NinePea — A Small 9P Library for Arduino and Plan 9

Eli Cohen
echoline@gmail.com

ABSTRACT

NinePea is a tiny library for 9P servers specifically designed for use with Arduino Mega boards connected to Unix-like operating systems and Plan 9.

1. Introduction

NinePea [1] was originally a project for Arduino. It implements the 9P [2] filesystem protocol for servers running on the Arduino board. Compared to other 9P libraries it is more compact. NinePea sacrifices many of the features they would have in favor of minimalism. It only supports the 9P2000 version of 9P and does not support auth. It does not do much error checking, and is only meant to be used locally. This is for the advantage of being able to work on systems with only a few kilobytes of RAM. *NinePea* is intended for Arduinos and packaged as an Arduino library, but it is portable C which can easily be compiled on other systems.

2. Synthetic Filesystems

Linux users may be familiar with the `/proc` filesystem which is a synthetic filesystem from the Linux kernel exposing information about running processes and other system information. *NinePea* is used to create synthetic filesystems such as these on separate hardware devices or as programs running in user-space. Examples of the usage of *NinePea* include pinfs which presents the digital GPIO pins of an Arduino as files, randomfs which implements a hardware random number device file using a mountable Arduino, V4LFS [3] for Linux which presents the most recent picture taken by a webcam as a JPEG, a hardware cons device using a television monitor and a PS/2 keyboard connected to an Arduino, and the unfinished etherESP32 [4] implementation which is intended to turn an ESP32 board into a mountable WiFi network device.

3. Arduino Boards

Arduinos are simple open-source electronics prototyping boards. Programs written in the Arduino IDE for use with these boards are called sketches. Arduino started as a board with simple Atmel AVR chips. The Arduino Mega 2560 is based around an Atmega2560 chip, a programmable MCU with only 8 KB RAM, 256 KB of programmable flash, and 4 KB of EEPROM storage. It has 16 analog-to-digital inputs, 70 pins that can be used for digital I/O, and a lot of other functionality for prototyping. These boards can drive pulse-width modulation signals, be used as controllers for SPI or I2C, and have 4 TTL serial ports. The Arduino project also encompasses a simple IDE in Java, which compiles C++ sketches to software for the AVR chip. This of course doesn't run on Plan 9, at least at the time of this writing. Plan 9 doesn't support Java or C++, and there is

little else available on Plan 9 for programming Atmel chips. All of these examples were compiled, uploaded, and used on Linux.

4. Pinfs Example

Pinfs is a simple example Arduino sketch. It serves a single directory that shows each of the digital pins as a file. Here is a usage case on Linux using the 9mount convenience program and netcat to listen on TCP/IP:

```
$ mkfifo pipe
$ ./tty9p /dev/ttyUSB0 < pipe | nc -l -p 2000 >> pipe &
$ 9mount tcp!localhost!2000 /mnt/arduino
$ ls /mnt/arduino
d00 d05 d10 d15 d20 d25 d30 d35 d40 d45 d50 d55 d60 d65
d01 d06 d11 d16 d21 d26 d31 d36 d41 d46 d51 d56 d61 d66
d02 d07 d12 d17 d22 d27 d32 d37 d42 d47 d52 d57 d62 d67
d03 d08 d13 d18 d23 d28 d33 d38 d43 d48 d53 d58 d63 d68
d04 d09 d14 d19 d24 d29 d34 d39 d44 d49 d54 d59 d64 d69
```

This is the pinfs example running on the Arduino. First a fifo pipe is made with the standard mkfifo command. The tty9p utility is used to set serial port parameters and packetize the 9P messages on the serial port, and it is looped through the fifo pipe and netcat to listen on TCP port 2000. 9mount is a convenience program for making 9P mounting simpler on Linux, and here it is being used to mount the serial port through the above command. Then the filesystem can just be accessed with the standard Linux tools. Shown here is a listing of all the synthetic files being generated by the Arduino board. This example is running on Linux, but a Plan 9 computer can easily mount the service on the Linux computer over TCP. The user can write a 1 or 0 to d13 to turn on or off the LED on digital pin 13 of the Arduino board, and can read or write any of the 70 files to access the digital I/O state of any pin.

On Plan 9, the serial port can be mounted directly. 9P doesn't care about the underlying transport, so on Plan 9 this works with any file giving a 9P server:

```
% mount /shr/usb/eiaU50b4c /n/arduino
```

5. Randomfs Example

Another example of *NinePea* is the randomfs sketch for Arduino. It presents one file, `random`, which can be mounted over `/dev` to replace Plan 9's `/dev/random`. This sketch is a bare example of a 9P fileserver, it only presents one file. On read it reads each of the 16 analog inputs on the Arduino Mega and XORs them with the system milliseconds counter:

```
for (pin = A0; pin < (NUM_ANALOG_INPUTS + A0); pin++) {
    seed <<= 1;
    seed ^= analogRead(pin) ^ millis();
}
```

This is to seed a Chacha20 pseudo-random number generator. The Chacha20 cipher code [5] in this sketch is borrowed from 9front. The file it presents can be bound before `/dev/random` to replace it with the random number generation from the Arduino. `AnalogRead(pin)` returns the ambient voltage on that pin which is XOR'ed with `millis()` which is used as the seed for a PRNG. `Millis()` returns the number of milliseconds the Arduino has been booted, and the ambient voltage is similar on each pin, so the randomness of the seed might not be great, but then the sketch uses a Chacha20 stream cipher from that seed. The manual page for `/dev/random`

specifically says it is only to be used on Plan 9 as a seed for better pseudo-random number generators in the programs themselves. This sketch has not been thoroughly assessed for how well it generates randomness or how cryptographically secure it is. Additional hardware could be attached to the analog inputs to better seed the hardware random number generator device. For example, decaying particles from a block of radioactive metal could be used to input random voltages to the pins for seeding the hardware random number generator.

6. Design Choices and Further Examples

The *NinePea* library was designed to be compact. It leaves out a lot of functionality that other 9P libraries have. It doesn't have anything in it specifically for any transport layer. It only operates on buffers which are filled and read from by the implementation. It doesn't multiplex connections; this has to be handled by the implementation, if at all. *NinePea* doesn't do anything special for delayed RReads or TFlushes. This can be handled separately by the implementation. TReads can be buffered and replied to only when they are ready. By default, everything is handled in-order by the proc9p function. An incoming TRead is processed immediately and an RRead is sent. One project using this library was a hardware cons device using an Arduino Mega 2560, a keyboard, and a television monitor. In that project, RReads sometimes needed to be delayed, and TFlushes needed to be handled. The way this was achieved was by buffering TReads and handling them separately from other message types elsewhere in a loop.

Another unfinished project used an ESP32 board as a 9P-speaking wifi dongle. The goal of this project was to make a wifi dongle for Plan 9 that only required a USB-to-serial driver, and then could simply be mounted as a 9P device. In that project delayed RReads were handled by every incoming 9P message spawning a separate thread. When threads were complete, they locked a shared lock and sent their response.

The fileserver introduced above is an example of *NinePea* being used on Linux, instead of an MCU. It layers *NinePea* over Linux's video4linux subsystem for webcams. It presents only one synthetic file, `jpeg`, which is a synthetic jpeg generated by taking a picture from the webcam. The contents of this jpeg change over time, always showing the latest image from the webcam. Most people who have used a modern computer are familiar with what a JPEG file is. It's a picture which is generally in disk storage. The user can open it, delete it, etc. A synthetic file's contents are generated, rather than stored on disk. Imagine opening a JPEG and it's the most recent image from a webcam. An implementation of exactly this is V4LFS for Linux which uses *NinePea* directly. *NinePea* was designed to be portable C which can be included on many different types of systems to create fileservers such as this.

7. Comparisons to Other 9P Libraries

NinePea is intended to be very portable and compact. Running `wc -l` to count the lines of code in all the .c files in /sys/src/lib9p shows that Plan 9's 9P library is only 2781 lines of code. Libixp for Unix-like systems shows 4045 lines of code in all the C files of the main library. *NinePea* however, is only 496 lines of code in NinePea.cpp. Arduino labels the file as C++, but it is completely portable C which can be used on many other systems. Plan 9's Lib9p does a lot of things that NinePea does not. NinePea does not handle authentication, it doesn't have built-in queueing, and it doesn't have any functionality to post a service or to handle threading or any type of listening. *NinePea* only operates on one buffer in memory, which the programmer must handle separately on their own. Outgoing response messages overwrite incoming messages in the same buffer, which is

only 4 KB by default.

8. NinePea Library Usage

NinePea is a minimal 9P server library which was originally intended for use with Arduino Megas. Some code in the header file for the structs, message types, and bit-field flags was borrowed from Plan9port [6]. Everything for a 9P server runs on the Atmega chip on the Arduino board. To use the library, first callback functions for each of the 9P message types [7] must be written and pointed to in a callbacks struct. Fids are handled manually with helper fid hash table functions.

```
Serial.begin(115200);

fs_fid_init(64);

callbacks.attach = fs_attach;
callbacks.flush = fs_flush;
callbacks.walk = fs_walk;
callbacks.open = fs_open;
callbacks.create = fs_create;
callbacks.read = fs_read;
callbacks.write = fs_write;
callbacks.clunk = fs_clunk;
callbacks.remove = fs_remove;
callbacks.stat = fs_stat;
callbacks.wstat = fs_wstat;
```

After that, a 9P message is read from the serial device and buffered into RAM. The buffer and callbacks structure are then passed to the proc9p function which processes the message. Proc9p calls the callbacks with at least an Fcall struct as a parameter, and also buffers for reads and writes. The 9P message buffer is overwritten by proc9p with 9P data to send back to the client, and proc9p returns the total length of the resulting 9P response. An Arduino sketch can then send that buffer back over the serial port.

```
r = 0;
while (r < 5) {
  while (Serial.available() < 1);
  msg[r++] = Serial.read();
}

i = 0;
get4(msg, i, msglen);

if (msg[i] & 1 || msglen > MAX_MSG || msg[i] < TVersion || msg[i] > TWStat) {
  // error
}

while (r < msglen) {
  while (Serial.available() < 1);
  msg[r++] = Serial.read();
}

msglen = proc9p(msg, msglen, &callbacks);

Serial.write(msg, msglen);
```

Handling fids is a bit tricky. Some helper functions make this easier:

```
struct hentry {
    unsigned long id;
    unsigned long data;
    struct hentry *next;
    struct hentry *prev;
    void *aux;
};

struct htable {
    unsigned char length;
    struct hentry **data;
};

struct hentry* fs_fid_find(unsigned long id);
struct hentry* fs_fid_add(unsigned long id, unsigned long data);
void fs_fid_del(unsigned long id);
void fs_fid_init(int l);
```

9P typically has a maximum of 8 KB per message, and these chips have only 8 KB of RAM. Linux's 9P support has a minimum of 4 KB message size, which barely fits here. The iounit is set to 4 KB by default. On a Linux computer the included tty9p program ensures that an entire 9P message is sent or received one at a time, but other than that everything runs on the Arduino. The serial port itself becomes a 9P fileserver endpoint.

9. Previous Work

Inferno's Styx protocol, which is compatible with 9P, was previously used [8] on Lego Mindstorms RCX bricks. That work was specific to the Lego RCX; it was never meant as a library. Styx-on-a-Brick was only used for one server for the Lego brick that exposed the motors and sensors. NinePea is a more general purpose library in portable C. It is a very small implementation of a 9P server intended for systems without many resources. It does borrow some structs and other header data from Plan9port, but it dispenses with a lot of functionality that would be available in other 9P server libraries.

10. Other Uses

NinePea was originally meant for use on Arduino boards. It could be made to work with a wifi shield to present a slow ethernet device for Plan 9 without writing any drivers. One could add a speaker and have a simple audio device. Arduino is meant for electronics prototyping, and although *NinePea* only builds under the Arduino IDE, once the board is configured and programmed as desired it can be plugged into a Plan 9 system and mounted like any other 9P server. It can be used to gather information from I2C or SPI sensors, to construct or read a signal, or for many other electronics prototyping applications. *NinePea* is also flexible; the Arduino library is labelled a C++ file but it's really just portable C. It's just a header and a C file that can be included with a project for simple 9P support. It does have some drawbacks. It isn't meant to be public-facing. It doesn't do authentication and it barely does any error checking. The V4LFS program shows how it can be included and used on Linux to wrap a webcam as a synthetic JPEG.

11. Performance of 9P

This is an example of the pinfs sketch. One interesting use of *NinePea* was using Plan 9 methodologies to bind the networking stack of the Linux machine the Arduino was plugged into over /net of a computer across the country and mounting the Arduino remotely:

```
linux$ ./tty9p /dev/ttyUSB0 < pipe | nc -l -p 2000 >> pipe

cpu% bind /mnt/term/net /net
cpu% srv tcp!localhost!2000 arduino
cpu% mount /srv/arduino /n/a
```

This command sequence serves the serial port on TCP port 2000 from Linux, switches over to using the Linux machine's networking stack on the remote Plan 9 computer, posts a 9P service for connecting to port 2000, and finally mounts the service. After doing so, writing a 0 or 1 across the country and back takes almost a full second of 9P traffic back and forth:

```
cpu% echo 1 > /n/a/d13
```

9P adds a lot of overhead of messages going back and forth, besides the data inside it. In this case only one byte was being sent, but the overhead of 9P and the Internet across the country and back caused the data to take quite a long time to be sent out across the Internet, return, and finally go out and back over the 115200 baud serial port. The serial port was not the main bottleneck in this case. 9P is still very slow over long distances because of all the overhead going back and forth for each operation. Each operation of writing a 1 to the d13 file involved several bytes of 9P out and back for walks, opens, writes, and closes. Mounting *NinePea* locally on the Linux computer, the main bottleneck as expected was the serial port itself, sending 9P back and forth as quickly as it could. The Arduino has an LED on digital pin 13 and LEDs for serial recieve and transmit. When it was mounted locally the LEDs for the serial port stayed on continuously while blinking the pin 13 LED in a loop, whereas when it was mounted remotely there was a visible delay as each 9P message was received.

12. References

[1] `https://github.com/echoline/NinePea` The source for this project
[2] `https://9p.cat-v.org` A site about 9P
[3] `https://github.com/echoline/V4LFS` NinePea-based V4LFS Linux webcam fileserver
[4] `https://github.com/echoline/etherESP32` ESP32 WiFi dongle
[5] `http://git.9front.org/plan9front/plan9front/HEAD/sys/src/libsec/port/chachablock.c/raw` Chacha20 stream cipher code
[6] `https://9fans.github.io/plan9port/` Plan9port website
[7] `intro(5)` Introduction to manual section 5 of Plan 9
[8] `http://doc.cat-v.org/inferno/4th_edition/styx-on-a-brick/` Inferno "Styx-on-a-Brick" paper