Realtime Embedded Coding: Threads

Bernd Porr

Introduction

1. Load balancing (using all CPU cores)
2. Endless loops with blocking I/O or GPIO wakeups to establish **precise timing** for callbacks.
3. **Asynchronous execution** of time-consuming tasks with a callback after the task has completed.

Thread and worker

A thread is just a *wrapper* for the actual method which is running independently.
The method being run in the thread is often called a *worker*.

Creating threads

In C++ a worker is a method within a class:

```
uthread = new std::thread(&MyClassWithAThread::run, this);
```

where `MyClassWithAThread` is a class containing the method "run":

```
class MyClassWithAThread {
        void run() {
                // ... hard work is done here
                doCallback(result); // hand the result over
        }
}
```

The "&" signifies a functor. You can also use a lambda function:

```
uthread = std::thread([this](){run();});
```

Lifetime of a thread

Threads terminate simply once the worker has finished its job. To wait for the termination of the thread use the “join()” method:

```
void stop() {
        uthread->join();
        delete uthread;
}
```

Timed callbacks with blocking I/O and endless loops

Threads with endless loops are often used in conjunction with blocking I/O which provide the timing:

```
void run() {
        running = true;
        while (running) {
                read(buffer); // blocking
                doCallback(buffer); // hand data to client
        }
}
```

Timing without blocking I/O: select()

Non-blocking I/O can be turned into blocking I/O with select() which waits till data has become available.

```
FD_ZERO(&rdset);
FD_SET(fileno,&rdset);
struct timeval timeout;
timeout.tv_sec = 0;
timeout.tv_usec = 500000;
int ret = select(fileno+1,&rdset,NULL,NULL,&timeout);
if (ret < 0) return ret;
// non-blocking I/O read
ret = read(fileno,buffer,bytes_per_sample * n_chan);
```

Timing without blocking I/O: `poll()`

`poll()` is similar to `select()` but has less temporal resolution (ms) to wake up a thread. Here used to wake up a thread after a GPIO pin has been triggered in `/sys`:

```
int SysGPIO::gpio_poll(int gpio_fd, int timeout) const
{
  struct pollfd fdset[1] = {};
  int nfds = 1;
  char buf[MAX_BUF] = { 0 };
  fdset[0].fd = gpio_fd;
  fdset[0].events = POLLPRI;

  int rc = poll(fdset, nfds, timeout);

  if (fdset[0].revents & POLLPRI) {
    (void)read(fdset[0].fd, buf, MAX_BUF);
  }
  return rc;
}
```

Running/stopping workers with endless loops

The flag `running` which is controlled by the main program and is set to zero to terminate the thread:

```
void stop() {
        running = false; // <----- HERE!!
        uthread->join();
        delete uthread;
}
```

Note that `join()` is a **blocking** operation and needs to be used with care not to lock up the main program. You probably only need it when your program is terminating. See `https://github.com/berndporr/rpi_AD7705_daq` for an example.

Timing within threads: Timing with blocking I/O

In this example the blocking `read` command creates the timing of the callback:

```
void run() {
       running = 1;
       while (running) {
              read(buffer); // <--- waits for data
              doCallback(buffer); // hand data to client
       }
}
```

Timing with Linux/pigpio timers

As a *last resort* one can use a timer. Similar to threads one can create timers which are then called at certain intervals. As with threads timers should be *hidden* within a C++ class as *private* members which then trigger *public callbacks* via C++ callback mechanisms as described above.
Generally it's *not recommended* to use timers for anything which needs to be reliably sampled, for example ADC converters or sensors with sampling rates higher than a few Hz. On the raspberry PI use the pigpio library and its timer callbacks — if needed at all.

Summary

- Threads do load balancing.
- Threads create timing by using blocking I/O.
- Threads prevent programs from locking up.

. . . and remember: threads running in classes and communicate via callbacks!