to run the routine.

To make our job easier, we'll assume the program is loaded at **$6300** (decimal 25344) with our initialization routine. Within our program we'll include the memory-resident routine at an offset of **$100** so it will reside at the easy to remember **$6400** or decimal 25600.

```
ADDR      = $6300               ; The actual loading address of the program

VID_BLNK  = $D000               ; Video blanking status register
VID_CNTL  = $D001               ; Video control register
VID_COLR  = $D002               ; Video color register
VID_BPTR  = $D003               ; Video base pointer register
VID_SCRL  = $D004               ; Video scroll register
VID_SCRC  = $D005               ; Video screen common colors register
VID_SPRC  = $D006               ; Video sprite control register

PROGEND   = $4B                 ; Boundary page for program memory
TEMP      = $FF                 ; Temporary variable in zero page
```

*Our program is loaded at a base address of **$6300**. It also uses some of the video registers and modifies locations in zero page.*

We use the same boilerplate and assembly directives to ensure the program starts at the location we expect.

```
; Program header for Cody Basic's loader (needs to be first)

.WORD ADDR                      ; Starting address (just like KIM-1, Commodore, etc.)
.WORD (ADDR + LAST - MAIN - 1)  ; Ending address (so we know when we're done loading)

; The actual program goes below here

.LOGICAL    ADDR                ; The actual program gets loaded at ADDR
```

Boilerplate for locating the start of the program in memory.

Our **MAIN** routine is quite small. It adjusts the **PROGEND** location so that future BASIC programs won't overwrite our memory-resident program and then returns back to Cody BASIC. We don't need to copy any code because we use the loader to put our program in the right place to begin with.

```
MAIN        LDA #>ADDR          ; Move the program memory bounds down
            STA PROGEND

            RTS
```

The short initialization routine called by Cody BASIC's loader.

The interesting code is in the **CYCLE** routine, which color-cycles the border color on the screen when called. We specify the location where this should be located and the **64tass** assembler emits the code at the proper location within our binary.

```
* = $6400                      ; Start address of the color-cycle routine

CYCLE       PHA                ; Preserve registers

            LDA VID_COLR       ; Increment the border color by one and store in temp
            INC A
            AND #$0F
            STA TEMP

            LDA VID_COLR       ; Combine the new value and the color register to update
            AND #$F0
            ORA TEMP
            STA VID_COLR

            PLA                ; Restore registers and return
            RTS
```

*The **CYCLE** routine is resident at address **$6400**.*

As with all other programs we have a bit of boilerplate at the end as well.

```
LAST                           ; End of the entire program

.ENDLOGICAL
```

The end of the program.

You can build this program like all the others. Run **64tass --mw65c02 --nostart -o resident.bin resident.asm** and check the output:

```
64tass Turbo Assembler Macro V1.59.3120
64TASS comes with ABSOLUTELY NO WARRANTY; This i
are welcome to redistribute it under certain con

Assembling file:   resident.asm
Output file:       resident.bin
Data:          9   $0000-$0008   $0009
Gap:         251   $0009-$0103   $00fb
Data:         21   $0104-$0118   $0015
Passes:            2
```

A successful build with 64tass.

Load the program over a serial link using **LOAD 1,1**. When the program is loaded and completed, you'll return back to Cody BASIC as though nothing had happened. In fact, it will look like the Cody Computer has just started up for the first time. However, when you call **SYS 25600** you should notice the screen's border color change. Try it several times to observe the results.

```
READY.
SYS 25600

READY.
SYS 25600

READY.
```

*Calling the resident program from Cody BASIC using the **SYS** statement.*

In some programs of this nature it might be better to introduce a jump table. In a jump table, each location actually contains a 65C02 **JMP** instruction that jumps to the actual implementation. The locations for each **JMP** instruction are hardcoded and do not change between releases, but the code they jump to can. This can be useful for drivers and libraries that may change internally but want to have a consistent external interface.

11

Cartridges and SPI

INTRODUCTION

The Cody Computer also supports cartridges that can be plugged into the expansion port. If a cartridge is detected, a binary program from the cartridge is loaded into memory and executed instead of booting to Cody BASIC. The program is contained inside the cartridge with a memory chip that supports the Serial Peripheral Interface (SPI) protocol, and certain pins on the expansion port are repurposed to implement SPI.

Cartridges are not necessary to use the Cody Computer. Assembly language programs can be loaded over a serial port just like Cody BASIC programs. Even if you plan not to use cartridges, examples in this chapter may be helpful if you plan to implement the SPI protocol with the Cody Computer.

SPI is probably the simplest data transfer protocol in common use. It's a three-wire protocol often used to communicate between microcontrollers and their peripherals. One line transmits data, one line receives data, and one line acts as a clock. A fourth line not involved in the actual communication acts as a chip select, telling a chip when an SPI data transaction is about to begin.

An SPI transaction begins by bringing the SPI chip select low. From there, data is clocked out on the output pin while data is read from the input pin, using the SPI clock pin for the clock signal. One or more bytes are transferred in this way. Often a command of some kind is clocked out first, with subsequent clocks used to read in the result of the command. The exact behavior depends on the device itself.

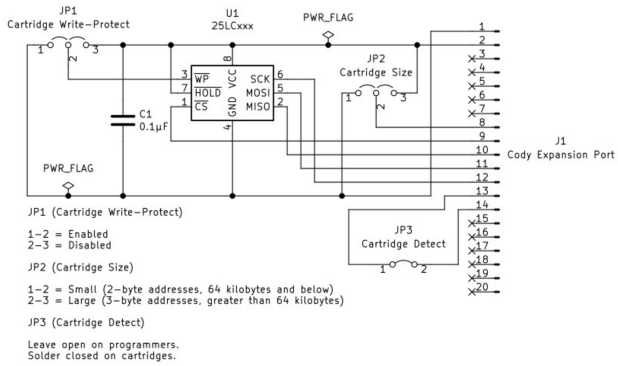
There are actually four different SPI modes. Each mode can differ based on the SPI clock signal's polarity, either being idle-high or idle-low. Each mode can also differ based on the clock phase when data is transmitted or received. This is one of the reasons it's preferable to bit-bang the SPI protocol using the 65C22's general-purpose I/O pins rather than relying on a limited subset of modes that can be supported by the built-in shift register.

The Cody Computer's cartridges are built around the SPI protocol with some extra modifications to support cartridge detection and size determination. The 65C22's CA1 and CA2 handshaking pins on expansion port pins 13 and 14 are used as a cartridge detect. If a cartridge is detected, expansion port pin 8 is used to read if the cartridge is 64K or smaller (0) or larger (1) based on the cartridge's configuration.

Once set up to read from a cartridge, expansion port pin 12 is connected to the SPI clock, pin 11 is connected to the SPI master output/slave input, pin 10 is connected to the master input/slave output, and pin 9 is connected to the SPI chip select. This pin configuration is used to implement the SPI protocol and load the program.

CARTRIDGE DESIGN

The Cody Computer cartridge is a relatively simple design, consisting at heart of an SPI EEPROM, a decoupling capacitor, and a connector to plug into the Cody Computer. It's really no more than a standardized pinout to interface an SPI EEPROM into the system's expansion port.



Schematic of the Cody Cartridge. Note that depending on assembly choices, the board can be either a programmer or just a cartridge.

The cartridge's interface is a 20-pin male header that connects to the female socket on the Cody Computer's expansion port. Most of the pins are unused, but several are in use and directly wired to pins on the SPI EEPROM. These are the SPI clock, MISO (master-in-slave-out), MOSI (master-out-slave-in), and inverted chip select.

Some other pins are used to support the Cody Computer's loading of cartridge data. Two pins are connected to each other on the cartridge itself, making it possible for the Cody Computer to detect a cartridge because the connection is closed when a cartridge is seated. Another pin is used to tell the Cody Computer whether the SPI EEPROM is a small EEPROM (a low value indicates a size of 64 kilobytes or less) or a large EEPROM (a high value indicates a size of over 64 kilobytes). This is necessary because the smaller EEPROMs only accept a two-byte address while the

larger ones require a three-byte address in their SPI transmissions.

The standard Cody Computer cartridge design is interesting in that it can be used to build either a cartridge or a programmer for the SPI EEPROMs used in cartridges. Instead of two versions of the board, there's just one version, but different jumper connections can be used to configure it. For a programmer, jumper wires can be replaced with pin headers and jumpers/shunts, thereby letting the user change the behavior just by moving the jumper blocks around.

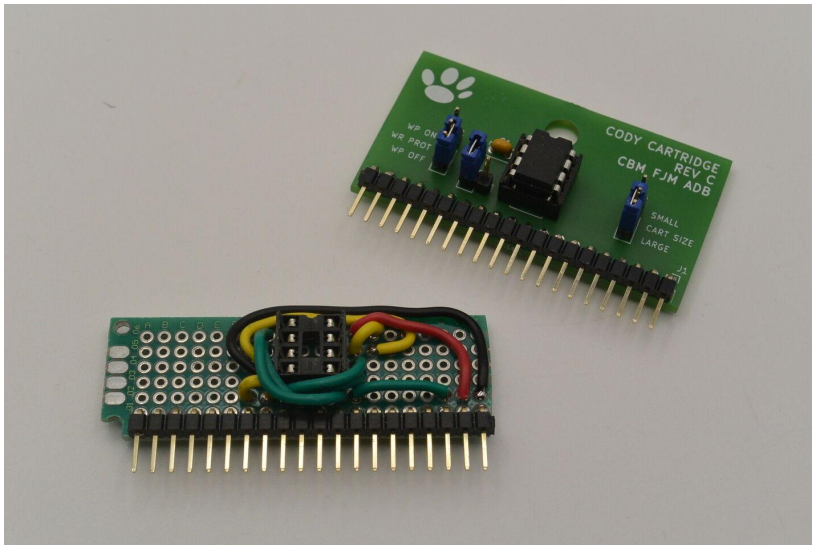
For development purposes we'll start by building a board for programming purposes. We'll cover building a board for a normal cartridge later in the chapter, along with a walkthrough of the mechanical assembly for the case.

CARTRIDGE PROGRAMMER ASSEMBLY

To build a cartridge's PCB as a programmer, header pins are soldered into the board instead of using wires. Jumpers can be used to toggle the different possibilities for the programmer's setup. They can also be used for testing cartridges after they're programmed. A socket is used to (more or less) easily insert and remove the SPI EEPROMs being programmed.

This circuit is actually simple enough that you could build it using point-to-point wiring on a protoboard, as long as the protoboard will fit into the Cody Computer's expansion port hole in the back.

Prototypes of the cartridge were built in exactly such a way during the Cody Computer's development.



A cartridge programmer PCB alongside its hand-wired prototype on protoboard.

However, the rest of the chapter assumes that you have printed circuit boards available.

INSTALLING THE EXPANSION CONNECTOR

The programmer, like the cartridges themselves, has a 20-pin right angle .100" male connector. This matches up with the female connector on the Cody Computer's expanson port when the cartridge is connected.

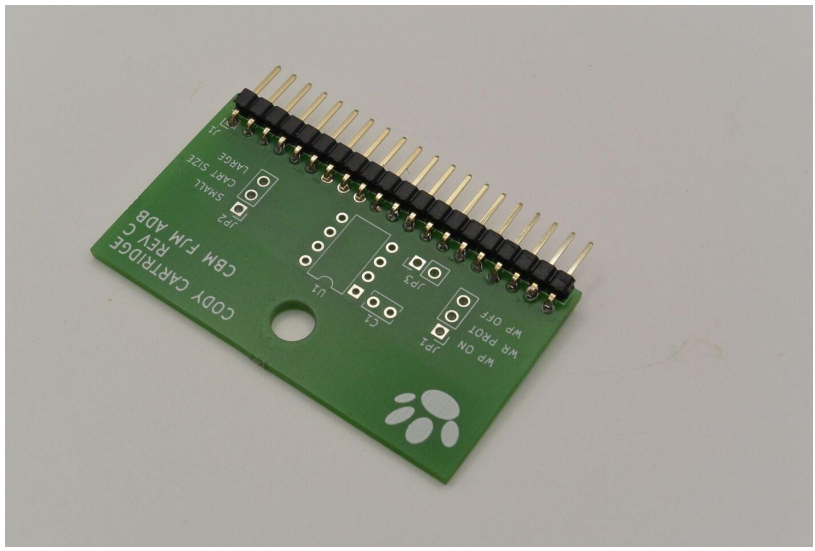
For this step you'll need the following:

- 1 20-pin male .100" right-angle header pin

For this step you need to place the header pins into J1, then solder the connector. It's very important that

the headers go on at a right angle so they will correctly line up with the expansion port's socket.

1. Insert the header into J1. Ensure the pins are at a right-angle to the board.
2. Solder the header to J1.



The board with the connector pins soldered at a right angle.

INSTALLING THE SOCKET AND CAPACITOR

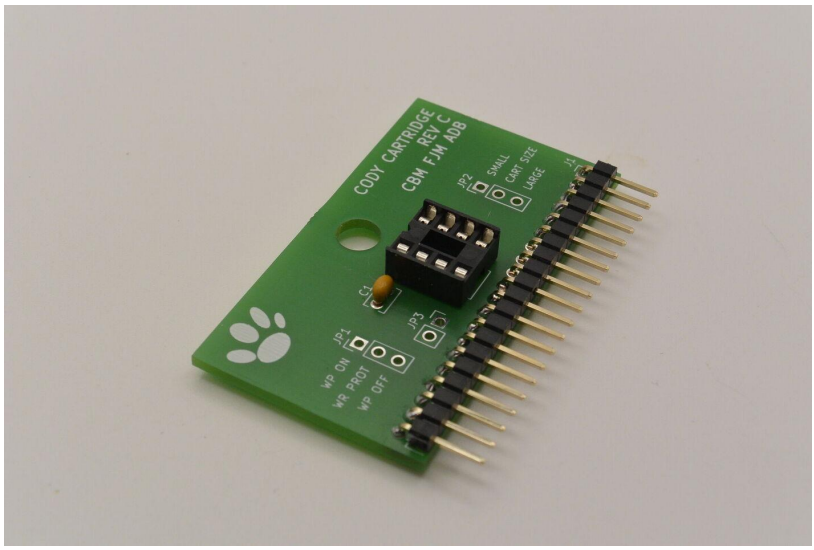
Once the connector is soldered on, it's time to add an 8-pin socket and decoupling capacitor for the SPI EEPROM. The socket makes it easier to insert and remove the IC to be programmed, while the decoupling capacitor serves the same purpose as it

does for ICs on the Cody Computer's main PCB. You'll need the following:

- 1 8-pin DIP socket
- 1 0.1µF ceramic capacitor (KEMET C315C104K1R5TA or equivalent)

For this step you need to solder the IC socket and the capacitor. The IC socket should have a small notch or other mark at the top, and it should align with the notch on the PCB's silkscreen for the part. The decoupling capacitor is not polarized and can be soldered in either direction.

1. Solder the capacitor to C1.
2. Solder the IC socket to U1.



The board with the socket and capacitor added. Note the mark on the IC socket.

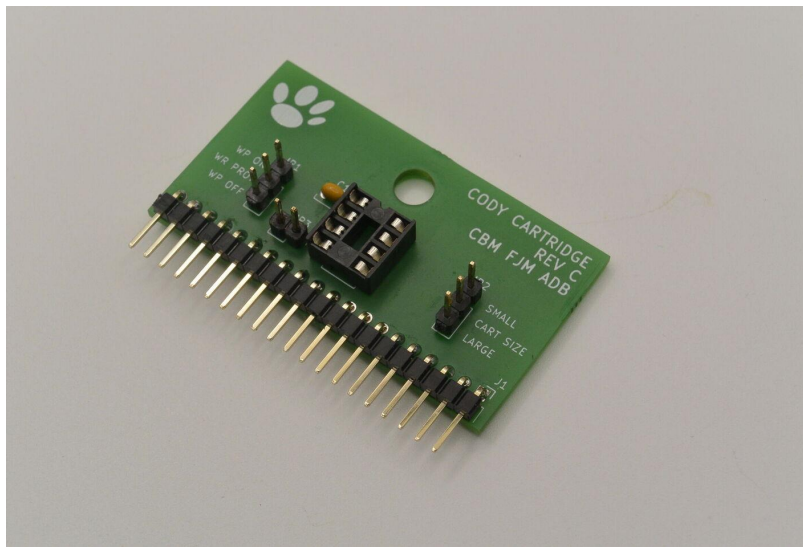
INSTALLING THE HEADERS

In this step we'll add some pin headers to the various jumper positions on the board. This makes it possible to reconfigure the cartridge programmer, whereas for an actual cartridge you could just solder them with jumper wire. This requires the following:

- 2 3-pin male .100" headers, vertical
- 1 2-pin male .100" header, vertical

Soldering the header pins is relatively straightforward:

1. Solder a 3-pin male header to JP1.
2. Solder a 3-pin male header to JP2.
3. Solder the 2-pin male header to JP3.



The board with the jumper headers added.

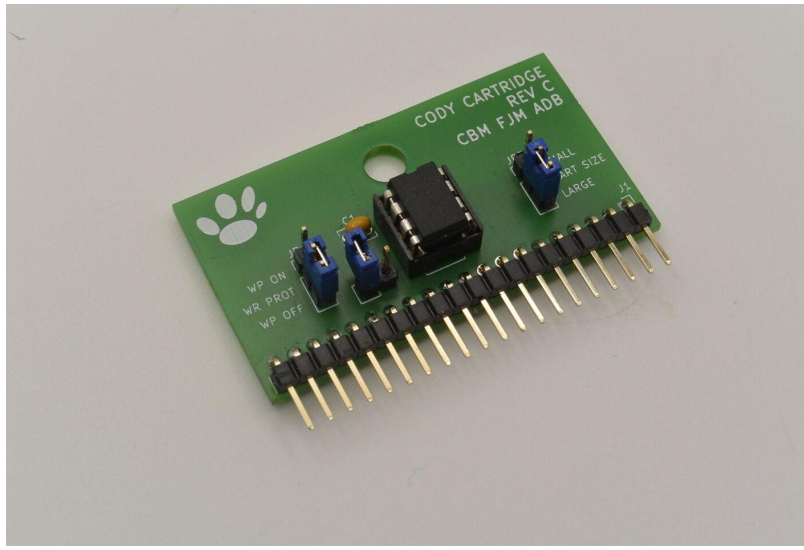
INSERTING THE IC AND JUMPERS

Now we can add the EEPROM IC and jumpers. These steps assume that a 128-kilobyte 25LC1024 SPI EEPROM is being used, so the jumpers will be configured appropriately.

- 1 25LC1024 128-kilobyte SPI EEPROM or equivalent (DIP-8)
- 3 2-pin jumpers/shunts (Harwin M7583-46 or equivalent)

The IC must be carefully inserted without bending the pins. Sliding the jumpers into position is often easier with a pair of tweezers or forceps.

1. Place a jumper on JP1 connecting WR PROT and WP OFF.
2. Place a jumper on JP2 connecting CART SIZE and LARGE.
3. Place a jumper on JP3 connecting only one of the two pins.
4. Insert the 25LC1024 into the socket so that the pin marks align.



The programmer as configured to program a 25LC1024 SPI EEPROM.

SPI PROGRAMMING IN BASIC

Now that you have a board set up to program a cartridge, it's time to learn how to program it. In order to program the SPI EEPROM you'll need to understand some of the key concepts about SPI programming, but you'll also need to understand how the 25LC21024 works when communicating over SPI. To help with that, we'll write some simple Cody BASIC programs before moving on to a more fully-featured programmer in assembly language.

SIMPLE SPI COMMUNICATION

Whenever you're attempting to use SPI to communicate with a device, it's a good idea to start with a simple example and work from there. SPI has

four different modes related to clock edges, and on top of that, not every device is without its own quirks. For our first example, we'll try to read an ID value from the 25LC1024 built into the cartridge as it's a relatively simple operation.

The following Cody BASIC program sends the 25LC1024 an RDID command (decimal 171), which wakes up the chip and reads its built-in ID. This is probably the easiest place to begin with the chip, as the expected ID value is a known quantity from the datasheet. Obtaining it from the chip will tell us that our external hardware is correctly connected and that our program is working as expected.

```
10 REM READ EEPROM RDID
20 GOSUB 1000
30 O=171
40 FOR N=1 TO 5
50 GOSUB 2000
60 NEXT
70 GOSUB 3000
80 PRINT "RDID ID: ",I
90 END
1000 REM SPI BEGIN TRANSACTION
1010 POKE 40706,11
1020 POKE 40704,8
1030 POKE 40704,0
1040 RETURN
2000 REM SPI TRANSMIT AND RECEIVE
2010 FOR Z=1 TO 8
2020 POKE 40704,0
2030 B=0
2040 IF O>127 THEN B=2
2050 POKE 40704,B
2060 POKE 40704,B+1
2070 O=AND(O*2,255)
2080 I=AND(I*2,255)
2090 B=AND(PEEK(40704),4)
2100 IF B>0 THEN I=I+1
2110 NEXT
2120 POKE 40704,0
2130 RETURN
3000 REM SPI END TRANSACTION
3010 POKE 40704,8
3020 RETURN
```

A program that reads the RDID from a 25LC1024 SPI EEPROM.

For this to work you'll need to have the cartridge connected to the expansion port. It's a good idea to turn the Cody Computer off, plug in the cartridge, and then power it on again. The expansion port is not intended to be hot-pluggable, and connecting some

pins before others could potentially cause unexpected behavior or even damage.

When run, the program reads the RDID value from the 25LC1024 EEPROM and prints the received value:

```
RUN
RDID ID: 41

READY.
```

Output from the program reporting the RDID value as 41 decimal.

A TEST PROGRAM

Now that we can talk to the EEPROM, we'll want to have some data to send into it. Because we're also trying to use this as an example of how cartridges work on the Cody Computer's expansion port, we'll put together a small program to store in the EEPROM's memory.

Below is a very short assembly language program that prints a short message on the screen. For this example, all we care about is that we can assemble this code into some data we'll program into the EEPROM.

```
;
; codycart.asm
;
; An example assembly language program for the Cody Computer. The program
; pokes the message "Cody!" into the default screen memory location after
; starting up, then loops forever.
;
; You can assemble the program with 64tass using the following command:
;
; 64tass --mw65c02 --nostart -o codycart.bin codycart.asm
;

ADDR    = $3000                 ; The actual loading address of the program
SCRRAM  = $C400                 ; The default location of screen memory

; Program header for Cody Basic's loader (needs to be first)

.WORD ADDR                      ; Starting address (just like KIM-1, Commodore, etc.)
```

```
.WORD (ADDR + LAST - MAIN - 1)  ; Ending address (so we know when we're done loading)

;
; The actual program.
;

.LOGICAL    ADDR                ; The actual program gets loaded at ADDR

MAIN        LDX #0              ; The program starts running from here

_LOOP       LDA TEXT,X          ; Copies TEXT into screen memory
            BEQ _DONE

            STA SCRRAM,X

            INX
            BRA _LOOP

_DONE       JMP _DONE           ; Loops forever

TEXT        .NULL "Cody!"       ; TEXT as a null-terminated string

LAST                            ; End of the entire program

.ENDLOGICAL
```

A simple assembly language program to store in an EEPROM.

You can assemble this program just like the ones the previous chapter. Assembled into a binary file, the program is only 26 bytes long. It can be represented as a sequence of 26 numbers (0, 48, 21, 48, 162, 0, 189, 16, 48, 240, 6, 157, 0, 196, 232, 128, 245, 76, 13, 48, 67, 111, 100, 121, 33, and 0). We'll rely on this knowledge to program it into the EEPROM chip for our example cartridge.

WRITING TO THE EEPROM

Now that you have a program to put into the EEPROM, you'll need a way to actually write it. Another Cody BASIC program very similar to the previous one can do this. Again, it's only an example, but it can write the values from **DATA** statements into the EEPROM's memory over SPI.

There are some details that need to be covered for this to work. In particular, the 25LC1024 is broken up into a sequence of 256-byte pages. While this is good

for the EEPROM (because write cycles are limited to certain subsets of the whole memory), it's less good for us. It means that we can't just start at memory address 0 and count our way through as we write to the chip. Instead, we have to stop our current write transaction and begin a new one at the end of each page.

Another complication is that the chip itself can take some time to write a byte. We don't need to worry about this in Cody BASIC because our program runs so slow, but in a better EEPROM writer, you would want to check the chip's internal registers to ensure the write cycle had completed.

On the 25LC1024, writes require two steps. We first send the WREN (write enable) command (decimal 6), followed by the actual WRITE (decimal 2) with the starting address to write to. We then just loop over our data until we reach the end, making sure that we stop the current transaction and start over at the end of each page.

```
10 REM WRITE EEPROM DATA
20 A=0
30 REM BEGIN NEW PAGE
40 GOSUB 1000
50 O=6
60 GOSUB 2000
70 GOSUB 3000
80 REM WRITE OPERATION
90 GOSUB 1000
100 O=2
110 GOSUB 2000
120 O=0
130 GOSUB 2000
140 O=A/256
150 GOSUB 2000
160 O=AND(A,255)
```

```
170 GOSUB 2000
180 READ N
190 IF N<0 THEN GOTO 260
200 O=N
210 GOSUB 2000
220 A=A+1
230 IF AND(A,255)>0 THEN GOTO 180
240 GOSUB 3000
250 GOTO 30
260 REM END OF DATA
270 GOSUB 3000
280 END
1000 REM SPI BEGIN TRANSACTION
1010 POKE 40706,11
1020 POKE 40704,8
1030 POKE 40704,0
1040 RETURN
2000 REM SPI TRANSMIT AND RECEIVE
2010 FOR Z=1 TO 8
2020 POKE 40704,0
2030 B=0
2040 IF O>127 THEN B=2
2050 POKE 40704,B
2060 POKE 40704,B+1
2070 O=AND(O*2,255)
2080 I=AND(I*2,255)
2090 B=AND(PEEK(40704),4)
2100 IF B>0 THEN I=I+1
2110 NEXT
2120 POKE 40704,0
2130 RETURN
3000 REM SPI END TRANSACTION
3010 POKE 40704,8
3020 RETURN
4000 REM DATA TO PROGRAM
4010 DATA 0,48,21,48,162,0,189,16
4020 DATA 48,240,6,157,0,196,232,128
4030 DATA 245,76,13,48,67,111,100,121
4040 DATA 33,0,-1
```

A program that writes data into a 25LC1024 SPI EEPROM.

READING THE EEPROM

Now that we've programmed the cartridge we should verify its contents. Fortunately we have another Cody BASIC program that reads from the cartridge instead of writing to it. it's very similar to the previous two SPI programs, particularly with respect to the various subroutines used for the actual SPI operations. Where it differs it that it's set up to run the READ command (decimal 3), which reads the data stored in the EEPROM. The READ operation is simpler as we only need to provide the starting address (0 in our case) and then keep reading data one byte at a time.

```
10 REM READ EEPROM DATA
20 A=0
30 GOSUB 1000
40 O=3
50 GOSUB 2000
60 FOR N=1 TO 3
70 O=0
80 GOSUB 2000
90 NEXT
100 FOR N=1 TO 16
110 GOSUB 2000
120 PRINT A,TAB(10),I
130 A=A+1
140 NEXT
150 PRINT
160 PRINT "MORE (Y/N)";
170 INPUT S$
180 IF S$="Y" THEN GOTO 100
190 GOSUB 3000
200 END
1000 REM SPI BEGIN TRANSACTION
1010 POKE 40706,11
1020 POKE 40704,8
1030 POKE 40704,0
1040 RETURN
```

```
2000 REM SPI TRANSMIT AND RECEIVE
2010 FOR Z=1 TO 8
2020 POKE 40704,0
2030 B=0
2040 IF O>127 THEN B=2
2050 POKE 40704,B
2060 POKE 40704,B+1
2070 O=AND(O*2,255)
2080 I=AND(I*2,255)
2090 B=AND(PEEK(40704),4)
2100 IF B>0 THEN I=I+1
2110 NEXT
2120 POKE 40704,0
2130 RETURN
3000 REM SPI END TRANSACTION
3010 POKE 40704,8
3020 RETURN
```

A program that reads the stored data from a 25LC1024 SPI EEPROM.

If you run the program you should see the same numbers that were in the **DATA** statements in the previous program:

```
RUN
0           0
1           48
2           21
3           48
4           162
5           0
6           189
7           16
8           48
9           240
10          6
11          157
12          0
13          196
14          232
15          128

MORE (Y/N)?
```

Reading the first bytes from the EEPROM.

BOOTING THE CARTRIDGE

Because the cartridge has been programmed, you can also boot from it and run the program it contains. Turn off the Cody Computer and reaffix jumper JP3 so that the cartridge detection is enabled on the cartridge side. Then power the Cody Computer back on.

If everything works as expected, the words "Cody" will appear at the top of the screen. It's as simple as that.

When you're done, shut off the Cody Computer and disconnect JP3, placing the header back on a single pin so that it doesn't get lost. This way the cartridge is ready to be programmed next time.

A PROGRAM FOR PROGRAMMING

It would be possible to write a cartridge programmer in Cody BASIC, but it would also run slower than you would probably prefer. Like we talked about in earlier chapters, you could write parts of your program in assembly language and call them from BASIC to speed them up. But it's probably better to just write a dedicated assembly language program in this case, so in this section that's what we're going to do.

What will our program need to do? Once loaded, the user must be able to send a binary file to the Cody Computer. Because our serial communications don't have any checks on them, we'll actually require the file to be sent twice. We can verify the contents are the same on both transmissions before proceeding. After that we'll want to program the SPI EEPROM with the data, then read back from the SPI EEPROM to make sure everything was copied over correctly.

We already know how to program SPI from the previous section and the provided Cody BASIC examples. We also have code in the Cody BASIC interpreter itself that can handle SPI communications so that cartridges can be loaded. In the chapter on assembly language, we wrote an assembly language program that received a binary file over the UART, in that case to play a SID file. So you've probably seen all the parts, just not assembled in quite this way.

THE CODYPROG PROGRAM

Like our other assembly language programs, this one starts out with a bunch of definitions that we get out of the way in a hurry. Many of them, such as those for screen memory addresses, 65C22 VIA addresses, and UART addresses, have been used in other programs earlier in the book.

```
ADDR      = $0300               ; The actual loading address of the program

SCRRAM    = $C400               ; Screen memory base address

UART1_BASE  = $D480             ; Register addresses for UART 1
UART1_CNTL  = UART1_BASE+0
UART1_CMND  = UART1_BASE+1
UART1_STAT  = UART1_BASE+2
UART1_RXHD  = UART1_BASE+4
UART1_RXTL  = UART1_BASE+5
UART1_TXHD  = UART1_BASE+6
UART1_TXTL  = UART1_BASE+7
UART1_RXBF  = UART1_BASE+8
UART1_TXBF  = UART1_BASE+16

VIA_BASE  = $9F00               ; VIA base address and register locations
VIA_IORB  = VIA_BASE+$0
VIA_IORA  = VIA_BASE+$1
VIA_DDRB  = VIA_BASE+$2
VIA_DDRA  = VIA_BASE+$3
VIA_T1CL  = VIA_BASE+$4
VIA_T1CH  = VIA_BASE+$5
VIA_SR    = VIA_BASE+$A
VIA_ACR   = VIA_BASE+$B
VIA_PCR   = VIA_BASE+$C
VIA_IFR   = VIA_BASE+$D
VIA_IER   = VIA_BASE+$E
```

Some common definitions at the start of the program.

The zero page variables we use are very similar to those in other programs. We also have some variables for a pointer, a top pointer, and a length of the program we're going to burn into the cartridge. Our SPI routines also need a couple of temporary variables we'll define here.

```
STRPTR    = $D0               ; Pointer to string (2 bytes)
SCRPTR    = $D2               ; Pointer to screen (2 bytes)
PRGPTR    = $D4               ; Pointer to the start of the program data
PRGTOP    = $D6               ; Pointer to the end of the program data
PRGLEN    = $D8               ; Length of the program in memory
```

```
KEYROW0   = $DA                ; Keyboard row 0
KEYROW1   = $DB                ; Keyboard row 1
KEYROW2   = $DC                ; Keyboard row 2
KEYROW3   = $DD                ; Keyboard row 3
KEYROW4   = $DE                ; Keyboard row 4
KEYROW5   = $DF                ; Keyboard row 5

SPIINP    = $E0                ; SPI input byte
SPIOUT    = $E1                ; SPI output byte
```

Zero-page variables used by the program.

We also define the start of our buffer for the binary data at **$1000**. Other new definitions include the pins we'll use to talk to the SPI EEPROM inside the cartridge. The expansion port pins we're interested in are wired to 65C22 VIA port B. These constants define the bits that correspond to each pin in its register.

```
PRGMEM    = $1000              ; Start of the program to burn into the EEPROM

CART_CLK  = $01                ; Bit masks for 65C22 port B cartridge pins
CART_MOSI = $02
CART_MISO = $04
CART_CS   = $08
CART_SIZE = $10
```

Other constants required by the program.

Our code contains the same preamble as the other assembly language programs:

```
; Program header for Cody Basic's loader (needs to be first)

.WORD ADDR                     ; Starting address (just like KIM-1, Commodore, etc.)
.WORD (ADDR + LAST - MAIN - 1)  ; Ending address (so we know when we're done loading)

; The actual program goes below here

.LOGICAL    ADDR               ; The actual program gets loaded at ADDR
```

The program's header containing the start and end addresses.

The **MAIN** routine is very similar to the CodySID program's main routine. It has fewer things to do and less to initialize, but the overall pattern is similar. We initialize some variables, draw the screen, and then

scan the keyboard for menu item selections. If a menu item is selected, we branch to that command and call the appropriate routine.

```
;
; MAIN
;
; Main loop of the programmer. Responsible for initialization, information display,
; and menu selection.
;
MAIN        STZ PRGLEN          ; Clear program length
            STZ PRGLEN+1

            JSR SHOWSCRN

_LOOP       JSR KEYSCAN         ; Scan the keyboard

            LDA KEYROW0         ; Pressed Q for quit?
            AND #%00001
            BNE _QUIT

            LDA KEYROW1         ; Pressed L for load?
            AND #%10000
            BNE _LOAD

            LDA KEYROW5         ; Pressed P for program?
            AND #%10000
            BNE _PROG

            BRA _LOOP           ; Repeat main loop

_QUIT       RTS                 ; Return to BASIC

_LOAD       JSR CMDLOAD         ; Run the load command
            BRA _LOOP

_PROG       JSR CMDPROG         ; Run the program command
            BRA _LOOP
```

The actual start of the program.

The **KEYSCAN** routine is also very similar. Again, we don't do any keyboard debouncing because for our particular use case, we don't need it. For general-purpose input, however, it would be a necessity.

```
;
; KEYSCAN
;
; Scans the keyboard matrix (so that key selections for menu options can be detected).
;
KEYSCAN     PHA                 ; Preserve registers
            PHX

            STZ VIA_IORA        ; Start at the first row and first key of the keyboard
            LDX #0

_LOOP       LDA VIA_IORA        ; Read the keys for the current row from the VIA port
            EOR #$FF
            LSR A
            LSR A
            LSR A
```

```
            STA KEYROWØ,X

            INC VIA_IORA          ; Move on to the next keyboard row
            INX

            CPX #6                ; Do we have any rows remaining to scan?
            BNE _LOOP

            PLX                   ; Restore registers
            PLA

            RTS
```

The keyboard-scanning routine.

The menu commands are significantly simpler than in the SID player, and nearly all of the operations are moved into subroutines closer to the action. **CMDLOAD** loads and verifies the binary file coming in over the serial link. **CMDPROG** programs the SPI EEPROM and reads its data back for verification.

```
;
; CMDLOAD
;
; Implements the menu option to load a binary file over the UART connection.
;
CMDLOAD     JSR SHOWSCRN        ; Clear screen

            JSR UARTON          ; Receive the binary file
            JSR LOADBIN
            JSR UARTOFF

            JSR SHOWSCRN        ; Redraw screen with file length

            JSR UARTON          ; Verify the binary file
            JSR VERIBIN
            JSR UARTOFF

            RTS                 ; All done

;
; CMDPROG
;
; Implements the menu option to program the SPI EEPROM on the cartridge.
;
CMDPROG     JSR SHOWSCRN        ; Clear screen

            JSR PROGCART        ; Program the cartridge

            JSR VERICART        ; Verify the cartridge contents

            RTS                 ; All done
```

Routines for the menu commands.

The **LOADBIN** routine is very similar to the SID player's **LOADDATA** routine. It starts at the beginning of the memory buffer and waits for input data. Once a

byte has been received, it enters a loop and continues to read bytes until a timeout is exceeded. Under normal operations the timeout would indicate the end of the incoming file.

```
;
; LOADBIN
;
; Loads a binary file into memory.
;
LOADBIN   LDA #<PRGMEM          ; Move to beginning of memory
          STA PRGPTR+0

          LDA #>PRGMEM
          STA PRGPTR+1

          LDX #MSG_WAITBINA     ; Display message about waiting for data
          JSR SHOWSTAT

_READ1    JSR UARTGET           ; Read the first byte
          BCC _READ1

          JSR _SAVE             ; Save it to memory

          LDX #MSG_RECVDATA     ; Display message about receiving data
          JSR SHOWSTAT

          LDX #$FF              ; Timeout counter

_READ2    DEX                   ; Wait for byte with timeout
          BEQ _DONE

          JSR UARTGET
          BCC _READ2

          JSR _SAVE             ; Save data

          LDX #$FF              ; Reset counter
          BRA _READ2

_DONE     SEC                   ; Calculate program length

          LDA PRGPTR+0
          SBC #<PRGMEM
          STA PRGLEN+0

          LDA PRGPTR+1
          SBC #>PRGMEM
          STA PRGLEN+1

          LDA PRGPTR+0          ; Update end of program
          STA PRGTOP+0

          LDA PRGPTR+1
          STA PRGTOP+1

          RTS

_SAVE     STA (PRGPTR)          ; Store data

          INC PRGPTR+0          ; Increment address
          BNE _NEXT
          INC PRGPTR+1

_NEXT     RTS
```

***LOADBIN** loads a binary file over the UART.*

Similar to **LOADBIN** is the **VERIBIN** routine. This routine verifies the content in the memory buffer is the same as the content coming in over the UART. In this situation, instead of storing each byte, we compare it with the matching byte we already have to make sure they're equal. Once we've come to the end of the file, we also have to make sure we read the same number of bytes both times.

```
;
; VERIBIN
;
; Verifies the binary file in memory.
;
VERIBIN   LDA #<PRGMEM          ; Move to beginning of memory
          STA PRGPTR+Ø

          LDA #>PRGMEM
          STA PRGPTR+1

          LDX #MSG_WAITREPE   ; Display message about waiting for data
          JSR SHOWSTAT

_READ1    JSR UARTGET           ; Read the first byte
          BCC _READ1

          JSR _VERIFY           ; Check the byte against the memory
          BNE _FAILED

          LDX #MSG_VERIDATA     ; Display message about verifying data
          JSR SHOWSTAT

          LDX #$FF              ; Timeout counter

_READ2    DEX                   ; Wait for byte with timeout
          BEQ _DONE
          JSR UARTGET
          BCC _READ2

          LDX #$FF              ; Reset counter

          JSR _VERIFY           ; Check the byte
          BNE _FAILED

          BRA _READ2

_DONE     LDA PRGPTR+Ø          ; Verify program length was the same
          CMP PRGTOP+Ø
          BNE _FAILED

          LDA PRGPTR+1
          CMP PRGTOP+1
          BNE _FAILED

          LDX #MSG_VERIFYOK     ; Update status message
          JSR SHOWSTAT

          RTS

_VERIFY   CMP (PRGPTR)          ; Compare bytes
          PHP

          INC PRGPTR+Ø          ; Increment address
          BNE _NEXT
          INC PRGPTR+1
```

```
_NEXT     PLP                   ; Restore flags and return
          RTS

_FAILED   STZ PRGLEN+0          ; Clear program length (bad file?)
          STZ PRGLEN+1

          LDX #MSG_VERIFYBAD    ; Update status message
          JSR SHOWSTAT

          RTS                   ; All done
```

*The **VERIBIN** routine verifies the program in memory.*

Once the program has been loaded the remaining task is to write the program into the EEPROM. The **PROGCART** routine takes care of this, and it's actually somewhat complicated. It has to send the instructions to enable writing to the EEPROM, then begin a second SPI transaction with the actual data and its start address in the EEPROM.

There are some complications here. One is that cartridges can either be small (64 kilobytes or less) or large (greater than 64 kilobytes). Small cartridges only need two bytes for an address but large cartridges use three bytes. We check the size pin on the expansion port to see what kind of cartridge the programmer is set up for.

Another complication comes from a limitation in the SPI EEPROM's writing protocol. Because of the EEPROM's design, we have to start a new write transaction on each 256-byte page. Because our memory buffer is page-aligned, every time we wrap to another page, we also close the current write transaction and begin a new one. Between them we must wait for the EEPROM to finish writing our data, so we poll the EEPROM's status register in between.

```
;
; PROGCART
;
; Writes the program in memory to the SPI EEPROM on the cartridge.
;
```

```
PROGCART  LDA #<PRGMEM          ; Move to beginning of memory
          STA PRGPTR+Ø

          LDA #>PRGMEM
          STA PRGPTR+1

          LDX #MSG_PROGDATA     ; Display message about programming data
          JSR SHOWSTAT

          JSR _BEGIN            ; Begin initial SPI transaction

_LOOP     LDA PRGPTR+Ø          ; Ensure we're not at the top of the data
          CMP PRGTOP+Ø
          BNE _CONT

          LDA PRGPTR+1
          CMP PRGTOP+1
          BNE _CONT

          JSR _END              ; Done programming

          LDX #MSG_CLEAR        ; Clear status message
          JSR SHOWSTAT

          RTS

_CONT     LDA (PRGPTR)          ; Send the next byte to the cartridge
          JSR CARTXFER

          INC PRGPTR+Ø          ; Increment address
          BNE _LOOP
          INC PRGPTR+1

          JSR _END              ; New page, need to start new transaction
          JSR _BEGIN

          BRA _LOOP

_BEGIN    JSR CARTON            ; Begin SPI transaction for write enable

          LDA #6                ; Write enable command
          JSR CARTXFER

          JSR CARTOFF           ; End SPI transction for write enable

          JSR CARTON            ; Begin SPI transaction for writing data

          LDA #2                ; Write starting address command
          JSR CARTXFER

          JSR CARTSIZE          ; Check cartridge size
          BEQ _ADDR

          LDA #Ø                ; Write address highest byte, greater than 64K only
          JSR CARTXFER

_ADDR     SEC                   ; Write address high byte
          LDA PRGPTR+1
          SBC #>PRGMEM
          JSR CARTXFER

          LDA #Ø                ; Write address low byte
          JSR CARTXFER

          RTS

_END      JSR CARTOFF           ; End previous transaction

          JSR CARTON            ; New transaction to read status register

_WAIT     LDA #5                ; Read status register command
          JSR CARTXFER

          LDA #Ø                ; Read the status register
          JSR CARTXFER

          AND #$Ø1              ; Wait until previous write is completed
```

```
        BNE _WAIT

        JSR CARTOFF          ; End transaction and return

        RTS
```

PROGCART *handles SPI EEPROM programming at a high level.*

We also want to make sure there weren't any glitches when we wrote to the EEPROM, so when we're done, we use the **VERICART** routine to check it. A simpler form of the **PROGCART** routine, it reads the data back from the EEPROM and compares each byte to the contents in the memory buffer.

```
;
; VERICART
;
; Reads the SPI EEPROM and compares it to the program in memory.
;
VERICART  LDA #<PRGMEM         ; Move to beginning of memory
          STA PRGPTR+Ø

          LDA #>PRGMEM
          STA PRGPTR+1

          LDX #MSG_VERIDATA    ; Display message about verifying data
          JSR SHOWSTAT

          JSR CARTON           ; Begin initial SPI transaction

          LDA #3               ; Read command
          JSR CARTXFER

          JSR CARTSIZE         ; Check cartridge size
          BEQ _ADDR

          LDA #Ø               ; Read address highest byte, greater than 64K only
          JSR CARTXFER

_ADDR     LDA #Ø               ; Read address high byte
          JSR CARTXFER

          LDA #Ø               ; Write address low byte
          JSR CARTXFER

_LOOP     LDA PRGPTR+Ø         ; Ensure we're not at the top of the data
          CMP PRGTOP+Ø
          BNE _CONT

          LDA PRGPTR+1
          CMP PRGTOP+1
          BNE _CONT

          JSR CARTOFF          ; Done reading

          LDX #MSG_VERIFYOK    ; Verify passed
          JSR SHOWSTAT

          RTS

_CONT     LDA #Ø               ; Read the next byte from the cartridge
          JSR CARTXFER
```

```
         CMP (PRGPTR)          ; Compare the bytes to verify
         BNE _FAILED

         INC PRGPTR+Ø          ; Increment address
         BNE _LOOP
         INC PRGPTR+1
         BRA _LOOP

_FAILED  JSR CARTOFF           ; Turn off SPI

         LDX #MSG_VERIFYBAD    ; Display verification failed message
         JSR SHOWSTAT

         RTS
```

*The **VERICART** routine checks the program contents against the EEPROM.*

While loading data or programming cartridges, we want to update the current status message on the screen. The **SHOWSTAT** routine lets us redraw just that part of the screen without affecting anything else.

```
;
; SHOWSTAT
;
; Shows a message in the status bar at the bottom of the screen.
; The message number should be in the X register.
;
SHOWSTAT  PHX                  ; Preserve message number

          LDX #Ø               ; Clear status bar
          LDY #11
          JSR MOVESCRN

          LDX #MSG_CLEAR
          JSR PUTMSG

          LDX #Ø               ; Print message
          LDY #11
          JSR MOVESCRN

          PLX
          JSR PUTMSG

          RTS
```

A simple routine to display a status message by number.

A larger routine, **SHOWSCRN** clears the entire screen and draws the menu. This is performed far less frequently, only at startup and at particular stopping points in the program.

```
;
```

```
; SHOWSCRN
;
; Shows the main screen.
;
SHOWSCRN  JSR CLRSCRN

          LDX #Ø
          LDY #Ø
          JSR MOVESCRN

          LDX #MSG_CODYPROG
          JSR PUTMSG

          LDX #Ø
          LDY #1
          JSR MOVESCRN

          LDX #MSG_SUBTITLE
          JSR PUTMSG

          LDX #Ø
          LDY #3
          JSR MOVESCRN

          LDX #MSG_LENGTH
          JSR PUTMSG

          LDX #9
          LDY #3
          JSR MOVESCRN

          LDA PRGLEN+1
          JSR PUTHEX

          LDX #11
          LDY #3
          JSR MOVESCRN

          LDA PRGLEN+Ø
          JSR PUTHEX

          LDX #Ø
          LDY #5
          JSR MOVESCRN

          LDX #MSG_LOADMENU
          JSR PUTMSG

          LDX #Ø
          LDY #6
          JSR MOVESCRN

          LDX #MSG_PROGMENU
          JSR PUTMSG

          LDX #Ø
          LDY #7
          JSR MOVESCRN

          LDX #MSG_QUITMENU
          JSR PUTMSG

          RTS
```

*A rather long **SHOWSCRN** draws most of the user interface.*

The underyling UART routines for loading binary files are identical to those in the SID player example in

the previous chapter. The **UARTON** routine is called before beginning a UART operation.

```
;
; UARTON
;
; Turns on UART 1.
;
UARTON    PHA
          PHY

_INIT     STZ UART1_RXTL          ; Clear out buffer registers
          STZ UART1_TXHD

          LDA #$0F                ; Set baud rate to 19200
          STA UART1_CNTL

          LDA #01                 ; Enable UART
          STA UART1_CMND

_WAIT     LDA UART1_STAT          ; Wait for UART to start up
          AND #$40
          BEQ _WAIT

          PLY
          PLA

          RTS                     ; All done
```

***UARTON** turns on UART 1.*

Its companion routine, **UARTOFF**, turns off the UART at the end of a read operation.

```
;
; UARTOFF
;
; Turns off UART 1.
;
UARTOFF   PHA

          STZ UART1_CMND          ; Clear bit to stop UART

_WAIT     LDA UART1_STAT          ; Wait for UART to stop
          AND #$40
          BNE _WAIT

          PLA

          RTS
```

***UARTOFF** shuts off UART 1.*

Reading from the UART is handled by the **UARTGET** routine. It checks to see if a byte is in the receive buffer. If not, it fails fast, but if there is, it reads the

byte and returns it in the accumulator. The carry flag is used to indicate if a byte was read.

```
;
; UARTGET
;
; Attempts to read a byte from the UART 1 buffer.
;
UARTGET   PHY

          LDA UART1_STAT          ; Test no error bits set in the status register
          BIT #$06
          BNE _ERR

          LDA UART1_RXTL          ; Compare current tail to current head position
          CMP UART1_RXHD
          BEQ _EMPTY

          TAY                     ; Read the next character from the buffer
          LDA UART1_RXBF,Y

          PHA                     ; Increment the receiver tail position
          INY
          TYA
          AND #$07
          STA UART1_RXTL
          PLA

          PLY
          SEC                     ; Set carry to indicate a character was read
          RTS

_EMPTY    PLY
          CLC                     ; Clear carry to indicate no character read
          RTS

_ERR      LDX #MSG_ERROR          ; UART error, display error status message
          JSR SHOWSTAT

_DONE     JMP _DONE
```

UARTGET *polls the UART and returns a byte if available.*

SPI routines are contained in the various **CART** routines that talk to the cartridge on the expansion port. Because of the simple nature of the SPI protocol, these routines are the same as those used to read a cartridge in Cody BASIC. We just use them differently.

The only new routine is the **CARTSIZE** routine that tests whether the cartridge is small or large. It does so by examining the value of the matching I/O pin.

```
;
; CARTSIZE
;
; Checks the cartridge size as small (64K or less) or large (greater than 64K).
```

```
; Cartridges greater than 64K require an additional address byte.
;
CARTSIZE  LDA VIA_IORB
          AND #CART_SIZE

          RTS
```

A simple routine to check a cartridge's size before writing.

The **CARTON** routine begins an SPI transaction by setting the appropriate pins on the expansion port. Most importantly, it brings the SPI chip select pin from high to low to initiate the transaction itself.

```
;
; CARTON
;
; Starts an SPI transation on the cartridge pins for the expansion port. The proper
; directions for 65C22 port B are set, outputs are set, and then the chip select is
; brought low.
;
; Calls to CARTON should be matched by a call to CARTOFF. The presence of a cartridge
; should be verified by a prior call to CARTCHECK.
;
CARTON    LDA #(CART_CLK | CART_MOSI | CART_CS)    ; Set port B directions
          STA VIA_DDRB

          LDA #CART_CS        ; Start with SPI select high
          STA VIA_IORB

          LDA #Ø              ; Bring select low to begin a cycle
          STA VIA_IORB

          RTS
```

***CARTON** begins an SPI transaction.*

CARTOFF brings the SPI chip select high to end the current transaction.

```
;
; CARTOFF
;
; Brings the chip select high at the end of an SPI transaction with a cartridge.
;
CARTOFF   LDA #CART_CS        ; Bring select high to end the transaction
          STA VIA_IORB

          RTS
```

***CARTOFF** ends the current SPI transaction.*

The **CARTXFER** routine is more complicated and handles the actual exchange of data. A byte is shifted

out over the SPI pins while another byte is shifted in at the same time. Rather than use the 65C22 VIA's shift register (which has complications that we won't cover here), we bit-bang the port directly. SPI data is sent with the highest bit first, so we shift ot the left and look at our carry bits.

```
;
; CARTXFER
;
; Transfers a single byte during an SPI transaction with a cartridge. The value
; to send should be stored in the accumulator, and it will be replaced by the
; value received.
;
CARTXFER  PHX

          STA SPIOUT

          STZ SPIINP

          LDX #8              ; 8 bits to read

_LOOP     STZ VIA_IORB        ; Bring the clock low

          LDA #0              ; Start with no data

          ROL SPIOUT          ; Get output bit

          BCC _SEND

          ORA #CART_MOSI      ; Output bit was a 1

_SEND     STA VIA_IORB        ; Put the bit on MOSI

          ORA #CART_CLK       ; Bring the SPI clock high
          STA VIA_IORB

          ROL SPIINP          ; Rotate SPI input for next bit

          LDA VIA_IORB        ; Read the incoming MISO
          AND #CART_MISO

          BEQ _NEXT

          LDA SPIINP
          ORA #1
          STA SPIINP

_NEXT     DEX                 ; Next loop (if any remain)
          BNE _LOOP

          PLX

          LDA SPIINP

          RTS
```

*The **CARTXFER** sends and receives a single SPI byte.*

The other routines are copied verbatim from earlier examples. **MOVESCRN** moves the current screen pointer to a particular row and column.

```
;
; MOVESCRN
;
; Moves the SCRPTR to the position for the column/row in the X and Y
; registers. All registers are clobbered by the routine.
;
MOVESCRN  LDA #<SCRRAM            ; Move screen pointer to beginning
          STA SCRPTR+0
          LDA #>SCRRAM
          STA SCRPTR+1

          INY                     ; Increment pointer for each row
_LOOPY    CLC
          LDA SCRPTR+0
          ADC #40
          STA SCRPTR+0
          LDA SCRPTR+1
          ADC #0
          STA SCRPTR+1
          DEY
          BNE _LOOPY

          CLC                     ; Add position on column
          TXA
          ADC SCRPTR+0
          STA SCRPTR+0
          LDA SCRPTR+1
          ADC #0
          STA SCRPTR+1

          RTS
```

A routine to position the next output on the screen.

Another routine you've seen before, **CLRSCRN**, clears the entire screen by filling it with whitespace.

```
;
; CLRSCRN
;
; Clear the entire screen by filling it with whitespace (ASCII 20 decimal).
;
CLRSCRN   LDA #<SCRRAM            ; Move screen pointer to beginning
          STA SCRPTR+0
          LDA #>SCRRAM
          STA SCRPTR+1

          LDA #20                 ; Clear screen by filling with whitespaces

          LDY #25                 ; Loop 25 times on Y

_LOOPY    LDX #40                 ; Loop 40 times on X for each Y

_LOOPX    STA (SCRPTR)            ; Store zero

          INC SCRPTR+0            ; Increment screen position
          BNE _NEXT
          INC SCRPTR+1

_NEXT     DEX                     ; Next X
          BNE _LOOPX
```

```
        DEY                     ; Next Y
        BNE _LOOPY

        RTS
```

The screen-clearing routine.

The **PUTMSG** routine puts a string identified by a message number onto the screen starting at the current location.

```
;
; PUTMSG
;
; Puts a message string (one of the MSG_XXX constants) on the screen.
;
PUTMSG      PHA
            PHY

            LDA MSGS_L,X        ; Load the pointer for the string to print
            STA STRPTR+Ø
            LDA MSGS_H,X
            STA STRPTR+1

            LDY #Ø

_LOOP       LDA (STRPTR),Y      ; Read the next character (check for null)
            BEQ _DONE

            JSR PUTCHR          ; Copy the character and move to next
            INY

            BRA _LOOP           ; Next loop

_DONE       PLY
            PLA

            RTS
```

***PUTMSG** prints a message on the screen.*

The **PUTCHR** routine is used internally to copy each individual character in the message.

```
;
; PUTCHR
;
; Puts an individual ASCII character on the screen.
;
PUTCHR      STA (SCRPTR)        ; Copy the character

            INC SCRPTR+Ø        ; Increment screen position
            BNE _DONE
            INC SCRPTR+1

_DONE       RTS
```

***PUTCHR** plots the individual characters.*

The **PUTHEX** routine plots the byte in the accumulator as two hex digits. In the SID player this routine was used a lot to show the current register values. In this program we only need it to display the program's length as a hex value for sanity checking.

```
;
; PUTHEX
;
; Puts a byte's hex value on the screen as two hex digits.
;
PUTHEX      PHA
            PHX
            TAX
            JSR HEXTOASCII
            PHA
            TXA
            LSR A
            LSR A
            LSR A
            LSR A
            JSR HEXTOASCII
            PHA
            PLA
            JSR PUTCHR
            PLA
            JSR PUTCHR
            PLX
            PLA
            RTS
HEXTOASCII  AND #$F
            CLC
            ADC #48
            CMP #58
            BCC _DONE
            ADC #6
_DONE       RTS
```

***PUTHEX** prints a byte as two hex digits.*

The message table in this program is different, so our constants below are different.

```
;
; IDs for the message strings that can be displayed in the program.
;
MSG_CODYPROG  = 0
MSG_SUBTITLE  = 1
MSG_LOADMENU  = 2
MSG_PROGMENU  = 3
MSG_QUITMENU  = 4
MSG_WAITBINA  = 5
MSG_WAITREPE  = 6
MSG_RECVDATA  = 7
MSG_PROGDATA  = 8
MSG_VERIDATA  = 9
MSG_VERIFYOK  = 10
MSG_VERIFYBAD = 11
MSG_LENGTH    = 12
MSG_CLEAR     = 13
```

```
MSG_ERROR     = 14
```

The constants for the messages in the string table.

The actual string contents of the messages, of course, are also different. The text relates to the menu options and status updates involved in programming the SPI EEPROM in the cartridge.

```
;
; The strings displayed by the program.
;
STR_CODYPROG  .NULL "CodyProg"
STR_SUBTITLE  .NULL "The Cody Cartridge Programmer"
STR_LOADMENU  .NULL "(L)oad binary"
STR_PROGMENU  .NULL "(P)rogram cartridge"
STR_QUITMENU  .NULL "(Q)uit"
STR_WAITBINA  .NULL "Waiting for binary data..."
STR_WAITREPE  .NULL "Waiting for repeat data to verify..."
STR_RECVDATA  .NULL "Receiving data..."
STR_PROGDATA  .NULL "Programming data..."
STR_VERIDATA  .NULL "Verifying data..."
STR_VERIFYOK  .NULL "Verify OK."
STR_VERIFYBAD .NULL "Verify FAILED."
STR_LENGTH    .NULL "Length: $"
STR_CLEAR     .NULL "                                      "
STR_ERROR     .NULL "ERROR"
```

The string literals for the program's messages.

The message table consists of the string addresses split into low and high bytes. As in the other programs, this allows a quick lookup of the string using an index.

```
;
; Low bytes of the string table addresses.
;
MSGS_L
  .BYTE <STR_CODYPROG
  .BYTE <STR_SUBTITLE
  .BYTE <STR_LOADMENU
  .BYTE <STR_PROGMENU
  .BYTE <STR_QUITMENU
  .BYTE <STR_WAITBINA
  .BYTE <STR_WAITREPE
  .BYTE <STR_RECVDATA
  .BYTE <STR_PROGDATA
  .BYTE <STR_VERIDATA
  .BYTE <STR_VERIFYOK
  .BYTE <STR_VERIFYBAD
  .BYTE <STR_LENGTH
  .BYTE <STR_CLEAR
  .BYTE <STR_ERROR

;
; High bytes of the string table addresses.
;
MSGS_H
  .BYTE >STR_CODYPROG
  .BYTE >STR_SUBTITLE
```

```
  .BYTE >STR_LOADMENU
  .BYTE >STR_PROGMENU
  .BYTE >STR_QUITMENU
  .BYTE >STR_WAITBINA
  .BYTE >STR_WAITREPE
  .BYTE >STR_RECVDATA
  .BYTE >STR_PROGDATA
  .BYTE >STR_VERIDATA
  .BYTE >STR_VERIFYOK
  .BYTE >STR_VERIFYBAD
  .BYTE >STR_LENGTH
  .BYTE >STR_CLEAR
  .BYTE >STR_ERROR
```

The low and high portions of the strings' addresses.

The program ends with the same boilerplate as the others.

```
LAST                            ; End of the entire program

.ENDLOGICAL
```

The end of the program.

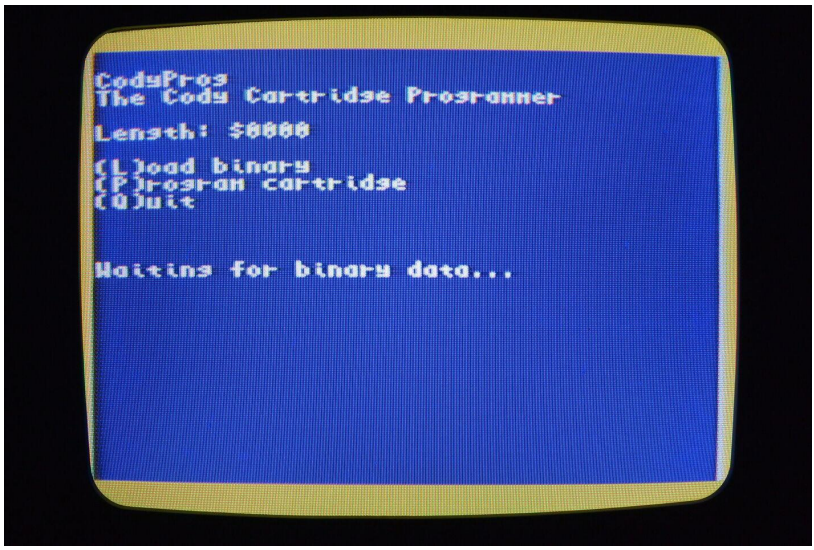
USING THE PROGRAMMER

Build the programmer utility by running it through *64tass* assembler on your PC. Just run **64tass --mw65c02 --nostart -o codyprog.bin codyprog.asm**. These are the same steps as in the previous chapter for assembly language programs.

Once you've done that, turn off the Cody Computer and plug the cartridge programmer into the expansion slot. Turn the Cody Computer back on and load the programmer utility using the **LOAD 1,1** command. Remember that the second argument is also a 1 because the program is a binary and not a BASIC program.

Once loaded we can begin programming a cartridge. Press the L key to load a binary to the programmer, then send the **codybros.bin** binary file you built in the previous chapter. You will actually be prompted for the

file twice, first for the load and the second time to verify the contents are identical.



The programmer program running and waiting for a binary file.

Once the binary is verified, press the P key to program the cartridge. This will begin the programming of the SPI EEPROM inserted into the DIP socket on the programmer board. It will take a few moments and then read the contents back to verify that no errors occurred while programming.

Once done you can test out the cartridge. Turn off the Cody Computer and reconnect JP3, the cartridge detect, on the cartridge programmer board. Turn the Cody Computer back on and watch the program load from the cartridge.

The Cody Bros example from the previous chapter now running as a cartridge.

CARTRIDGE CASE ASSEMBLY

Cartridges, particularly the more permanent kind, can be built into a case. STL files are provided for a case that will fit the cartridge PCB. Assembly is relatively straightforward.

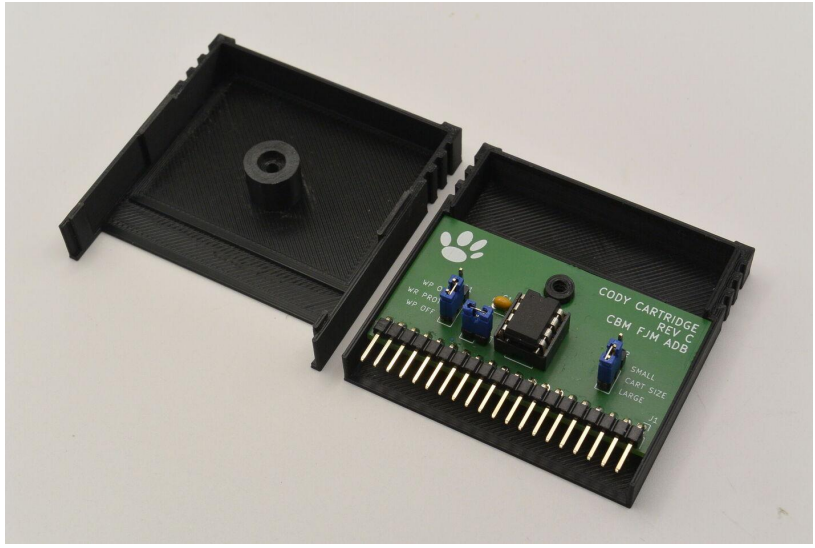
When building a cartridge PCB for use as an actual cartridge rather than as a programmer, it's better if you solder actual jumpers on the board rather than using header pins and blocks. You would make the same connections the jumper blocks would when the programmer is used in cartridge mode (including the JP3 cartridge-detect), but make them in a more permanent fashion. However, even the PCB built as a programmer will (barely) fit into the provided cartridge case design.

For this step you'll need the following:

- 1 completed cartridge PCB (see above notes)
- 1 cartridge top (**CartridgeTop.stl**)
- 1 cartridge bottom (**CartridgeBottom.stl**)
- 1 4 M3 x 10mm self-tapping screw, round/pan head (US #4 x 3/8")
- Screwdriver

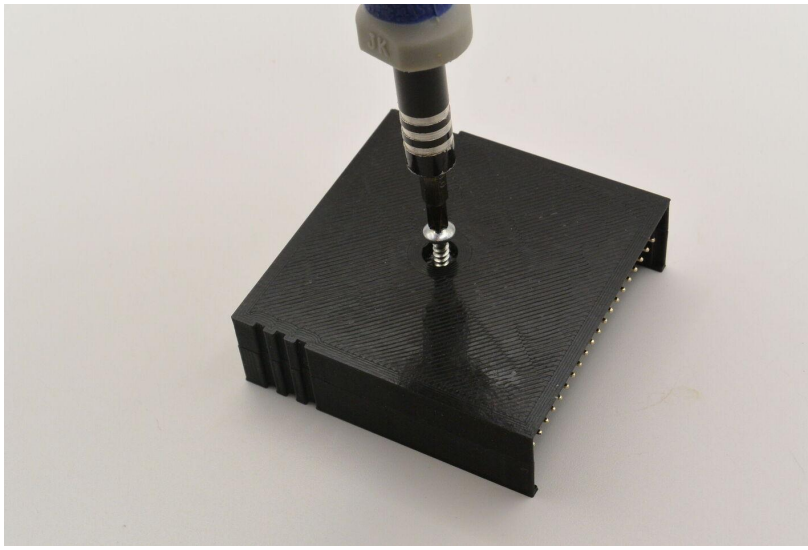
The cartridge halves are intended to be printed with the outside parts against the print bed. For the top half of the cartridge, it will require some supports for the recessed label area. Removing these supports shouldn't be too difficult, and with some care, any damage from the removal should be hidden under the label area.

To begin ensure that the finished PCB fits into the cartridge bottom. The PCB should fit regardless of whether it was built as a cartridge or a programmer. Sanding may be required.



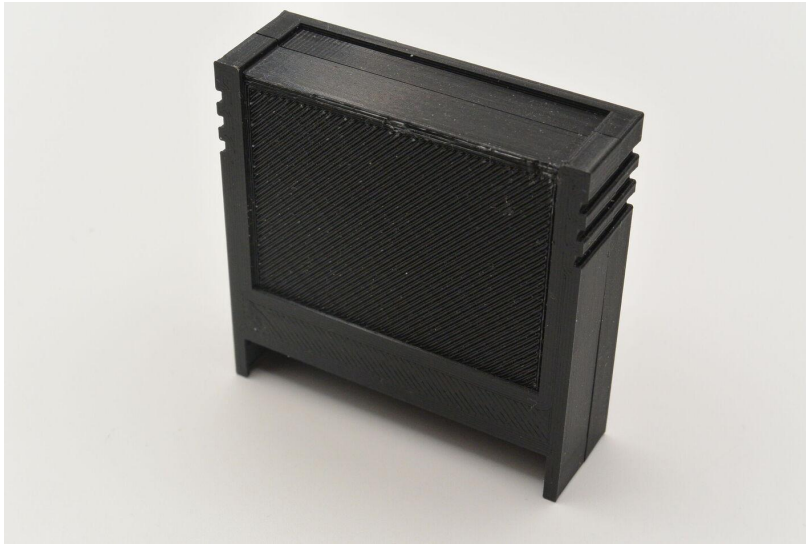
The cartridge case parts with board inserted. For a true "cartridge" the PCB should be built as an actual cartridge rather than a programmer, but it should fit mechanically either way.

With the board in place, pop the top and bottom halves of the cartridge together. Some sanding may again be required to ensure a snug fit. Take the M3 screw and screw it into the cartridge through the back.



Inserting the M3 screw that holds the cartridge together.

This should affix the two halves together as well as secure the board. A recessed area on the cartridge is suitable for affixing a permanent label. Additional sanding or post-processing may be required to ensure a smooth surface for affixing the label.



The finished cartridge waiting for a label.

Afterword

ONE GOOD LITTLE DUDE

He wasn't much of a dog, but he was a great little kid. A few memories of the real Cody as we knew him.

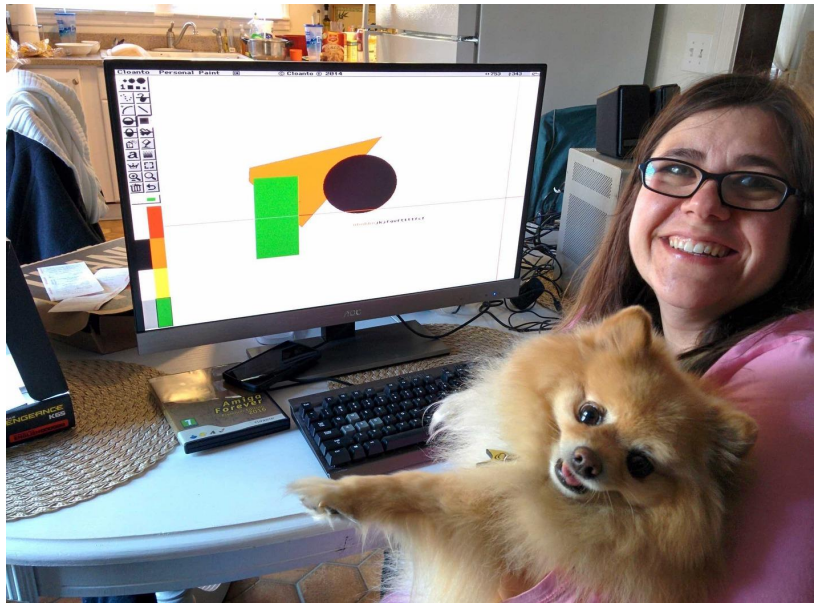
This Used to Be the Future. Cody gazing at relics of the space shuttle program. Pima Air and Space Museum, Tucson, Arizona.

Model Behavior. Studying a wooden model of the ESA's Jules Verne as docked with Zvezda. Ripley's Believe-It-or-Not, Saint Augustine, Florida.

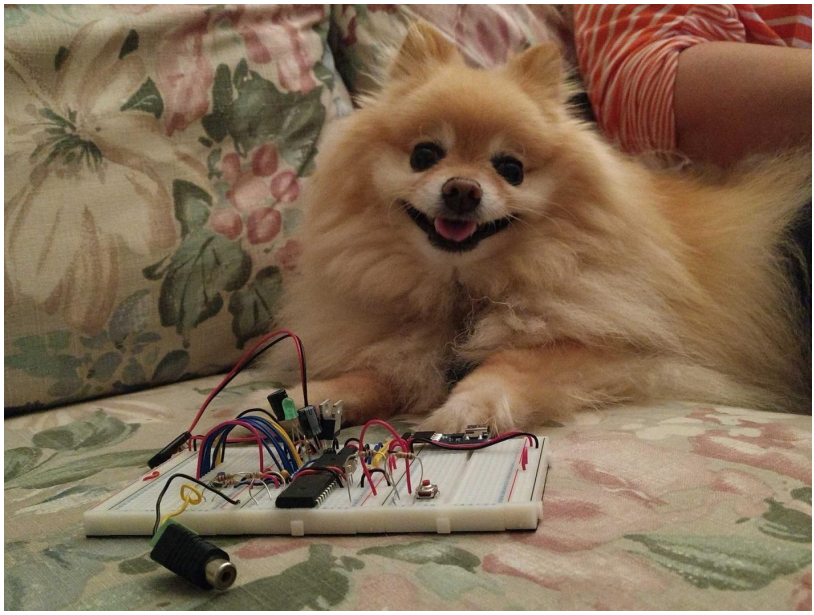
Star Trekkin'. Science Officer Cody conducting a routine planetary survey near Kodachrome Basin State Park. Devil's Garden, Utah.

Digitize Me, Daddy! Cody retracing the steps of Galaxy Quest. Goblin Valley State Park, Utah.

Preparing for Launch. Cody watching as I fumble around in a bag for a model rocket engine and igniter. Bonneville Salt Flats, Utah.

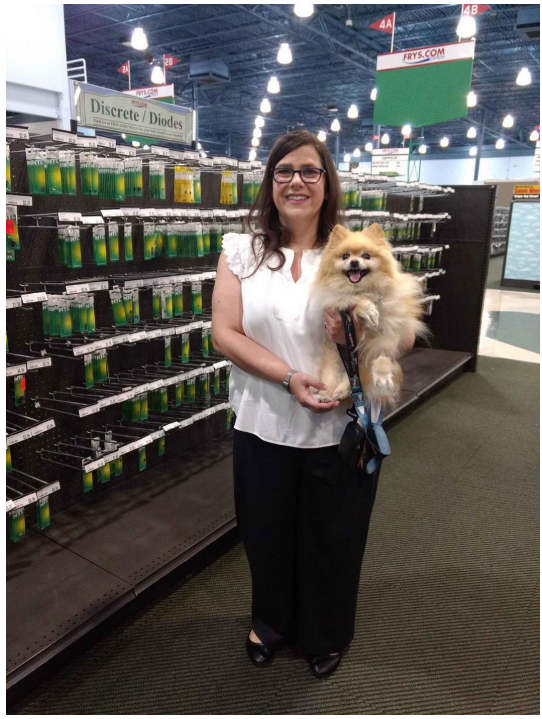


Artiste. Cody and his mom taking a break from the Commodore Amiga's Personal Paint. Folkston, Georgia.



Just a Wee Calculator. Cody with an early version of the circuit that would grow into the Cody Computer. Folkston, Georgia.

Design Review. Cody posing with a late revision of the Cody Computer on a breadboard (literally). Mesa, Arizona.



Shopping Trip. Cody and his mom in the semiconductor aisle of a now-defunct Fry's Electronics. Phoenix, Arizona.

Duplication. Cody watching our new Creality Ender 3 Pro print a tiny little dog for a test print. Mesa, Arizona.

Appendices

APPENDIX A: MEMORY MAP

The Cody Computer's 64 kilobytes of memory contains different RAM and ROM regions as well as several memory-mapped peripherals. This memory map will help you when designing the layout of your own programs, particularly in assembly language. You will need to know the addresses of the various peripherals whether programming in Cody BASIC or in assembly language.

Address	Description
$0000	65C02 zero page variables
$0100	65C02 stack page
$9F00	65C22 Versatile Interface Adapter (VIA) registers
$A000	Beginning of Propeller shared memory
$D000	Video Interface Device (VID) registers
$D040	Video Interface Device (VID) control bank
$D060	Video Interface Device (VID) data bank
$D080	Video Interface Device (VID) sprite banks
$D400	Sound Interface Device (SID) registers
$D480	UART 1 registers
$D4A0	UART 2 registers
$E000	Cody BASIC ROM (character set)
$E800	Cody BASIC ROM (BASIC interpreter)
$FFFF	End of memory

65C02 ZERO PAGE VARIABLES

In Cody BASIC most of the 65C02 zero page is used by the interpreter. Several of these memory locations are intended for use by Cody BASIC programs through the **PEEK** and **POKE** operations.

The **ISRPTR** address is relevant to assembly language programs that wish to register an interrupt handler. Cody BASIC already registers an interrupt handler at this address on startup.

Address	Description
$0000	**SYS** call A register (Cody BASIC)
$0001	**SYS** call X register (Cody BASIC)
$0002	**SYS** call Y register (Cody BASIC)
$0008	**ISRPTR** (2 bytes, assembly)
$000E	**INPUT** prompt character code (Cody BASIC)
$0010	Keyboard row 0 state (Cody BASIC)
$0011	Keyboard row 1 state (Cody BASIC)
$0012	Keyboard row 2 state (Cody BASIC)
$0013	Keyboard row 3 state (Cody BASIC)
$0014	Keyboard row 4 state (Cody BASIC)
$0015	Keyboard row 5 state (Cody BASIC)
$0016	Joystick 1 state (Cody BASIC)
$0017	Joystick 2 state (Cody BASIC)
$004B	Boundary page for program memory (Cody BASIC)

65C22 VERSATILE INTERFACE ADAPTER (VIA) REGISTERS

The 65C22 is a 6502-family I/O chip currently in production by the Western Design Center. Aside from the UARTs implemented by the Propeller, all of the Cody Computer's input and output is handled by this chip. It's the modern version of the classic 6522 VIA used in many vintage computers.

The below table lists the VIA registers as they exist within the Cody Computer's memory map. Port A is used internally for keyboard and joystick scanning while port B is open for use on the expansion port.

For detailed documentation on the chip's functions, refer to WDC's data sheet.

Address	Description
$9F00	Input/output register B
$9F01	Input/output register A
$9F02	Data direction register B
$9F03	Data direction register A
$9F04	Timer 1 latch/counter (low byte)
$9F05	Timer 2 counter (high byte)
$9F06	Timer 1 latch (low byte)
$9F07	Timer 1 latch (high byte)
$9F08	Timer 2 latch/counter (low byte)
$9F09	Timer 2 counter (high byte)
$9F0A	Shift register
$9F0B	Auxiliary control register
$9F0C	Peripheral control register
$9F0D	Interrupt flag register

Address	Description
$9F0E	Interrupt enable registr
$9F0F	Input/output register A (no handshake)

VIDEO INTERFACE DEVICE (VID) REGISTERS

The Cody VID is a software-implemented video device built using the Propeller. It is inspired by, but different from, the VIC-II and its multicolor graphics mode.

Address	Description
$D000	Blanking register (nonzero during blanking interval)
$D001	Control register

- Bit 0 disables screen output.
- Bit 1 enables vertical scrolling (24 rows).
- Bit 2 enables horizontal scrolling (38 columns).
- Bit 3 enables row effects.
- Bit 4 enables bitmap mode.

$D002 Color register

- Bits 0-3 contain border color.
- Bits 4-7 contain color memory location.

$D003 Base register

- Bits 0-3 contain character memory location.
- Bits 4-7 contain screen memory location.

$D004 Scroll register

Address	Description

- Bits 0-3 contain vertical scroll (0-7).
- Bits 4-7 contain horizontal scroll (0-3).

$D005 Screen colors register

- Bits 0-3 contain character color 2.
- Bits 4-7 contain character color 3.

$D006 Sprite register

- Bits 0-3 contain common sprite color.
- Bits 4-7 contain current sprite bank.

The Video Interface Device also has two banks responsible for implementing row effects. A row effect changes part of the screen for one of the 25 character rows and replaces the the raster interrupt effects used on the Commodore 64. One bank controls the effect to apply while the other bank contains the replacement value.

Address	Description
$D040	Row effect control bank (32 bytes)

- Bits 0-4 contain row number.
- Bits 5-6 contain destination (see below).
- Bit 7 enables the effect.

Destinations can be the following:

- 00 overrides the base register.

Address	Description
	• 01 overrides the scroll register. • 10 overrides the screen register. • 11 overrides the sprite register.
$D060	Row effect data bank (32 bytes)

The VID has four different sprite banks that take up the remainder of the page:

Address	Description
$D080	Sprite bank 0
$D0A0	Sprite bank 1
$D0C0	Sprite bank 2
$D0E0	Sprite bank 3

Each entry in a sprite bank is a contiguous group of four bytes. A single sprite bank has eight sprites, all of which are set up exactly like the below table.

Offset	Description
+0	Sprite x-coordinate (0 to 184)
+1	Sprite y-coordinate (0 to 242)
+2	Sprite colors • Bits 0-3 contain color 1. • Bits 4-7 contain color 2.
+3	Sprite location.

SOUND INTERFACE DEVICE (SID) REGISTERS

The Cody Computer has a sound interface device based on the Commodore/MOS 6581. It is implemented within the Propeller chip as a software emulation. Not all SID features are supported and the implementation is not an exact SID replacement. Filters and combined waveforms, among other features, are not implemented at all.

Refer to Chapter 8, Sound and Music Programming, for an explanation of the frequency and ADSR values.

Address	Description
$D400	Voice 1 frequency value (low byte)
$D401	Voice 1 frequency value (high byte)
$D402	Voice 1 pulse duty cycle (low byte)
$D403	Voice 1 pulse duty cycle (high byte)

- Bits 0-3 contain the high nibble.
- Bits 4-7 are unused.

$D404 Voice 1 control register

- Bit 0 ("gate") plays/ends the sound.
- Bit 1 syncs with voice 3 oscillator.
- Bit 2 enables ring modulation with voice 3.
- Bit 3 resets the voice internally.
- Bit 4 selects a triangle wave.
- Bit 5 selects a sawtooth wave.
- Bit 6 selects a pulse wave.
- Bit 7 selects a random noise output.

Address	Description
$D405	Voice 1 attack and decay register

- Bits 0-3 contain the decay value.
- Bits 4-7 contain the attack value.

$D406	Voice 1 sustain and release register

- Bits 0-3 contain the release value.
- Bits 4-7 contain the sustain value.

$D407	Voice 2 frequency value (low byte)
$D408	Voice 2 frequency value (high byte)
$D409	Voice 2 pulse duty cycle (low byte)
$D40A	Voice 2 pulse duty cycle (high byte)

- Bits 0-3 contain the high nibble.
- Bits 4-7 are unused.

$D40B	Voice 2 control register

- Bit 0 ("gate") plays/ends the sound.
- Bit 1 syncs with voice 1 oscillator.
- Bit 2 enables ring modulation with voice 1.
- Bit 3 resets the voice internally.
- Bit 4 selects a triangle wave.
- Bit 5 selects a sawtooth wave.
- Bit 6 selects a pulse wave.
- Bit 7 selects a random noise output.

$D40C	Voice 2 attack and decay register

- Bits 0-3 contain the decay value.

Address **Description**

- Bits 4-7 contain the attack value.

$D40D Voice 2 sustain and release register

- Bits 0-3 contain the release value.
- Bits 4-7 contain the sustain value.

$D40E Voice 3 frequency value (low byte)
$D40F Voice 3 frequency value (high byte)
$D410 Voice 3 pulse duty cycle (low byte)
$D411 Voice 3 pulse duty cycle (high byte)

- Bits 0-3 contain the high nibble.
- Bits 4-7 are unused.

$D412 Voice 3 control register

- Bit 0 ("gate") plays/ends the sound.
- Bit 1 syncs with voice 2 oscillator.
- Bit 2 enables ring modulation with voice 2.
- Bit 3 resets the voice internally.
- Bit 4 selects a triangle wave.
- Bit 5 selects a sawtooth wave.
- Bit 6 selects a pulse wave.
- Bit 7 selects a random noise output.

$D413 Voice 3 attack and decay register

- Bits 0-3 contain the decay value.
- Bits 4-7 contain the attack value.

$D414 Voice 1 sustain and release register

Address	Description

- Bits 0-3 contain the release value.
- Bits 4-7 contain the sustain value.

$D415	Reserved
$D416	Reserved
$D417	Reserved
$D418	Volume control

- Bits 0-3 contain the global volume.

$D419	Reserved
$D41A	Reserved
$D41B	Voice 3 oscillator (read)
$D41C	Voice 3 envelope (read)

UART 1 REGISTERS

Cody Computer UART 1 is connected to the Prop Plug port on the back of the computer. As with most Cody Computer peripherals, it is implemented using the Propeller. This device is generally used for serial communications with your PC or for transferring files. Bit rate options are copied from the 6551 ACIA:

- $0 is not supported.
- $1 for 50 BPS.
- $2 for 75 BPS.
- $3 for 110 BPS.
- $4 for 135 BPS.
- $5 for 150 BPS.
- $6 for 300 BPS.

- $7 for 600 BPS.
- $8 for 1200 BPS.
- $9 for 1800 BPS.
- $A for 2400 BPS.
- $B for 3600 BPS.
- $C for 4800 BPS.
- $D for 7200 BPS.
- $E for 9600 BPS.
- $F for 19200 BPS.

Address	Description
$D480	Control register

- Bits 0-3 contain the bit rate.

$D481	Command register

- Bit 0 enables or disables the UART.

Wait for status register bit 6 after changes.

$D482	Status register

- Bit 1 indicates a framing error.
- Bit 2 indicates an overrun error.
- Bit 3 indicates receive in progress.
- Bit 4 indicates transmit in progress.
- Bit 6 indicates on (1) or off (0).

$D483	Reserved
$D484	Receive ring buffer head register

- Bits 0-2 contain the position in the buffer.

Address	Description
$D485	Receive ring buffer tail register

- Bits 0-2 contain the position in the buffer.

$D486	Transmit ring buffer head register

- Bits 0-2 contain the position in the buffer.

$D487	Transmit ring buffer head register

- Bits 0-2 contain the position in the buffer.

$D488	Receive ring buffer (8 bytes)
$D490	Transmit ring buffer (8 bytes)

UART 2 REGISTERS

Cody Computer UART 2 is identical in function to UART 1. However, UART 2 is connected to the expansion port.

Address	Description
$D4A0	Control register

- Bits 0-3 contain the bit rate.

$D4A1	Command register

- Bit 0 enables or disables the UART.

Wait for status register bit 6 after changes.

$D4A2	Status register

Address	Description

- Bit 1 indicates a framing error.
- Bit 2 indicates an overrun error.
- Bit 3 indicates receive in progress.
- Bit 4 indicates transmit in progress.
- Bit 6 indicates on (1) or off (0).

$D4A3 Reserved

$D4A4 Receive ring buffer head register

- Bits 0-2 contain the position in the buffer.

$D4A5 Receive ring buffer tail register

- Bits 0-2 contain the position in the buffer.

$D4A6 Transmit ring buffer head register

- Bits 0-2 contain the position in the buffer.

$D4A7 Transmit ring buffer head register

- Bits 0-2 contain the position in the buffer.

$D4A8 Receive ring buffer (8 bytes)

$D4B0 Transmit ring buffer (8 bytes)

APPENDIX B: COLOR CODES

The color codes used by the Cody Computer's Video Interface Device are the same as those from the Commodore VIC-II chip. The actual colors used are from the Propeller NTSC palette.

Code (dec)	Code (hex)	Color
0	$0	Black
1	$1	White
2	$2	Red
3	$3	Cyan
4	$4	Purple
5	$5	Green
6	$6	Blue
7	$7	Yellow
8	$8	Orange
9	$9	Brown
10	$A	Light red
11	$B	Dark gray
12	$C	Gray
13	$D	Light green
14	$E	Light blue
15	$F	Light gray

APPENDIX C: CODY BASIC

This appendix contains a brief reference for Cody BASIC. For more information and examples refer to *Chapter 5: Using Cody BASIC* and *Chapter 6: Advanced Cody BASIC*.

LINE NUMBERS

All Cody BASIC statements in a program must have a line number. A handful of statements and commands can be evaluated immediately at the BASIC prompt, but this is the exception and not the rule.

COMMENTS

Lines beginning with the **REM** (remark) statement will be ignored. Each line incurs a small performance penalty as the statement's token must be processed and the rest of the line skipped over.

VARIABLES

Numeric variables are the letters **A** through **Z**. Each variable can store a 16-bit signed integer from -32768 to 32767 inclusive. When used in certain situations, such as **POKE** statements, numbers are interpreted as their unsigned equivalents to address the entire Cody Computer memory.

A numeric variable is actually the first element in a numeric array of 128 values. A specific element can be accessed by indexing with a number or numeric

expression, such as **A(10)**. Arrays are declared by default in Cody BASIC.

String variables are the letters **A$** through **Z$** (note the trailing dollar sign character). Each string can store up to 255 possible characters and a terminating null character. Strings are declared by default.

Assignment is made using the = operator. Each assignment must be on its own line and the type of the expression must match the type of the variable. A numeric variable must have a numeric expression on the right side, while a string variable must have a string expression on the right side instead.

NUMERIC EXPRESSIONS

Supported numeric operators are **+** (addition), **-** (subtraction), * (multiplication) and **/** (division). Order of operations is obeyed, with mulitplication and division occurring before addition and subtraction.

Expressions can be grouped using **(** (left parenthesis) and **)** (right parenthesis). A leading **-** (unary minus) can be used to obtain the negative of a number or expression.

STRING EXPRESSIONS

The only supported operator for strings is **+** (concatenation). This operator is only supported in very limited circumstances involving explicit string expressions (assignment, **PRINT**, and the right side of expressions in **IF** statements).

RELATIONAL EXPRESSIONS

Relational expressions are only used in **IF** statements. Supported relational operators are **<** (less than), **>** (greater than), **<=** (less than or equal), **>=** (greater than or equal), **=** (equal), and **<>** (not equal).

For numbers a relational expression consists of two numeric expressions with a relational operator. For strings a relational expression consists of a string variable on the left side and a string expression on the right side.

MATHEMATICAL FUNCTIONS

Several mathematical functions are present in Cody BASIC.

- **ABS(*n*)** returns the absolute value of a number.
- **MOD(*m*, *n*)** returns the result of *m* modulo *n*.
- **SQR(*n*)** returns the integer square root of a number.
- **RND()** returns a pseudorandom number.
- **RND(*n*)** seeds the pseudorandom generator with a new value.

BITWISE FUNCTIONS

The typical bitwise operations are implemented as Cody BASIC functions.

- **AND(*m*, *n*)** returns the bitwise-and of two numbers.
- **OR(*m*, *n*)** returns the bitwise-or of two numbers.

- **XOR(*m*, *n*)** returns the bitwise exclusive-or of two numbers.
- **NOT(*n*)** returns the bitwise negation of a number.

STRING FUNCTIONS RETURNING NUMBERS

Some functions that take a string variable argument are used in numeric expressions.

- **ASC(*s$*)** returns the number of the first character in a string variable.
- **VAL(*s$*)** parses a number from the start of a string variable.
- **LEN(*s$*)** returns the number of characters in a string variable.

STRING FUNCTIONS RETURNING STRINGS

Other string functions return strings and are used in string expressions.

- **CHR$(*n*,...,*n*)** converts one or more numbers to string characters.
- **STR$(*n*)** converts a number to its string representation.
- **SUB$(*s$*,*m*,*n*)** returns a substring of length *n* starting at *m*.

FORMATTING FUNCTIONS

Two functions can only be used to control formatting in **PRINT** statements.

- **AT(*x*,*y*)** moves the output to the specified coordinates.

- **TAB(*n*)** moves the output to a particular tab column on screen.

OTHER FUNCTIONS

A couple of functions don't fit into a specific category.

- **PEEK(*n*)** returns the byte at a specific memory address.
- **TI** returns the current time count in jiffies (1/60th of a second).

COMMANDS

Several commands are used to interact with rudimentary Cody BASIC facilities.

- **NEW** clears the program memory and starts a new program.
- **LOAD *m,n*** saves the current program on UART *m* and mode *n*. Use 0 for BASIC programs and 1 for binary programs.
- **SAVE *n*** saves the current program on UART *n*.
- **RUN** runs the current BASIC program starting at the first line.
- **LIST** lists the program.
- **LIST *m*** lists the program starting with a particular line.
- **LIST *m,n*** lists the program between two line numbers.

CONTROL STATEMENTS

Control statements manage the flow through a Cody BASIC program.

- **IF *r* THEN *s*** evaluates statement *s* if relational expression *r* is true.
- **GOTO *n*** jumps to a particular line in the program.
- **GOSUB *n*** calls a particular line with the intention of **RETURN**ing.
- **RETURN** returns to the line after the last **GOSUB**.
- **FOR *i*=*m* TO *n*** loops *i* from *m* to *n* with a matching **NEXT**.
- **NEXT** starts the next loop with the matching **FOR**.
- **END** exits the current program.

INPUT AND OUTPUT STATEMENTS

Cody BASIC has several statements for structured input and output.

- **INPUT v,...,v** reads one-per-line numeric or string values into one or more variables *v*.
- **PRINT** prints a blank line.
- **PRINT *e*,...,*e*** prints one or more numeric or string expressions. The statement will move on to the next line unless **;** (semicolon) is at the end.
- **OPEN *m*,*n*** redirects future **INPUT** and **PRINT** statements to UART *m* with bit rate specifier *n*.
- **CLOSE** closes a UART and directs back to the keyboard and screen.

The most recent keyboard and joystick matrix scans performed by the BASIC interpreter can be read from zero page addresses 16 through 23. The input prompt character can be changed by changing zero page address 14.

DATA STATEMENTS

Cody BASIC supports a limited form of **DATA** statements for literals. Data will be read from each statement in the program starting at the beginning and going to the end.

- **DATA n,..,n** declares one or more numeric literals separated by commas.
- **READ v,..,v** reads one or more literals from **DATA** into number variables.
- **RESTORE** moves the data location back to the beginning of the program.

OTHER STATEMENTS

Some statements don't easily fit into a specific category.

- **POKE *m,n*** pokes byte *n* into memory address *m*.
- **SYS *n*** calls address *n* in assembly language. Values for registers **A**, **X**, and **Y** can be passed in the first three zero page variables.

ERRORS

Cody BASIC has limited error handling inspired by Tiny BASIC.

- **LOGIC** errors occur when a statement was syntactically valid but wrong in context.
- **SYNTAX** errors occur when a statement could not be correctly parsed.
- **SYSTEM** errors occur when a statement fails because of low-level problems.

APPENDIX D: CODSCII TABLE

The CODSCII character set is the default character set used by the Cody Computer and Cody BASIC. It's an extended ASCII character set with the top 128 values used for Commodore PETSCII characters and custom control codes for colors and terminal operations.

Dec	Hex	Description	Image
0	$00	Null	
1	$01	Start of heading	
2	$02	Start of text	
3	$03	End of text	
4	$04	End of transmission	
5	$05	Enquiry	
6	$06	Acknowledge	
7	$07	Bell	
8	$08	Backspace	
9	$09	Horizontal tab	
10	$0A	Line feed	
11	$0B	Vertical tab	
12	$0C	Form feed	
13	$0D	Carriage return	
14	$0E	Shift out	
15	$0F	Shift in	
16	$10	Data link escape	

17	$11	Device control 1 (XON)	
18	$12	Device control 2	
19	$13	Device control 3 (XOFF)	
20	$14	Device control 4	
21	$15	Negative acknowledge	
22	$16	Synchronous idle	
23	$17	End of transmission block	
24	$18	Cancel	
25	$19	End of medium	
26	$1A	Substitute	
27	$1B	Escape	
28	$1C	File separator	
29	$1D	Group separator	
30	$1E	Record separator	
31	$1F	Unit separator	
32	$20	Whitespace	
33	$21	Exclamation mark	
34	$22	Double quotes	
35	$23	Hash symbol	
36	$24	Dollar sign	
37	$25	Percent	
38	$26	Ampersand	
39	$27	Single quote	
40	$28	Left parenthesis	
41	$29	Right parenthesis	

42	$2A	Asterisk	
43	$2B	Plus	
44	$2C	Comma	
45	$2D	Minus	
46	$2E	Period	
47	$2F	Slash	
48	$30	Zero	
49	$31	One	
50	$32	Two	
51	$33	Three	
52	$34	Four	
53	$35	Five	
54	$36	Six	
55	$37	Seven	
56	$38	Eight	
57	$39	Nine	
58	$3A	Colon	
59	$3B	Semicolon	
60	$3C	Less than	
61	$3D	Equal	
62	$3E	Greater than	
63	$3F	Question mark	
64	$40	At symbol	
65	$41	Uppercase A	
66	$42	Uppercase B	

67	$43	Uppercase C	
68	$44	Uppercase D	
69	$45	Uppercase E	
70	$46	Uppercase F	
71	$47	Uppercase G	
72	$48	Uppercase H	
73	$49	Uppercase I	
74	$4A	Uppercase J	
75	$4B	Uppercase K	
76	$4C	Uppercase L	
77	$4D	Uppercase M	
78	$4E	Uppercase N	
79	$4F	Uppercase O	
80	$50	Uppercase P	
81	$51	Uppercase Q	
82	$52	Uppercase R	
83	$53	Uppercase S	
84	$54	Uppercase T	
85	$55	Uppercase U	
86	$56	Uppercase V	
87	$57	Uppercase W	
88	$58	Uppercase X	
89	$59	Uppercase Y	
90	$5A	Uppercase Z	
91	$5B	Left bracket	

92	$5C	Backslash	
93	$5D	Right bracket	
94	$5E	Caret	
95	$5F	Underscore	
96	$60	Backquote	
97	$61	Lowercase a	
98	$62	Lowercase b	
99	$63	Lowercase c	
100	$64	Lowercase d	
101	$65	Lowercase e	
102	$66	Lowercase f	
103	$67	Lowercase g	
104	$68	Lowercase h	
105	$69	Lowercase i	
106	$6A	Lowercase j	
107	$6B	Lowercase k	
108	$6C	Lowercase l	
109	$6D	Lowercase m	
110	$6E	Lowercase n	
111	$6F	Lowercase o	
112	$70	Lowercase p	
113	$71	Lowercase q	
114	$72	Lowercase r	
115	$73	Lowercase s	
116	$74	Lowercase t	

117	$75	Lowercase u	
118	$76	Lowercase v	
119	$77	Lowercase w	
120	$78	Lowercase x	
121	$79	Lowercase y	
122	$7A	Lowercase z	
123	$7B	Left brace	
124	$7C	Pipe	
125	$7D	Right brace	
126	$7E	Tilde	
127	$7F	Unused/Reserved	
128	$80	Pound sign	
129	$81	Up arrow	
130	$82	Left arrow	
131	$83	Horizontal line	
132	$84	Spade	
133	$85	Vertical line	
134	$86	Horizontal line	
135	$87	Horizontal line up 1	
136	$88	Horizontal line up 2	
137	$89	Horizontal line down 1	
138	$8A	Vertical line left 1	
139	$8B	Vertical line duplicate	
140	$8C	Quarter circle bottom left	
141	$8D	Quarter circle top right	

142	$8E	Quarter circle top left	
143	$8F	Box bottom left corner	
144	$90	Diagonal down	
145	$91	Diagonal up	
146	$92	Box top left corner	
147	$93	Box top right corner	
148	$94	Dot	
149	$95	Horizontal line down 2	
150	$96	Heart	
151	$97	Vertical line left 1 duplicate	
152	$98	Quarter circle bottom right	
153	$99	X	
154	$9A	Dot with hole	
155	$9B	Club	
156	$9C	Vertical line duplicate	
157	$9D	Diamond	
158	$9E	Cross	
159	$9F	Dotted left	
160	$A0	Vertical line duplicate	
161	$A1	Pi	
162	$A2	Filled diagonal top right	
163	$A3	Blank	
164	$A4	Filled box left	
165	$A5	Filled box bottom	
166	$A6	Horizontal line top	

167	$A7	Horizontal line bottom	
168	$A8	Vertical line left	
169	$A9	Dotted square	
170	$AA	Vertical line right	
171	$AB	Dotted bottom	
172	$AC	Diagonal filled top left	
173	$AD	Vertical line right duplicate	
174	$AE	T right	
175	$AF	Filled quarter box bottom right	
176	$B0	Box top right	
177	$B1	Box bottom left	
178	$B2	Horizontal line bottom duplicate	
179	$B3	Box bottom right	
180	$B4	T up	
181	$B5	T down	
182	$B6	T left	
183	$B7	Vertical line left duplicate	
184	$B8	Filled left half duplicate	
185	$B9	Filled right half duplicate	
186	$BA	Horizontal line top	
187	$BB	Horizontal partial fill top	
188	$BC	Horizontal partial fill bottom	
189	$BD	Box bottom right corner	
190	$BE	Filled box lower left	
191	$BF	Filled box top right	

192	$C0	Box top left	
193	$C1	Filled box top left	
194	$C2	Checkered square	
195	$C3	Unused/Reserved	
196	$C4	Unused/Reserved	
197	$C5	Unused/Reserved	
198	$C6	Unused/Reserved	
199	$C7	Unused/Reserved	
200	$C8	Unused/Reserved	
201	$C9	Unused/Reserved	
202	$CA	Unused/Reserved	
203	$CB	Unused/Reserved	
204	$CC	Unused/Reserved	
205	$CD	Unused/Reserved	
206	$CE	Unused/Reserved	
207	$CF	Unused/Reserved	
208	$D0	Unused/Reserved	
209	$D1	Unused/Reserved	
210	$D2	Unused/Reserved	
211	$D3	Unused/Reserved	
212	$D4	Unused/Reserved	
213	$D5	Unused/Reserved	
214	$D6	Unused/Reserved	
215	$D7	Unused/Reserved	
216	$D8	Unused/Reserved	

217	$D9	Unused/Reserved	☐
218	$DA	Unused/Reserved	☐
219	$DB	Unused/Reserved	☐
220	$DC	Unused/Reserved	☐
221	$DD	Unused/Reserved	☐
222	$DE	Clear screen	☐
223	$DF	Reverse field	☐
224	$E0	Background black	☐
225	$E1	Background white	☐
226	$E2	Background red	☐
227	$E3	Background cyan	☐
228	$E4	Background purple	☐
229	$E5	Background green	☐
230	$E6	Background blue	☐
231	$E7	Background yellow	☐
232	$E8	Background orange	☐
233	$E9	Background brown	☐
234	$EA	Background light red	☐
235	$EB	Background dark gray	☐
236	$EC	Background gray	☐
237	$ED	Background light green	☐
238	$EE	Background light blue	☐
239	$EF	Background light gray	☐
240	$F0	Foreground black	☐
241	$F1	Foreground white	☐

242	$F2	Foreground red	☐
243	$F3	Foreground cyan	☐
244	$F4	Foreground purple	☐
245	$F5	Foreground green	☐
246	$F6	Foreground blue	☐
247	$F7	Foreground yellow	☐
248	$F8	Foreground orange	☐
249	$F9	Foreground brown	☐
250	$FA	Foreground light red	☐
251	$FB	Foreground dark gray	☐
252	$FC	Foreground gray	☐
253	$FD	Foreground light green	☐
254	$FE	Foreground light blue	☐
255	$FF	Foreground light gray	☐