EMBEDDED LINUX APPLIKATIONEN MIT C++ ENTWICKELN

Günter Obiltschnig
Applied Informatics Software Engineering GmbH
St. Peter 33
9184 St. Jakob im Rosental
Austria
guenter.obiltschnig@appinf.com

Ein Weg, die steigende Komplexität in der Embedded-Software zu bändigen, ist der Einsatz einer objektorientierten Programmiersprache und entsprechender Klassenbibliotheken. Speziell für Linux-basierte Embedded-Systeme bietet sich aufgrund der Verfügbarkeit eines hervorragenden Compilers (GCC) die Sprache C++ an. Dieser Artikel zeigt wie mit Hilfe der frei verfügbaren und quelloffenen POCO C++ Libraries eine Applikation entwickelt wird, die Messwerte über ein Web-Interface darstellt.

Einleitung

In der Entwicklung von Embedded-Systemen sieht man sich zunehmend mit dem Problem rapide steigender Anforderungen, und somit steigender Komplexität der Software konfrontiert. Ein Weg diese Komplexität zu bändigen ist der Einsatz einer objektorientierten Programmiersprache und entsprechender Klassenbibliotheken. C++ erfreut sich hier zunehmender Beliebtheit. Speziell Linux-basierte Embedded-Systeme bieten sich aufgrund der Verfügbarkeit eines hervorragenden C++ Compilers (GCC) für die C++ Entwicklung geradezu an. Durch den Einsatz von C++ Bibliotheken kann der Entwicklungsaufwand für eine Applikation noch weiter reduziert werden, in dem auf vorgefertigte Code-Bausteine der Bibliothek zurückgegriffen wird. Beispielsweise bieten die hier verwendeten POCO C++ Libraries [1] eine Vielzahl von Bausteinen an, angefangen von Klassen zum vereinfachten Umgang mit Threads und dem Dateisystem, zum Auswerten von Konfigurationsdateien, bis hin zu einem Webserver, der direkt in die Applikation integriert werden kann.

Nachfolgend wird beschrieben, wie mit C++ und den POCO C++ Libraries eine Applikation für ein Linux-basiertes Embedded-System erstellt werden kann. Die Applikation beinhaltet einen Webserver welcher ein dynamisches HTML Dokument bereitstellt, über das ein Temperatur-Messwert dargestellt wird. Der Messwert wird über einen Analog-Digital Wandler, welcher über einen Treiber (Linux Kernel-Modul) angesprochen wird, gelesen.

Hardware-Zugriff und I/O

Embedded Linux unterscheidet sich von vielen anderen Embedded- und Echtzeit-Betriebssystemen dadurch, dass es einer Applikation nicht möglich ist, direkt auf die Hardware zuzugreifen. Der Zugriff auf die Hardware erfolgt immer über einen Treiber, welcher als Kernel-Modul direkt im Linux-Kernel läuft. Die Kommunikation zwischen Applikation und Treiber erfolgt normalerweise über die I/O Funktionen, die der Kernel der Applikation bereitstellt – *read()*, *write()* und *ioctl()*.

Für das vorliegende Beispiel sei angenommen, dass ein Analog-Digital Wandler (ADC) über ein SPI (Serial Peripheral Interface) an einen ARM9 Microcontroller angeschlossen ist. Der ADC wird über einen Treiber angesprochen, der, wie für Linux üblich, als Kernel-Modul implementiert ist. Die eigentliche Implementierung des Kernel-Moduls ist für das Beispiel nicht von Interesse, sehr wohl aber die Schnittstelle, welche der Treiber bereitstellt. Angenommen, für den Treiber existiert im */dev* Verzeichnis ein Eintrag, z. B. */dev/spiadc0*, so kann mit dem Code-Fragment in Beispiel 1 ein Wert vom ADC gelesen werden:

```
unsigned short value;
```

```
int fd = open("/dev/spiadc0", O_RDWR);
if (fd != -1)
{
    int rc = ioctl(fd, SPIADC_IOCRD, &value);
    close(fd);
    if (rc < 0)
        handleError();
}
else handleError();
```

Beispiel 1: Lesen eines Wertes vom Analog-Digital Wandler

Es ist allerdings nicht sehr elegant, jedes Mal wenn ein Wert vom Temperatursensor gelesen werden soll, den Code aus Beispiel 1 zu verwenden. Im Sinne der objekt-orientierten Programmierung soll der Zugriff auf den ADC in einer Klasse gekapselt werden. Beispiel 2 zeigt die Deklaration, und Beispiel 3 eine mögliche Implementierung einer solchen Klasse.

```
class ADC
{
public:
    ADC(const std::string& device);
    ~ADC();

    Poco::UInt16 read() const;

private:
    ADC();
    ADC(const ADC&);
    ADC& operator = (const ADC&);

    int _fd;
};
```

Beispiel 2: Klasse für den Zugriff auf den Analog-Digital Wandler

Beim Erstellen eines ADC Objektes wird über den Konstruktor der Dateiname des Gerätes (also z. B. */dev/spiadc0*) angegeben. Über *read()* wird der aktuelle Wert vom ADC ausgelesen. Der Typ *Poco::UInt16* entspricht einem vorzeichenlosen 16-bit Integer. Alternativ könnte *uint16_t* (aus *<stdint.h>)* verwendet werden. Wie die Implementierung der ADC Klasse aussehen könnte zeigt Beispiel 3.

```
ADC::ADC(const std::string& device)
{
    _fd = ::open(device.c_str(), O_RDWR);
    if (_fd == -1) throw Poco::OpenFileException(device);
}

ADC::~ADC()
{
    ::close(_fd);
}

Poco::UInt16 ADC::read() const
{
    Poco::UInt16 val;
    if (::ioctl(_fd, SPIADC_IOCRD, &val) < 0)
        throw Poco::IOException(Poco::format(
            "Cannot read ADC (ioctl failed with errno=%d)", errno));
    return val;
}
```

Beispiel 3: Implementierung der ADC Klasse

Natürlich kann das Design dieser Klasse noch etwas verfeinern werden, etwa in dem die Elementfunktion *read()* als *pure virtual* deklariert wird, und die eigentliche Implementierung der

Funktionalität in einer von *ADC* abgeleiteten Klasse *SPIADC* erfolgt. Dies würde die Möglichkeit bieten, in der Applikation zwischen verschiedenen ADC Implementierungen zu wechseln. Besonders hilfreich ist dies beim Testen. Mit einer Dummy-Implementierung des ADC kann die Applikation bereits getestet werden, bevor die Zielhardware zur Verfügung steht.

Das Auslesen des ADC ist aber nur der erste Schritt zur Ermittlung des Sensorwertes. Der 16-bit Integer Wert vom ADC muss noch in eine Temperatur (z. B. in Grad Celsius) umgewandelt werden. Dafür wird eine Klasse mit dem Namen *TemperatureSensor* erstellt. Die Deklaration dieser Klasse ist in Beispiel 4 gezeigt.

```
class TemperatureSensor
{
public:
    TemperatureSensor(ADC& adc);
    ~TemperatureSensor();

    void calibrate(float low, float high);
    float read() const;

private:
    TemperatureSensor();
    TemperatureSensor(const TemperatureSensor&);
    TemperatureSensor& operator = (const TemperatureSensor&);

    ADC& _adc;
    float _low;
    float _high;
};
```

Beispiel 4: Deklaration der Klasse TemperatureSensor

Dem Konstruktor von *TemperatureSensor* wird ein ADC Objekt mitgegeben, über das der aktuelle Messwert ermittelt werden kann. Mit *read()* wird die aktuelle Temperatur gelesen. Bevor allerdings eine gültige Temperatur gelesen werden kann, muss der Sensor kalibriert werden. Dies erfolgt über einen Aufruf der Elementfunktion *calibrate()*, welcher ein minimaler und ein maximaler Temperaturwert mitgegeben wird (der Einfachheit halber sei angenommen, dass der Sensor linear arbeitet). Die Implementierung der Klasse *TemperatureSensor* findet sich in Beispiel 5.

```
TemperatureSensor::TemperatureSensor(ADC& adc):
    _adc(adc),
    _low(0.0f),
    _high(0.0f)
{
}

TemperatureSensor::~TemperatureSensor()
{
}

void TemperatureSensor::calibrate(float low, float high)
{
    poco_assert (low < high);

    _low  = low;
    _high = high;
}

float TemperatureSensor::read() const
{
    poco_assert (_low < _high);

    return _low + (_high - _low)*_adc.read()/0xFFFF;
}
```

Beispiel 5: Implementierung der Klasse TemperatureSensor

Webserver und Applikation

Nachdem der Zugriff auf den Temperatursensor implementiert ist, folgt als nächstes die Implementierung des Webservers. Dazu wird der Webserver aus den POCO C++ Libraries verwendet. Dieser Webserver zeichnet sich dadurch aus, dass er einfach in eine bestehende Applikation eingebettet werden kann, und als eigener Thread innerhalb einer Applikation läuft. Gegenüber einem als eigenen Prozess laufenden Webserver wie z. B. Boa, thttpd oder Apache hat dies zwei große Vorteile. Erstens ist es zur Darstellung von dynamischen Webseiten nicht notwendig ein separates CGI Programm zu schreiben. Das Starten eines CGI Programms durch den Webserver belastet immer das System und benötigt etwas Zeit. Bei vielen Zugriffen trägt das, besonders bei langsameren Prozessoren, erheblich zur Systemlast bei. Zweitens hat man aus dem Webserver heraus direkten Zugriff auf den gesamten Zustand der Applikation. Bei einem CGI Programm ist immer zu überlegen, wie das CGI Programm auf die internen Daten der Applikation zugreifen kann (z. B. über Dateien, Shared Memory oder andere Mechanismen der Interprozesskommunikation).

Dem Webserver der POCO C++ Libraries muss mitgeteilt werden, wie auszuliefernde Webseiten zu erstellen sind. Dies funktioniert durch Verwendung des Abstract Factory Design Patterns [2]. Für jedes Dokument, welches vom Server ausgeliefert wird, ist eine Unterklasse von *Poco::Net::HTTPRequestHandler* zu erstellen. Dabei ist die Elementfunktion *handleRequest()* so zu überschreiben, dass das entsprechende HTML Dokument erstellt wird. Beispiel 6 zeigt die Implementierung dieser Klasse zur Darstellung der aktuellen Temperatur.

```
class TempRequestHandler: public Poco::Net::HTTPRequestHandler
{
public:
    TempRequestHandler(const TemperatureSensor& sensor):
        _sensor(sensor)
    {
    }

    void handleRequest(Poco::Net::HTTPServerRequest& request,
                       Poco::Net::HTTPServerResponse& response)
    {
        response.setChunkedTransferEncoding(true);
        response.setContentType("text/html");

        std::ostream& ostr = response.send();
        ostr << "<html>"
             << "<head><title>Temperature Sensor</title></head>"
             << "<body>"
             << "<p style=\"text-align: center; font-size: 48px;\">"
             << _sensor.read()
             << "</p>"
             << "</body>"
             << "</html>";
      }

private:
      const TemperatureSensor& _sensor;
};
```

Beispiel 6: Implementierung eines HTTP Request Handlers

Zusätzlich muss die dazugehörende Factory-Klasse erstellt werden. Diese muss von *Poco::Net::HTTPRequestHandlerFactory* abgeleitet sein. Beispiel 7 zeigt die Implementierung dieser Klasse.

```
class TempRequestHandlerFactory: public HTTPRequestHandlerFactory
{
public:
```

```
    TempRequestHandlerFactory(const TemperatureSensor& sensor):
        _sensor(sensor)
    {
    }

    HTTPRequestHandler* createRequestHandler(
        const HTTPServerRequest& request)
    {
        return new TempRequestHandler(_sensor);
    }

private:
    const TemperatureSensor& _sensor;
};
```

Beispiel 7: Implementierung einer HTTP Request Handler Factory

Schlussendlich bleibt noch die eigentliche Applikation zu schreiben. Hier kann auf weitere Features der POCO C++ Libraries zurückgegriffen werden, z. B. das automatische Auslesen von Konfigurationsdateien bei Verwendung der Klasse *Poco::Util::ServerApplication*. Beim Starten der Applikation wird zunächst eine Konfigurationsdatei ausgelesen, in welcher Parameter wie der TCP Port auf dem der Webserver laufen soll, der Name der Gerätedatei für den ADC, sowie die Kalibrierungswerte für den Temperatursensor gespeichert sind. Beispiel 9 zeigt einen möglichen Inhalt dieser Datei. Danach werden die Objekte für ADC und Temperatursensor initialisiert. Schlussendlich wird der Webserver gestartet. Dem Webserver wird dabei eine Instanz der *TempRequestHandlerFactory* mitgegeben. Sobald nun eine Anfrage den Webserver erreicht wird die Kontrolle vom Webserver an die Elementfunktion *createRequestHandler()* im *TempRequestHandler* übergeben. Diese erzeugt ein *TempRequestHandler* Objekt. Dessen Elementfunktion *handleRequest()* wird anschließend vom Webserver aufgerufen. In *handleRequest()* wird der Wert des Temperatursensors gelesen, sowie eine einfache HTML Seite erstellt, um die Temperator in einem Webbrowser anzeigen zu können. Beispiel 8 zeigt den Code zum Initialisieren der Applikation.

```
class TempServer: public Poco::Util::ServerApplication
{
    void initialize(Application& self)
    {
        loadConfiguration();
        ServerApplication::initialize(self);
    }

    void uninitialize()
    {
        ServerApplication::uninitialize();
    }

    int main(const std::vector<std::string>& args)
    {
        std::string adcDevice = config().getString("adc.device");
        unsigned short port = static_cast<unsigned short>(
            config().getInt("http.port", 80));
        float lowTemp = static_cast<float>(
            config().getDouble("temp.low"));
        float highTemp = static_cast<float>(
            config().getDouble("temp.high"));

        ADC adc(adcDevice);
        TemperatureSensor tempSensor(adc);
        tempSensor.calibrate(lowTemp, highTemp);

        ServerSocket svs(port);
        HTTPServer srv(
            new TempRequestHandlerFactory(tempSensor),
```

```
            svs,
            new HTTPServerParams);
        srv.start();
        waitForTerminationRequest();
        srv.stop();

        return Application::EXIT_OK;
    }
};

int main(int argc, char** argv)
{
    TempServer app;
    return app.run(argc, argv);
}
```

Beispiel 8: Implementierung der Applikation

```
http.port = 8080
adc.device = /dev/spiadc0
temp.low = -50
temp.high = 130
```

Beispiel 9: Konfigurationsdatei

Ressourcenbedarf

Das Executable der fertigen Applikation hat, für ARM compiliert und statisch gelinkt, eine Größe von 1,6 MByte. Der Speicherbedarf zur Laufzeit beträgt etwa 2 MB. Speziell die Größe des Executables mag auf den ersten Blick etwas groß erscheinen, jedoch darf nicht vergessen werden, dass bereits die C++ Standardbibliothek mehrere hundert KByte davon ausmacht. Dynamisch gelinkt wäre das Executable der Applikation nur ca. 20 bis 30 KByte groß. Typische Linux-basierte Embedded-Systeme haben mittlerweile eine RAM Größe von 32 MByte oder mehr und mindestens 16 MByte Flash, so dass dies nicht ins Gewicht fällt.

Zusammenfassung

Es wurde gezeigt, wie mit C++ und den POCO C++ Libraries eine Applikation für ein Embedded Linux System gebaut wurde, welches einen Temperatursensor ausliest und den Wert über einen Webserver bereitstellt. Auf sauberes objekt-orientiertes Design der Applikation wurde besonderer Wert gelegt, insbesondere beim der Schnittstelle zur Hardware (Analog-Digital Wandler und Temperatursensor).

Der komplette Quellcode der Applikation kann beim Autor angefordert werden.

Quellen

[1] POCO C++ Libraries
http://pocoproject.org

[2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns – Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.

Zum Autor

Günter Obiltschnig ist Geschäftsführer der Applied Informatics Software Engineering GmbH und Gründer des POCO C++ Libraries Open Source Projektes. Er verfügt über 15 Jahre Erfahrung in der Entwicklung von Software für verschiedene Systeme, von verteilten Unternehmensapplikationen bis hin zu Microcontroller-basierten Geräten. Seit mehreren Jahren beschäftigt er sich intensiv mit dem Einsatz von C++ für Embedded-Systeme.