Running Linux on a Two-Chip STM32F4 Design

Electronic Design
Vladimir Khusainov
Tue, 2015-02-24 15:56

With the emergence of STMicroelectronics' powerful STM32F42X/43X microcontrollers (MCUs), the topic of running Linux, in its uClinux form, on the STM32 devices is as hot as ever. In the MCU world, embedded designers have historically been limited to using an RTOS or even “bare-metal” firmware. Requirements for microcontroller applications have been growing, however, and at a very rapid pace. Feature-rich embedded devices, with a powerful set of sophisticated communication, storage and UI interfaces, are now a norm.

More often than not, a new STM32 design is not about adding just a single feature that’s missing in the existing implementation. The designer typically faces the task of adding support for multiple I/O interfaces, all of which have to be covered by appropriate device drivers, software stacks, and application-programming interfaces.

Linux does start looking attractive as an OS choice at this point, given long lists of new software requirements. Typically, it would support all functional features one may require in a modern MCU application out-of-box. For those requirements not immediately supported out-of-box, an engineer would be able to find an array of open-source Linux projects that can be used as a starting point for software development.

Today’s Linux delivers a number of benefits: it’s royalty-free; full source is available for every component; it’s ubiquitous in all computational domains; knowledgeable developers are relatively easy to find; abundant materials are available on pretty much every aspect of the OS implementation from a mere Google search; and tons of tools, libraries, and applications are out there on the Internet ready for immediate download. It’s easy to see how use of Linux could significantly drive down project costs and reduce the time to market.

Having said all that, concerns often arise that Linux may be too “heavyweight” for a Cortex-M3/M4 design, even with powerful MCU devices like the STM32F42X/43X MCU family. Indeed, though time to market and project costs are crucial, it’s still a fact of life that the decision-making process in a MCU project is also driven by the need to maintain a low component count, minimal bill of materials (BOM), and miniature printed-circuit-board (PCB) footprints.

This article explains how a practical Linux platform can be developed using the STM32F42X/3X MCUs. In its most cost-optimized form, the platform is essentially reduced to only two chips—specifically, the STM32F4 MCU itself and an external SDRAM device.

Linux Execution Model

To start, it’s important to explain the execution model of Linux. As is custom with Linux, things can be done in different ways. However, the default execution model, as deployed on Cortex-M3/M4 MCUs, can be described by the following boot sequence:

)n power-up reset, the Cortex-M3/M4 hardware invokes the U-Boot firmware bootloader from integrated h. U-Boot runs from on-chip flash and uses on-chip RAM for stack, volatile data, and buffers. It requires no external memory unless an explicit command that operates on external flash or external RAM is called.

2. U-Boot provides a command interface that can be used for many instances, but the default boot sequence is for U-Boot to run a command that loads a bootable Linux image to external RAM and passes control to the kernel point. A bootable Linux image can reside in an arbitrary non-volatile storage device as long as U-Boot has a device driver for it. That can be parallel flash, SPI flash, SD Card, USB memory stick, etc.—essentially any I/O interface that's supported by a particular Cortex-M3/M4 MCU and U-Boot has an accompanying device driver. As a special case (for the interface that's typically used during software development phase), a bootable image can reside on a TFTP host and be loaded by U-Boot to the target via Ethernet.

Related

Practical Advice on Running uClinux on Cortex-M3/M4

Build Cortex-MCU-Based Wearables, Accessories with Bluetooth Connectivity

Playing Games With the STM32

3. Once called by U-Boot, the Linux kernel proceeds with bootstrap, initializing its memory-management system, device drivers, various I/O stacks, etc., and eventually mounting the root filesystem. Generally speaking, the root filesystem can reside on any storage device supported by an appropriate Linux device driver, such as flash, an SD card, USB stick, NFS share, and others. One special type of root filesystem, which is frequently used in embedded applications, uses the so-called initramfs. In this scenario, the bootable Linux image is self-sufficient. It doesn't rely on availability of a root filesystem on an external device, and comprises two major pieces: the Linux kernel itself and a cpio representation of the root filesystem. Without going into details, cpio is a Unix archive format that can't be mounted as a filesystem directly, but can be expanded by the kernel into a RAM-based initramfs filesystem and mounted as rootfs.

4. After mounting the initramfs root filesystem in RAM, Linux proceeds to run an init utility from the rootfs, which in turn calls whatever startup scripts are required by a particular application. Typically, startup scripts spawn an interactive shell on the Linux console. Generally speaking, though, one can do whatever makes sense in a particular embedded design. Any user-space tools and applications in the root filesystem are run directly from the fast initramfs in RAM, without need to copy them from a slower storage device. This not only helps to reduce boot times, but also boosts the overall system performance.

An interesting variation of the execution model described above occurs when using the STM32F42X/43X devices: A bootable Linux image can be stored in the on-chip flash, which is large enough (2 MB) to store an image worth of a reasonable Linux functionality. As long as the bootable Linux image is stored in the on-chip flash, there's no need to design in an additional storage device.

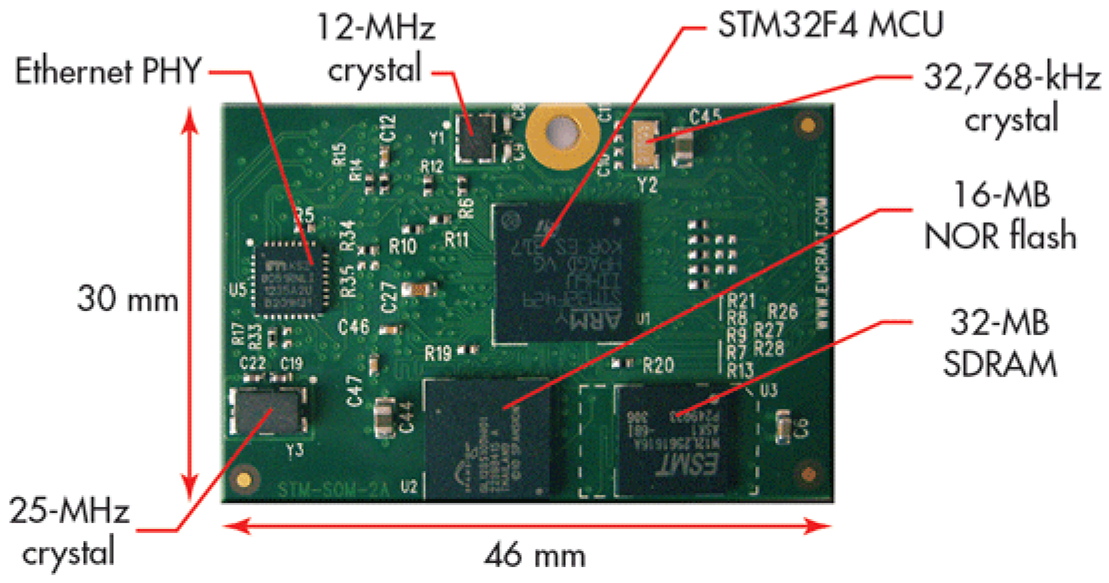
Furthermore, when the bootable Linux image is stored in the on-chip flash, it's possible to run the kernel directly from flash, rather than copy the image to external RAM prior to calling the kernel entry point. This achieves two important goals:

• The Linux bootstrap time is reduced because there's no need to copy the bootable Linux image from a relatively slow storage device to RAM.

• Running the kernel from the integrated zero-wait-state flash substantially improves the overall Linux performance.

can be seen, Linux indeed can run on the STM32F42X/43X from a two-chip design. The following sections vide a demonstration of the available sample Linux features when using this execution model.

Hardware Reference Platform



The sample Linux session provided in the following sections was run on Emcraft Systems' STM32F4 system-on-module (SOM) *(Fig. 1)*. The STM32F4 is a miniature (30 by 46 mm) mezzanine module that includes the following components:

• *STMicro's single-chip STM32F429 MCU:* It combines a 180-MHz, 32-bit ARM Cortex-M4 processor core with an integrated SDRAM interface and set of sophisticated I/O peripherals.

• *12-MHz crystal:* The reference clock is needed in an STM32F42X/43X design regardless of whether Linux or some other RTOS is run on the STM32F4.

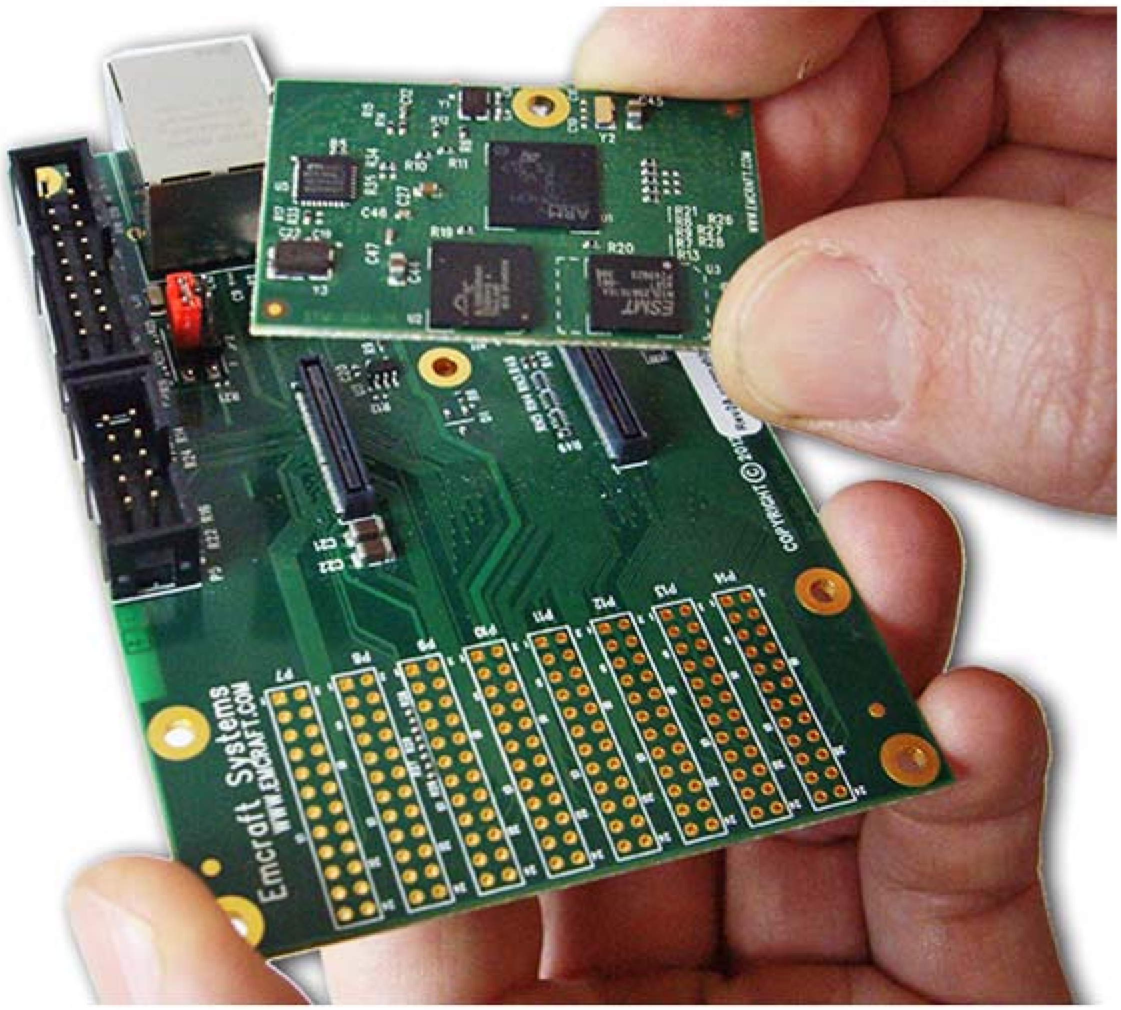
• *32-MB SDRAM:* SDRAM is a must for Linux; however, it could be omitted if an RTOS or bare-bones firmware is run on the STM32F4.

• *16-MB NOR flash:* This is an optional component that adds non-volatile storage.

• *Ethernet PHY and associated 25-MHz crystal:* These components are needed only if a design implements an Ethernet link.

• *32.786-kHz crystal:* This is an optional component used as the reference for the STM32F4 real-time clock (RTC).

In the tests shown below, the SOM was installed into the prototyping and development baseboard included with the STM32F4 SOM starter kit. The baseboard provides immediate access to key STM32F4 I/O interfaces, such as JTAG, serial console using UART, Ethernet, USB OTG, etc., and makes all other STM32F4 signals available on the breadboard area for easy prototyping *(Fig. 2)*.



Sample Linux Session

The following sample session demonstrates the Linux TCP/IP stack running on the STM32F42X/43X. The sophisticated set of features implemented by the Linux TCP/IP stack provides an off-the-shelf foundation for any MCU application that requires feature-rich and robust connectivity. In the example below, TCP/IP runs over Ethernet. However, it can potentially be configured to run over other types of physical links, too. The most obvious examples include Wi-Fi using a USB- or SDIO-based Wi-Fi module, or PPP over UART.

As described, the Linux kernel runs directly from the fast on-chip flash. As a result, the Linux shell is available in about one second from power-on or reset:

```
U-Boot 2010.03 (Oct 27 2014 - 12:55:43)
...
Starting kernel ...

Linux version 2.6.33-arm1 (vlad@ocean.emcraft.com) (gcc version 4.4.1 (Sourcery G++ Lite
2010q1-189) ) #38 Mon Dec 8 11:39:30 MSK 2014
CPU: ARMv7-M Processor [410fc241] revision 1 (ARMv7M)
CPU: NO data cache, NO instruction cache
Machine: STMicro STM32
Built 1 zonelists in Zone order, mobility grouping off. Total pages: 8128
Kernel command line: stm32_platform=stm32f4x9-som console=ttyS0,115200 panic=10
ip=172.17.4.206:172.17.0.1:::stm32f4x9-som:eth0:off ethaddr=C0:B1:3C:88:88:85
PID hash table entries: 128 (order: -3, 512 bytes)
```

```
 entry cache hash table entries: 4096 (order: 2, 16384 bytes)
 node-cache hash table entries: 2048 (order: 1, 8192 bytes)
Memory: 32MB = 32MB total
Memory: 32296k/32296k available, 472k reserved, 0K highmem
Virtual kernel memory layout:
vector : 0x00000000 - 0x00001000 ( 4 kB)
fixmap : 0xfff00000 - 0xfffe0000 ( 896 kB)
vmalloc : 0x00000000 - 0xffffffff (4095 MB)
lowmem : 0xc0000000 - 0xc2000000 ( 32 MB)
modules : 0xc0000000 - 0xc2000000 ( 32 MB)
.init : 0xc0008000 - 0xc000a000 ( 8 kB)
.text : 0xc000a000 - 0xc0016000 ( 48 kB)
.data : 0xc0016000 - 0xc0025480 ( 62 kB)
Hierarchical RCU implementation.
NR_IRQS:90
Calibrating delay loop... 156.87 BogoMIPS (lpj=784384)
Mount-cache hash table entries: 512
NET: Registered protocol family 16
bio: create slab at 0
Switching to clocksource cm3-systick
NET: Registered protocol family 2
IP route cache hash table entries: 1024 (order: 0, 4096 bytes)
TCP established hash table entries: 1024 (order: 1, 8192 bytes)
TCP bind hash table entries: 1024 (order: 0, 4096 bytes)
TCP: Hash tables configured (established 1024 bind 1024)
TCP reno registered
RPC: Registered udp transport module.
RPC: Registered tcp transport module.
RPC: Registered tcp NFSv4.1 backchannel transport module.
Block layer SCSI generic (bsg) driver version 0.4 loaded (major 254)
io scheduler noop registered
io scheduler deadline registered
io scheduler cfq registered (default)
Serial: STM32 USART driver
stm32serial.0: ttyS0 at MMIO 0x40011000 (irq = 37) is a STM32 USART Port
console [ttyS0] enabled
Initializing STM32F4 mapper, copying code from c0026000 to 200001a8, size 560
Using SRAM as buffer with start 20000400 and size 400
stm32f4 platform flash device: 01000000 at 64000000
stm32f4-flash: Found 1 x16 devices at 0x0 in 16-bit bank
Amd/Fujitsu Extended Query Table at 0x0040
number of CFI chips: 1
RedBoot partition parsing not available
Using stm32f4 partition information
Creating 3 MTD partitions on "stm32f4-flash":
0x000000000000-0x000000020000 : "flash_uboot_env"
0x000000020000-0x000000300000 : "flash_linux_image"
0x000000300000-0x000001000000 : "flash_jffs2"
blackfin-eth: Using SRAM for DMA buffers from 20001000
blackfin-eth: found MAC at 0x40028000, irq 61
blackfin_mii_bus: probed
found PHY id 0x221556 addr 0
eth0: using MII interface
eth0: attached PHY driver [Generic PHY] (mii_bus:phy_addr=00:00, irq=-1)
JFFS2 version 2.2. (NAND) б╘ 2001-2006 Red Hat, Inc.
TCP cubic registered
NET: Registered protocol family 17
IP-Config: Guessing netmask 255.255.0.0
IP-Config: Complete:
device=eth0, addr=172.17.4.206, mask=255.255.0.0, gw=255.255.255.255,
host=stm32f4x9-som, domain=, nis-domain=(none),
bootserver=172.17.0.1, rootserver=172.17.0.1, rootpath=
ARMv7-M VFP Extension supported
PHY: 00:00 - Link is Up - 100/Full
```

```
 reeing init memory: 8K
 nit started: BusyBox v1.17.0 (2014-11-18 17:08:28 MSK)
~ #
```

Now let's test the TCP/IP stack on the STM32F4. From the development host, validate that the STM32F4 is visible using ping:

```
-bash-4.2$ ping 172.17.4.206
PING 172.17.4.206 (172.17.4.206) 56(84) bytes of data.
64 bytes from 172.17.4.206: icmp_seq=1 ttl=64 time=1.53 ms
64 bytes from 172.17.4.206: icmp_seq=2 ttl=64 time=0.244 ms
64 bytes from 172.17.4.206: icmp_seq=3 ttl=64 time=0.257 ms
64 bytes from 172.17.4.206: icmp_seq=4 ttl=64 time=0.229 ms
^C
--- 172.17.4.206 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3972ms
rtt min/avg/max/mdev = 0.229/0.567/1.539/0.561 ms
-bash-4.2$
```

To test connectivity in the other direction, let's ping the development host from the STM32F4:

```
~ # ping 172.17.0.1
PING 172.17.0.1 (172.17.0.1): 56 data bytes
64 bytes from 172.17.0.1: seq=0 ttl=64 time=3.302 ms
64 bytes from 172.17.0.1: seq=1 ttl=64 time=8.921 ms
64 bytes from 172.17.0.1: seq=2 ttl=64 time=8.977 ms
64 bytes from 172.17.0.1: seq=3 ttl=64 time=9.438 ms
64 bytes from 172.17.0.1: seq=4 ttl=64 time=0.516 ms
64 bytes from 172.17.0.1: seq=5 ttl=64 time=0.506 ms
^C
--- 172.17.0.1 ping statistics ---
6 packets transmitted, 6 packets received, 0% packet loss
round-trip min/avg/max = 0.506/5.276/9.438 ms
~ #
```

On the target, start the telnetd daemon to allow remote connections to the STM32F4:

```
~ # telnetd
~ # ps
PID USER VSZ STAT COMMAND
1 root 352 S init
2 root 0 SW [kthreadd]
3 root 0 SW [ksoftirqd/0]
4 root 0 SW [events/0]
5 root 0 SW [khelper]
6 root 0 SW [async/mgr]
7 root 0 SW [sync_supers]
8 root 0 SW [bdi-default]
9 root 0 SW [kblockd/0]
10 root 0 SW [rpciod/0]
11 root 0 SW [kswapd0]
12 root 0 SW [mtdblockd]
13 root 0 SW [nfsiod]
19 root 367 S /bin/hush -i
22 root 332 S telnetd
23 root 348 R ps
~ #
```

Now we can connect to the STM32F4 from the development host using telnet. The target is configured to accept empty password for root, so just hit Enter when asked for the password:

```
-bash-4.2$ telnet 172.17.4.206
Trying 172.17.4.206...
Connected to 172.17.4.206.
Escape character is '^]'.

stm32f4x9-som login: root
Password:
~ # ls
bin dev etc httpd init mnt proc root sys usr var
~ # exit
Connection closed by foreign host.
-bash-4.2$
```

Let's configure a default gateway and a name resolver on the STM32F4. Note how the sample configuration below uses the public name server provided by Google. Also note the use of the vi editor to edit files on the target:

```
~ # route add default gw 172.17.0.1
~ # vi /etc/resolv.conf
nameserver 8.8.8.8

~
~ #
```

Let's run ntpd to synchronize the time on the STM32F4 with the time provided by the public server:

```
~ # date
Thu Jan 1 00:07:47 UTC 1970
~ # ntpd -p 0.fedora.pool.ntp.org; sleep 5
~ # date
Mon Dec 8 08:54:10 UTC 2014
~ #
```

Here's how we use wget to download a file from a remote server:

```
~ # wget ftp://ftp.gnu.org/README
Connecting to ftp.gnu.org (208.118.235.20:21)
README 100% |*******************************| 1962 --:--:-- ETA
~ # cat README
This is ftp.gnu.org, the FTP server of the the GNU project.
...
~ #
```

We can mount a directory exported by the development host over NFS. This gives us immediate access to files on the host. For instance, we could build target applications on the host and immediately run them on the target over NFS, without having to reboot or re-flash the target:

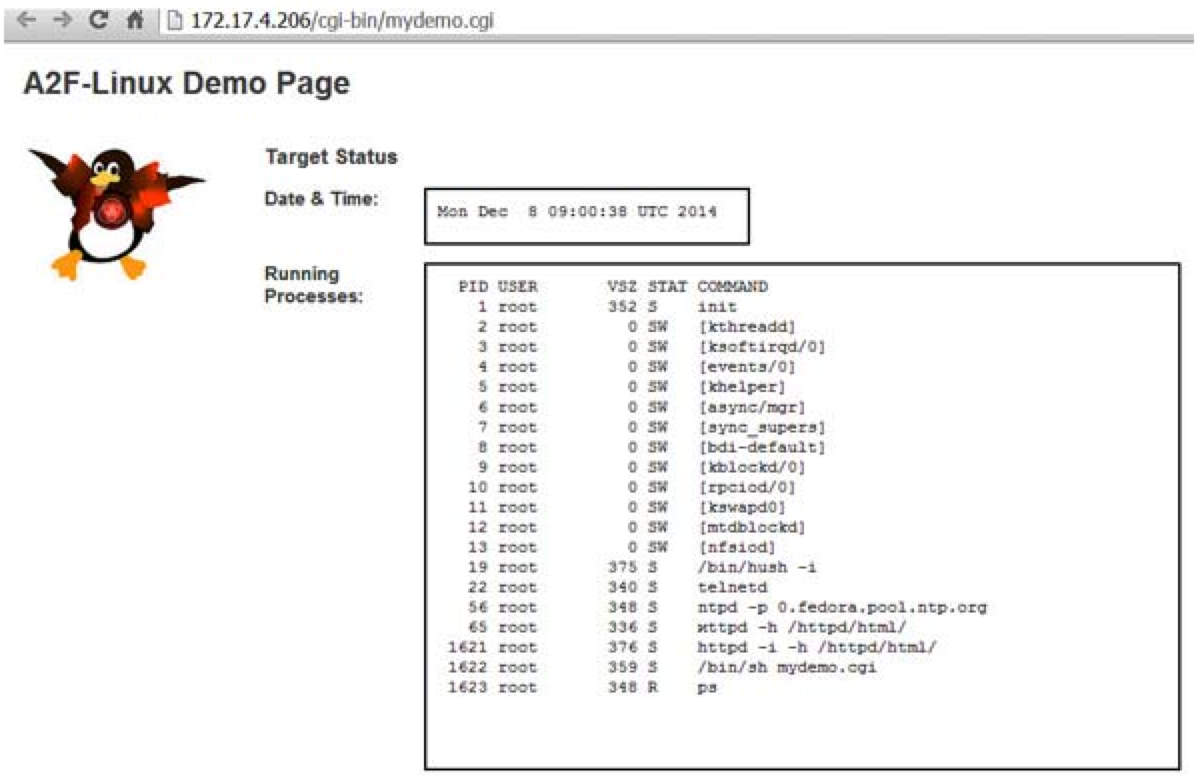
```
~ # mount -o nolock,rsize=1024 172.17.0.1:/home/vlad /mnt
~ # ls -lta /mnt
drwxr-xr-x 12 root root 0 Dec 8 08:54 ..
drwxr-xr-x 17 19270 19270 4096 Dec 8 08:41 .
-rw------- 1 19270 19270 12594 Dec 8 08:41 .viminfo
drwxr-xr-x 18 19270 19270 12288 Dec 8 08:35 .ccache
-rw------- 1 19270 19270 16500 Dec 4 09:21 .bash_history
...
```

```
#
```

Here's how we start the HTTP daemon:

```
~ # httpd -h /httpd/html/
~ #
```

Now that we have the Web server running on the STM32F4, we can open a Web browser on the development host and watch the demo web page provided by the target *(Fig 3)*.



```
Mon Dec  8 09:00:38 UTC 2014
```

```
  PID USER       VSZ STAT COMMAND
    1 root       352 S    init
    2 root         0 SW   [kthreadd]
    3 root         0 SW   [ksoftirqd/0]
    4 root         0 SW   [events/0]
    5 root         0 SW   [khelper]
    6 root         0 SW   [async/mgr]
    7 root         0 SW   [sync_supers]
    8 root         0 SW   [bdi-default]
    9 root         0 SW   [kblockd/0]
   10 root         0 SW   [rpciod/0]
   11 root         0 SW   [kswapd0]
   12 root         0 SW   [mtdblockd]
   13 root         0 SW   [nfsiod]
   19 root       375 S    /bin/hush -i
   22 root       340 S    telnetd
   56 root       348 S    ntpd -p 0.fedora.pool.ntp.org
   65 root       336 S    xttpd -h /httpd/html/
 1621 root       376 S    httpd -i -h /httpd/html/
 1622 root       359 S    /bin/sh mydemo.cgi
 1623 root       348 R    ps
```

What is Linux's Minimal Footprint on STM32F4?

This is perhaps one of the most frequently asked questions about Linux on Cortex-M3/M4. Let's discuss it specifically for the STM32F4.

First of all, external RAM is a must for Linux. Even the smallest Linux configuration requires at least several megabytes of RAM to run from. All Cortex-M3/M4 devices (at least that the author is aware of) limit their internal SRAM to hundreds of kilobytes at best—and STM32F42X/43X is no exception. There's no way Linux can be run from such amounts of RAM. The implication is that, as of this writing, Linux can not run from a single-chip Cortex-M3/M4 device.

As discussed earlier, though, a two-chip design (Cortex-M + external RAM) is a possibility. Two separate "footprint" metrics need to be considered:

• *Size of the bootable Linux image:* As noted, the bootable image consists of two major pieces—the Linux kernel itself and a cpio representation of the root filesystem. The size of a bootable image starts at maybe 512 KB for truly minimal configurations and ranges to whatever, depending on what's in the root filesystem and, to a lesser extent, what configuration options were enabled in your kernel.

e size of a practical bootable image, with Ethernet, TCP/IP, and a reasonable set of user-space tools and plications configured, would be in the 1.5- to 2-MB ballpark. As shown above, an image of that size can fit into the STM32F42X/43X devices' integrated flash.

• *Size of external RAM required for run-time Linux operation:* When our customers ask how much RAM is needed, we reply "the more the better, but no less than 8 MB." It may be possible to run some very basic configurations with rootfs mounted from NFS or some external device even out of 2 MB. Frankly, though, this is more of a toy than a configuration one can build a practical design upon.

As the rule of thumb, consider at least 32-MB RAM if your intention is to use Linux in a serious product. If it becomes clear that you can fit into less RAM as you approach deployment, downsizing to a compatible 16-MB or even 8-MB RAM device becomes a possibility. Still, the safe advice is to start with more RAM rather than less. Requirements for embedded applications grow at an incredibly fast rate, and it's an almost sure bet that in one year's time you will want to add new software features to your product.

The bottom line is that if you're considering Linux for your MCU application, you want to avoid the "if only I had another 512 KB of RAM" kind of situation often encountered by MCU developers. If you want Linux and "features," plan for reasonable amounts of RAM.

On the practical side, and given the specific context of SDRAM memory compatible with the STM32F42X/43X MCUs, the BOM differences between 8-, 16-, and 32-MB SDRAM devices are often quite tolerable. Again, our advice is to play safely and plan for more RAM rather than less.

With that background in mind, let's see how much RAM we have left after running the sample Linux session shown in the previous section. First, here is the list of processes we have running at this point:

```
~ # ps
PID USER VSZ STAT COMMAND
1 root 352 S init
2 root 0 SW [kthreadd]
3 root 0 SW [ksoftirqd/0]
4 root 0 SW [events/0]
5 root 0 SW [khelper]
6 root 0 SW [async/mgr]
7 root 0 SW [sync_supers]
8 root 0 SW [bdi-default]
9 root 0 SW [kblockd/0]
10 root 0 SW [rpciod/0]
11 root 0 SW [kswapd0]
12 root 0 SW [mtdblockd]
13 root 0 SW [nfsiod]
19 root 375 S /bin/hush -i
22 root 340 S telnetd
56 root 348 S ntpd -p 0.fedora.pool.ntp.org
65 root 336 S Xttpd -h /httpd/html/
506 root 348 R ps
~ #
```

And here's the situation with available RAM:

```
~ # cat /proc/meminfo
MemTotal: 32304 kB
MemFree: 27120 kB
Buffers: 0 kB
Cached: 620 kB
SwapCached: 0 kB
```

```
 ctive: 440 kB
 nactive: 180 kB
Active(anon): 0 kB
Inactive(anon): 0 kB
Active(file): 440 kB
Inactive(file): 180 kB
Unevictable: 0 kB
Mlocked: 0 kB
MmapCopy: 3196 kB
SwapTotal: 0 kB
SwapFree: 0 kB
Dirty: 0 kB
Writeback: 0 kB
AnonPages: 0 kB
Mapped: 0 kB
Shmem: 0 kB
Slab: 1196 kB
SReclaimable: 148 kB
SUnreclaim: 1048 kB
KernelStack: 144 kB
PageTables: 0 kB
NFS_Unstable: 0 kB
Bounce: 0 kB
WritebackTmp: 0 kB
CommitLimit: 16152 kB
Committed_AS: 0 kB
VmallocTotal: 0 kB
VmallocUsed: 0 kB
VmallocChunk: 0 kB
~ #
```

As can be seen, we still have a lot of RAM left. However, as you add new features and run processes in parallel, memory will be consumed quickly. Once again to stress the point, we advise you to design in more RAM, rather than less.

Vladimir Khusainov, director of engineering and co-founder of Emcraft, holds a Master's degree in computer science from Moscow State University. He can be reached at vlad@emcraft.com.

Source URL: http://electronicdesign.com/microcontrollers/running-linux-two-chip-stm32f4-design