Running Linux on the Sega Dreamcast

Bill Gatliff (Sept. 24, 2001)

Looking for a low−cost way to get started with embedded Linux? Or a fun weekend project? In this detailed how−to article, Bill Gatliff explains everything you need to do to install Linux on a Sega Dreamscast gaming console. Even the necessary Linux kernel, bootloader, and utility kernel patches are included and available for download.

Story navigation . . .

Introduction

One of the more challenging aspects of learning about embedded Linux is the scarcity of cheap, compatible hardware. Sure, Linux runs fine on a personal computer, but does being able to run Linux on your workstation mean you can call yourself an embedded Linux guru?

Hardly. You may not believe me now, but you will shortly.

What's needed to really explore Linux as an embedded operating system is a well−documented, inexpensive and readily available hardware platform that isn't based on an Intel x86−compatible microprocessor. By excluding PCs, the list of candidates becomes: PDAs, internet appliances, and gaming consoles.

Yes, gaming consoles: that's the ticket. To encourage consumers to buy the games, gaming consoles are often sold at or below cost. And one of the cheapest ones around at the moment is the Sega Dreamcast, by virtue of the fact that it has been recently discontinued by Sega as they make way for their new products.

But will the Dreamcast run Linux? Sure! In fact, it already does.

The rest of this article tells you how to get it running on *your* Dreamcast.

How Linux works on the Dreamcast

Before we can discuss how to run Linux on the Dreamcast, you have to know a little bit about how to get Linux *inside* the Dreamcast in the first place. As it turns out, the magic is in the format of the CD you insert into the Dreamcast before powering up. In other words: no tools required!

The Sega Dreamcast

The Dreamcast sports a Hitachi SH7750 CPU running at 200 MHz, with 16MB of memory. It can drive either a television (PAL or NTSC) display or, with an adapter, a VGA monitor. You can still find Dreamcasts at your local toy store, and for around $100 USD you can get one *plus* the optional keyboard and mouse peripherals. You can also find its discontinued ethernet adapter (Sega calls it a Broadband Adapter, or BBA) listed on your favorite internet auction site from time to time.

Getting Linux into the Dreamcast

On powerup, the Dreamcast's onboard bootloader firmware expects to see a CD−R (*not* a CD−RW) containing a minimum of two recording sessions. The first session is assumed to be an audio track, which the bootloader ignores. The second session must be a CD/XA data track (mode 2, form 1) containing a standard ISO9660 filesystem, and must include a bootstrap information structure commonly referred to as `IP.BIN` in the first 16 sectors of that filesystem.

The `IP.BIN` data structure contains the name of the file the bootloader is to launch at the end of the boot process. IP.BIN also includes metadata that identifies the hardware on which the CD's programs are designed to run (Dreamcast vs. other Sega products), area symbols that control geographic compatability of the program (NTSC vs. PAL video output, for example), the list of peripherals required by the program (which controller, game pack, etc.), and a CRC.

Once the Dreamcast's bootloader reads and validates the information in `IP.BIN`, it loads the contents of the named file from the ISO9660 filesystem into memory and gives complete control of the system over to it. In our case, the program loaded will our Linux operating system and related programs and data; other possibilities would be a small monitor that knows how to download another application from the Dreamcast's

serial or ethernet port; an RTOS like eCos; or −− work with me here −− a game of some kind. Or so I've heard. (*The author has no games for his Dreamcast. Really. When it's playtime, he heads upstairs to his three self−propelled gaming units. They are loads more fun, but also considerably more expensive to feed and clothe. Which makes the price of the Dreamcast an even better value.*)

The Dreamcast's system firmware also provides some BIOS−like functionality for system operations like reading from the GD−ROM drive (a modified CD−ROM drive that can hold 1GB of data), getting system information, and accessing onboard flash memory; code to utilize these features in the Dreamcast's Linux kernel is currently under development, and will probably be ready for use by the time you read this article.

And then what happens?

Now that we know how to get an executable image into the Dreamcast, we need to know exactly how to use this capability to boot and run Linux. To answer this question, we need to cover a bit more background.

Traditionally, Linux kernels are not self−booting. Instead, they depend on a bootloader to establish a stable system state, gather information about the host platform, and then feed this information to the kernel. Many Linux setups also include decompression code in the bootloader, which speeds the overall boot process by reading a compressed (instead of full−sized) kernel image from slow, cheap media like floppy disks. On a typical Linux PC setup, the bootloader is called LILO or Grub . In an embedded setting, the bootloader is called whatever you choose to name it.

Briefly restated, the customary boot process in a Linux−based environment goes like this . . .

1. A bootloader is given initial control of the host system from the host's BIOS or boot firmware, or the bootloader *is* the system's boot firmware.

2. The bootloader drags the kernel image into memory and decompresses it.

3. The bootloader then jumps into the decompressed memory image, which contains a Linux operating system kernel.

4. The kernel initializes itself, mounts a root directory, loads device drivers, initializes hardware, and starts running applications.

Now that we know all of this, the contents of the executable image on the CD become a little clearer. The image must contain a bootloader and the Linux kernel, and we must arrange the image so that the bootloader gets control at the end of the Dreamcast's boot process, finds the Linux kernel, and transfers control to it.

Linux vs. applications

You might think that getting the Linux kernel booting and running is the end of the story. It turns out, however, that it is only the beginning; once the kernel is running, it needs applications to tell it what to do. If this statement sounds strange, then you probably have missed one of the fundamental ways that general−purpose, desktop−style operating systems differ from their embedded counterparts.

On the desktop, an operating system only coordinates interactions between applications. In embedded systems, the operating system and application code are often one in the same, because the two are bound so tightly together that distinguishing between them is impossible, or at least impractical.

Linux is a general−purpose, desktop−oriented operating system. As such, we must have applications available for it to run. We will produce such an application −− a shell −− towards the end of this article.

Linux vs. filesystems

If you wander through Linux's startup code (*this could be the subject of a future article, if anyone expresses any interest*), you end up with the kernel looking for an executable file called "init" to load into memory and run. This not only implies that we need a program named `init` , it also means that we need a filesystem to retrieve it from.

You are probably aware that the Dreamcast does not include a hard disk, which would be an obvious place to put files. You may not be aware, however, that most PC workstations also lack hard disks −− at least until the operating system gets around to initializing the machine's IDE or SCSI controller hardware, that is. So Linux must already have a means of emulating a filesystem at startup, or it wouldn't be of much use in a workstation−class personal computer. Of course, we all know that it is.

To emulate a filesystem when no filesystem−oriented hardware is avalable, Linux uses a *ramdisk* : a memory structure that masquerades as a disk partition. A workstation bootloader loads a ramdisk image from the computer's disk hardware at boot time, and passes its address to the Linux kernel. The kernel reads device drivers for the workstation's disk controllers from the ramdisk, then begins talking to the physical disk drives. Once the "real" disk interface is working, the memory allocated for the ramdisk is discarded.

A ramdisk can provide all the disk−oriented facilities that Linux requires. (*Although a ramdisk device can facilitate writing of data files, those files are lost at system shutdown because the original ramdisk image is re−read from the boot media at the next startup.*) Consequently, there is no need for a hard disk at all if the host system has enough ramdisk memory to store all the programs and device drivers that the system needs. This is the approach we will take on the Dreamcast.

What you need to build Linux for the Dreamcast

You must have root access on a true Linux or other Unix−like workstation in order to build your own Linux−on−Dreamcast system, because the setup process requires construction of special files called *device nodes* that have no equivalent representation in a non−unix−like environment.

You also need a CD−R burner that can write a multisession CD using the CD/XA data track (mode 2, form 1) format. Support for this configuration is widespread in all but the least expensive CD−R burners and programming software. Note that the Dreamcast's GD−ROM drive cannot read a CD−RW disk, but it can read a CD−R disk produced by a CD−RW burner.

Finally, you need the list of files found in Figure 1. The standard GNU distributions are available from the Free Software Foundation's website or a mirror site , and other other software is available from the locations mentioned in the figure. The extensive list of patches required is simply my way of helping to automate the configuration and setup process.

- binutils−2.11.2.tar.gz −− The linker, assembler, and object management utilities.
- gcc−3.0.1.tar.gz −− The GNU Compiler Collection. Contains the C/C++ compiler.
- glibc−2.2.4.tar.gz −− The GNU C runtime library.
- busybox−0.60.1.tar.gz −− Small, embedded versions of common Unix utilities.
- kernel−sh−linux−dreamcast.tar.gz −− Linux kernel for the Dreamcast.
- sh−boot−20010831−1455.tar.gz −− Linux bootloader for the Dreamcast.
- binutils−2.11.2−sh−linux.diff −− Patches for *binutils*.
- gcc−3.0.1−sh−linux.diff −− Patches for *gcc*.
- glibc−2.2.4−sh−linux.diff −− Patches for *glibc*.
- busybox−0.60.1−sh−linux.diff −− Patches for *Busybox*.
- kernel−sh−linux−dreamcast.diff −− Patches for the Linux kernel.
- sh−boot−20010831−1455.diff −− Patches for *sh−boot*.

Figure 1: Software needed to run Linux on the Dreamcast.

Building the tools

The first step in the process of getting Linux running on the Dreamcast is to construct the tools we need, including a cross assembler, linker, compiler, and a C runtime library. In the next section we will use these tools to build the operating system, and a basic shell application that Linux will run at the end of its boot process. Finally, we'll organize everything into a ramdisk image and burn a CD.

The process is long and involved, but highly instructive.

Building a cross assembler, linker, and bootstrap compiler

We'll start by building a cross assembler and linker, and then a *bootstrap compiler* : a minimal compiler that can be used to build runtime libraries and operating system kernels, but not generic applications. We can't build a complete compilation environment all at once, because many of the header files needed to do so come from the C runtime library, and don't exist until the runtime library itself is built.

Log in as the root user, and follow the script shown in Figure 2. The first steps in the figure set up some environment variables to save typing later, and to make sure that `/usr/local/bin` is in the `PATH` . The source code for the *binutils* package is decompressed, and then patched with minor changes that make it more specific to the Dreamcast's microprocessor. The *binutils* package is then configured, compiled and installed; the resulting executables end up in `/usr/local/bin` , with names like `sh4−linux−as` and `sh4−linux−ld` .

The same process is then repeated for the bootstrap compiler, only with slightly different arguments that reflect the fact that we are not building a complete compiler setup yet. The bootstrap compiler's executable is called `sh4−linux−gcc`, and like the *binutils* package, is installed in `/usr/local/bin`.

```
# export TARGET=sh4−linux
# export PREFIX=/usr/local
# export PATH=${PATH}:${PREFIX}/bin

# tar xzf binutils−2.11.2.tar.gz
# patch −p0 < binutils−2.11.2−sh−linux.diff
# mkdir −p build−binutils && cd build−binutils
# ../binutils−2.11.2/configure −−target=$TARGET \
  −−prefix=$PREFIX
# make all install
# cd ..

# tar xzf gcc−3.0.1.tar.gz
# patch −p0 < gcc−3.0.1−sh−linux.diff
# mkdir −p build−gcc && cd build−gcc
# ../gcc−3.0.1/configure \
  −−target=$TARGET −−prefix=$PREFIX \
  −−without−headers −−with−newlib \
  −−disable−shared −−enable−languages=c
# make all−gcc install−gcc
# cd ..
```

Figure 2: Commands to build *binutils* and a bootstrap *gcc*

Configure the kernel sources

The next step in the toolchain construction process is to configure Linux kernel header files so that the runtime library's build process can extract important bits of information from them during its build process. The idea is to run the equivalent of the `configure` command on the kernel source code, which you accomplish by using the procedure is shown in Figure 3. Run these commands as they appear in the figure, but don't change any of the settings in the menu that appears: the kernel is already properly configured. Simply exit `menuconfig` by selecting `Exit` with the right arrow key on your keyboard, then press ENTER .

```
# tar xzf kernel−sh−linux−dreamcast.tar.gz
# patch −p0 < kernel−sh−linux−dreamcast.diff
# cd kernel
# make ARCH=sh CROSS_COMPILE=sh4−linux− menuconfig
# cd ..
```

Figure 3: Configuring the kernel sources.

In the instructions, the trailing dash at the end of `CROSS_COMPILE` is not a mistake: the `CROSS_COMPILE` macro is used as a prefix for the tools invoked by `make`; without the dash at the end, the macro won't work.

Building a runtime library

Now that we have a bootstrap compiler and properly−configured kernel, we can build a runtime library environment and header files. The runtime library we will use is GNU's *glibc*; this library includes familiar functions like printf() , but it also includes the dynamic linker (Linux's equivalent of Win32's DLLs) and several other programs. We can get by without these programs, however, so the steps needed to set them up will be the subject of a future article.

The procedure for building *glibc* is shown in Figure 4. The first commands uncompress and patch the library, then copy the Linux kernel header files into their proper locations. In contrast to the previous procedures, we actually invoke `make` twice when building glibc: the first invocation builds the library, but does not install it; the second step installs the library, specifying in detail the installation locations of the library's various components.

In the interest of saving some time, the `touch` command used between the two `make` invocations tricks glibc into thinking it has properly built several programs that we don't need. And finally, the `echo` command writes a linker command file called `libc.so` without path information, which (perhaps nonintuitively) enables the compiler to properly locate the installed libraries.

If you rebooted your computer or logged out since you built *binutils* and the bootstrap *gcc*, you will need to restore the values of `PATH`, `TARGET` and `PREFIX` to the values used in previous steps before building *glibc*.

Glibc is a substantial, complicated body of code, and you will notice that it takes a long time to build even on fairly powerful hardware. After you get the process going, this would be a good time to take a break.

```
# tar xzf glibc−2.2.4.tar.gz
# patch −p0 < glibc−2.2.4−sh−linux.diff
# mkdir −p build−glibc && cd build−glibc

# mkdir −p ${PREFIX}/${TARGET}/include
# cp −r ../kernel/include/linux \
   ${PREFIX}/${TARGET}/include
# cp −r ../kernel/include/asm−sh \
   ${PREFIX}/${TARGET}/include/asm

# CC=sh4−linux−gcc ../glibc−2.2.4/configure \
   −−host=$TARGET −−prefix=$PREFIX \
   −−disable−debug −−disable−profile \
   −−disable−sanity−checks \
   −−with−headers=${PREFIX}/${TARGET}/include

# make
# touch iconv/iconv_prog login/pt_chown
# make install_root=${PREFIX}/${TARGET} \
   prefix="" install
# echo "GROUP ( libc.so.6 libc_nonshared.a )" \
    > ${PREFIX}/${TARGET}/lib/libc.so
# cd ..
```

Figure 4: Instructions to build *glibc*

Rebuild the cross compiler

With properly installed header files and a runtime library, we can now build a complete c/c++ cross compiler. The commands to do so are in Figure 5. We are reusing the source tree from the bootstrap compiler, so we skip the familiar untar−plus−patch steps.

```
# mkdir −p build−gcc2 && cd build−gcc2
# ../gcc−3.0.1/configure −−target=$TARGET \
   −−prefix=$PREFIX −−enable−languages=c,c++
# make all install
# cd ..
```

Figure 5: Instructions to build the complete cross compiler

Building a Dreamcast Linux kernel

Now that we have a complete cross development toolchain, it's time to put the tools to work. As I mentioned before, I have pre−configured our Linux kernel sources so that they are already set up for the Dreamcast, so all that's needed now is to run the commands in Figure 6 to actually compile and link the kernel. The result is a file called `zImage`, located in `kernel/arch/sh/boot/` ; this file contains a compressed kernel image, wrapped by a short procedure that knows how to decompress the kernel into memory.

Later on, after you have successfully booted this kernel as−is, feel free to use `menuconfig` to adjust kernel settings. Recompile using the procedure in Figure 6, and observe the results by booting (or not, as the case may be) the new kernel on the Dreamcast. One useful exercise is to see just how small a kernel image you can produce that still contains the functionality you need.

```
# cd kernel
# make ARCH=sh CROSS_COMPILE=sh4−linux− \
  clean dep zImage
# cd ..
```

Figure 6: Building the Dreamcast Linux kernel

Building a Dreamcast Bootloader

With a successfully compiled kernel image, it would seem that the next step would be to produce a bootloader for loading the kernel image into the Dreamcast's memory at startup.

As it turns out, however, we aren't ready to do that just yet. Recall that the Dreamcast's firmware loads a single executable image into RAM at the end of the boot process, and unless that application knows how to work with the Dreamcast's GD−ROM drive there is no way to load additional data from the disk. The bootloader we will use does not know how to operate the GD−ROM drive, so we must load everything we will need at runtime in one fell swoop: the bootloader, the kernel image, and the ramdisk image.

Once everything is loaded, the bootloader needs to dissect memory back into a distinct kernel and ramdisk image, which requires the bootloader to know their exact sizes prior to startup. The only way to determine this is to actually build the kernel (which we have now done) and ramdisk image, compute their sizes, and then provide this information to the bootloader's source code during its build process.

So, we will build some applications, populate a ramdisk image with them, and then return to build the bootloader/kernel/ramdisk image for the Dreamcast's boot firmware to load into memory.

Building an application

Perhaps the most important application in a basic Linux system is a command shell. Without a shell, it is impossible to interactively instruct the operating system to load other programs, mount remote directories, or simply probe the system's setup to troubleshoot problems or just see what's going on.

If you don't have a keyboard for your Dreamcast, then you may not find the availability of an interactive shell to be all that interesting: you don't have any way to type commands! Build a shell anyway, however, because a shell can also be used to run command scripts that you include on the ramdisk. And once you have seen the procedure for getting a shell up and running on the Dreamcast, you will be able to replace the shell with any other program you want to run.

The procedure in Figure 7 describes how to build BusyBox for the Dreamcast. In addition to a basic shell facility, Busybox also includes small versions of several other useful utilities, including the `mount`, `ls` and `modprobe` programs.

The first steps in Figure 7 create a directory called `initrd`, which will contain the contents of the initial ramdisk. (initrd *is the traditional name for an initial ramdisk.*) The `PREFIX` parameter passed to `make` causes Busybox to install itself properly relative to the location of this directory. The `DOSTATIC` setting tells `sh4−linux−gcc` to not use shared libraries for Busybox, which is what we want because we have not installed a dynamic linker.

```
# mkdir −p initrd
# export INITRD=`pwd`/initrd

# tar xzf busybox−0.60.1.tar.gz
```

```
# patch −p0 < busybox−0.60.1−sh−linux.diff

# cd busybox−0.60.1
# make CROSS=sh4−linux− DOSTATIC=true \
    CFLAGS_EXTRA="−I ${PREFIX}/${TARGET}/include" \
    PREFIX=${INITRD} clean all install

# cd ..
```

Figure 7: Commands to build and install Busybox

Like all the other software we have used so far, Busybox is highly configurable. Its most important settings can be found in `Config.h` and `libbb/libbb.h`.

Make device nodes on the initial ramdisk

Since the contents of the `${INITRD}` directory will be the contents of Linux's root directory on the Dreamcast, it has to contain everything the kernel could possibly need at run time. In addition to a shell or some other application, then, we must also provide *device nodes* so that applications can communicate with Linux's device drivers.

On the Dreamcast, there is only one absolutely essential device node: the `/dev/console` node. Without this node, text−mode applications cannot communicate with the console device, which means you cannot see any text output on the Dreamcast's display. Use the commands in Figure 8 to create the console device node.

```
# mkdir −p ${INITRD}/dev
# mknod ${INITRD}/dev/console c 5 1
```

Figure 8: Commands to create the `/dev/console` device node

Creating a ramdisk image

Now that we have populated a directory structure to look like our ramdisk, it is time to take an image of that directory structure so that we can bind it to the Dreamcast's bootloader. The commands in Figure 9 use the *loop* device to create a compressed snapshot of the `${INITRD}` directory in the file `initrd.bin`. The file `initrd.img` is the uncompressed snapshot.

```
# dd if=/dev/zero of=initrd.img bs=1k count=4096
# mke2fs −F −vm0 initrd.img
# mkdir initrd.dir
# mount −o loop initrd.img initrd.dir
```

```
# (cd initrd ; tar cf − .) | (cd initrd.dir ; tar xvf −)
# umount initrd.dir
# gzip −c −9 initrd.img > initrd.bin
```

Figure 9: Creating a compressed ramdisk image

Building the bootloader

The Dreamcast bootloader is part of a collection of Hitachi SH bootloaders called *sh−boot*. The Dreamcast−specific code is buried deep inside the sh−boot directory tree, in the subdirectory `tools/dreamcast/` , and includes both a `Makefile` for building the bootloader image itself, and a script called `roast.sh` that can be used to build a bootable Dreamcast CD from the bootloader image. Sh−boot also includes a utility called `scramble` , which "scrambles" the contents of the Dreamcast CD's ISO9660 data into a primitive encryption format expected by the Dreamcast's boot firmware.

The procedure for building the Dreamcast bootloader image is shown in Figure 10. Do those commands now.

```
# tar xzf sh−boot−20010831−1455.tar.gz
# patch −p0 < sh−boot−20010831−1455.diff
# cd sh−boot/tools/dreamcast
# cp ../../../kernel/arch/sh/boot/zImage ./zImage.bin
# cp ../../../initrd.bin .
# make scramble kernel−boot.bin
```

Figure 10: Commands to make a bootloader image

The bootloader image is the file `kernel−boot.bin`. This file contains the bootloader itself, plus the compressed Linux kernel and initial ramdisk images.

Building the Dreamcast boot CD

We're in the home stretch now! All that's left to do is to burn `kernel−boot.bin` onto a CD, move the CD to the Dreamcast, hit the POWER button, and watch your new kernel boot.

As I mentioned in the previous section, the *sh−boot* package contains a script called `roast.sh` that automates the CD burning process. This script renames `kernel−boot.bin` to `1ST_READ.BIN` (the startup file name as recorded in `IP.BIN`), invokes `mkisofs` to construct an ISO9660 filesystem image, and then uses `cdrecord` to actually burn the CD.

To make the script work, you must set the value of `CDRECORD` at the top of the script file to the identity of your CD−R burner. Use the commands in Figure 11 to get the ID, then change the `1,0,0` in `roast.sh` to

the appropriate value if necessary.

The `cdrecord` program relies on Linux's `ide−scsi` device driver, hence the call to `modprobe` at the start of the procedure. You will almost certainly need to exit X−Windows if you are running it, because most Linux X−Window managers install device drivers that prevent `ide−scsi` from initializing properly. If the call to `modprobe` fails with an error indicating that the `ide−scsi` device cannot be installed, exit from X−Windows and try again.

```
# modprobe ide−scsi
# cdrecord −scanbus
```

scsibus1: 1,0,0 100) 'PHILIPS ' 'PCRW804 ' ' 1.5' Removable CD−ROM

Figure 11: Finding your CD−R burner's device id

With `CDRECORD` in `roast.sh` set properly, insert a CD−R and run the script using the commands in Figure 12. Sit back, relax and congratulate yourself while your Dreamcast Linux CD cooks. Once that's done, pop the CD into the Dreamcast, apply power, and in a few seconds you should see Linux boot and run the Busybox shell application.

```
# ./roast.sh kernel−boot.bin
```

Figure 12: Burning the CD

Thoughts for the future

This article covers a lot in a short space. In addition to building a Hitachi SH cross compiler toolchain, you also built the GNU C runtime library, a Linux kernel, and a ramdisk image containing the Busybox shell application. And as if that wasn't enough, you put everything together into a bootloader image and burned a CD that booted in the Dreamcast.

Where do you want to go next? If you think that an embedded Linux system lies in your future, then you owe it to yourself to play with Linux on an inexpensive, non−PC setup like the Dreamcast as much as possible before you need to make a living at it. One suggestion is to replace Busybox with an application of your own design, compiled and statically linked with the Hitachi SH compilation tools. Another idea is to gather up several other bits of software from the Internet, and try to make your Dreamcast look and act as much like

your Linux PC workstation possible.

If you stick with the Dreamcast, then before long you will want to invest in a Sega Broadband Adapter, which will give your Dreamcast an Internet−ready connection of its own (*the author routinely checks his email and browses the Web from his Dreamcast console, just because he can*), as well as eliminate the need to burn CD−R's. Or maybe you want to dig out your soldering iron and a handful of RS232 level conversion chips, and put a serial port on your Dreamcast to use for downloading code and who−knows−what else. Links to information on these and other ideas can be found in the section called *Resources*, at the end of this article.

Far from "just a gaming console", the inexpensive and flexible Dreamcast hardware platform is just the thing for an embedded Linux guru−in−training. Hopefully, this article is just your beginning.

Resources . . .

- http://www.billgatliff.com −− Additional information on embedded Linux, and GNU programming for embedded systems.
- http://www.linuxdc.org −− The most organized of several sites dedicated to running Linux on the Dreamcast. This site hosts an IRC channel (irc.openprojects.net#linuxdc), provides several HOWTOs related to the Dreamcast, and hosts the source and patch distributions for this article.
- http://linuxsh.sourceforge.net −− The official repository of Linux kernels and tools for the Hitachi SH microprocessor, including the kernel for the Dreamcast.
- http://www.m17n.org −− The Organization for Multilingulization. Provides a Dreamcast Linux distribution CD via their website, which is thoroughly under−documented and poorly supported. Also provides LinuxSH−specific GNU tool distributions, some of which were used in this article.
- http://mc.pp.se/dc −− Marcus Comstedt's website, which contains lots of engineering−level information about the Dreamcast. Also includes a HOWTO for building a Dreamcast serial cable.
- http://www.cerc.utexas.edu/~andrewk/dc/ −− Andrew Kieschnick's website, which includes programs for loading code into the Dreamcast via the Broadband or serial cable adapters.
- http://www.linuxfromscratch.org −− The Linux From Scratch website. Provides details on how to build a complete Linux setup from source code.

Acknowledgements . . .

The author wishes to acknowledge the following individuals . . .

- M. R. Brown and Karl Trygve Kalleberg, for their help and support in preparing and reviewing this article.
- Marcus Comstedt, for his reverse−engineering of the Dreamcast hardware.

About the Author: Bill Gatliff is an independent consultant with almost ten years of embedded development and training experience. He specializes GNU−based embedded development, and in using and adapting GNU tools to meet the needs of difficult development problems. He welcomes the opportunity to participate in projects of all types. Bill is a Contributing Editor for Embedded Systems Programming Magazine , a member of the Advisory Panel for the Embedded Systems Conference , maintainer of the Crossgcc FAQ, creator of

the gdbstubs project , and a noted author and speaker. Bill welcomes feedback and suggestions. Contact information is on his website .

Related stories:

- The Linux on Dreamcast Project
- GNU/Linux on the SEGA Dreamcast
- LinuxSH −− Linux on the Hitachi SuperH processor
- Metro Link ports Micro−X to Sony PlayStation
- Sony survey guages interest in Linux for PlayStation2
- Enter Runix: Linux for the PlayStation
- Sony tests Linux on PlayStation 2

PDF editing and generation by Terry 'Mongoose' Hendrix (Oct. 07, 2001)
Email stu7440@westga.edu for PDF corrections.