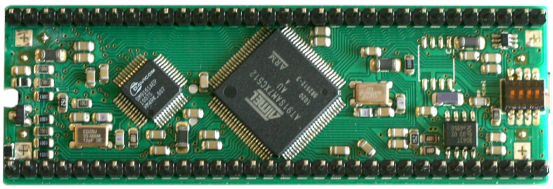
Nut/OS Webserver / Relais Demo mit eNet-sam7X



Simples Web-I/O Interface mit zwei Relais, steuerbar über einen Webbrowser

Dokument Rev. 1.0
30.05.2011

Inhaltsverzeichnis

Web basiertes Messen, Steuern, Regeln mit dem eNet-sam7X

Mikrocontroller Modul mit OpenSource Multitasking Betriebssystem

Wer glaubt das Internet besteht nur aus eMail, Facebook und YouTube etc, der verpasst die aufregende Welt der embedded Ethernet Geräte. Wie wäre es, wenn man seine Geräte von überall auf der Welt kontrollieren kann? Die Solaranlage mit dem Smart-Phone vom Strand aus steuern, die Heizung zum Feierabend vom Büro aus einschalten und die Rolläden automatisch steuern? Aber nicht nur für das Eigenheim, natürlich auch für die Industrie ist dies von großer Bedeutung. So kann man seine Produktionsprozesse von jedem Ort der Welt ganz einfach über das Internet kontrollieren. Vorausgesetzt die Steuerung hat einen Zugang zum weltweiten Netz.

Und genau darum geht es hier...

Im Folgenden zeigen wir Schritt für Schritt, wie man mit geringem Aufwand innerhalb kürzester Zeit zu einem vollständigen embedded Webserver kommt, der einfache Steuerungsaufgaben übernehmen kann. Das Projekt ist absichtlich sehr einfach gehalten und soll nur das Prinzip beschreiben. Deutlich komplexere Systeme sind jederzeit problemlos möglich.

Die Projektanforderungen

Das Projekt soll folgenden Anforderungen gengen:

- Wenig externe Bauteile
- Steuerung von 2 Relais über ein Web-Interface
- Webseiten sollen auf Micro-SD Karte gespeichert werden und einfach austauschbar sein
- Steuerung der Relais über ein einfaches CGI Interface
- Fixe Netzwerkkonfiguration (IP Adresse)

Das Ziel ist ein einfacher „Internet-Schalter“. Also ein Gerät, mit dem man über eine Webseite z.B. zwei Geräte ein und aus schalten können soll.

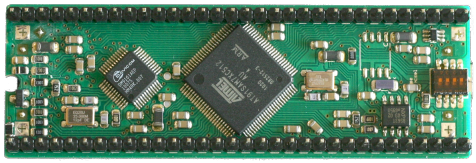
ARM7 Webserver Kompakt

Als Basis für das Projekt setzen wir das eNet-sam7X Modul ein, ein embedded Mikrocontroller Modul auf Basis der AT91SAM7XC512 CPU von Atmel, welches viel Leistung und viele I/O Schnittstellen auf kleinstem Raum bietet und sich ideal in eigene Applikation einbinden lässt.

Die Idee für die Entwicklung des Moduls ergab sich aus der Suche nach einer kompakten Steuerung, die sich einfach in C programmieren lässt, einen Ethernet Port besitzt und in der Lage ist auch komplexe Peripherie zu steuern sowie Messdaten direkt zu verarbeiten und speichern.

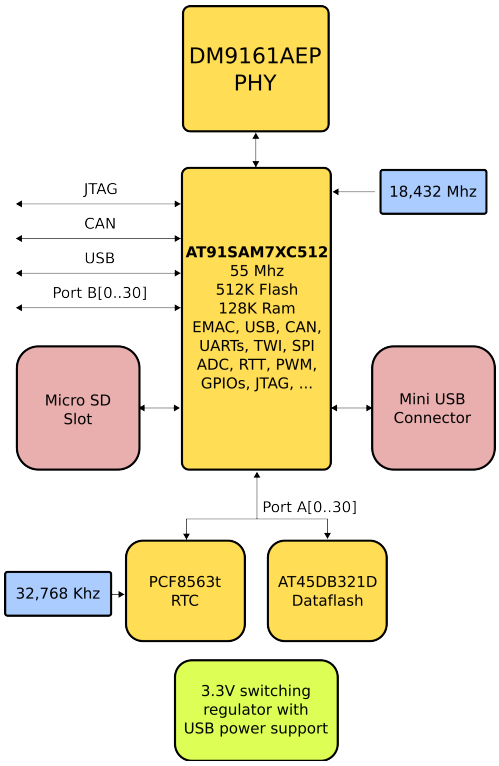
Die zweite wichtige Voraussetzung war, dass das Modul vollständig durch OpenSource Software unterstützt werden muss. Ein echtes Multitasking Betriebssystem mit vollständigem TCP/IP Stack sollte zum Einsatz kommen. Als Betriebssystem wird das OpenSource Echtzeit Betriebssystems Nut/OS eingesetzt.

eNet-sam7X



Im Herzen ganz groß, im Platz ganz klein.

Das Herz des Boards ist eine ARM7 CPU von Atmel, der AT91SAM7XC512. Dieser Mikrocontroller bietet alles, was das Entwickler Herz höher schlagen lässt. Neben 512 KB internem Flash für eigene Programme und Daten sowie 128KB Statischem RAM bietet er diverse Peripheriekomponenten wie z.B. 100 MBit Ethernet, 2 vollwertige und eine Debug UART, einen CAN-Bus, zwei SPI- und einen I2C Bus, ein I2S Interface, Timer, GPIOs, PWMs, A/D Wandler USB 2.0 und vieles mehr. Ergänzt wird der Mikrocontroller auf dem Board durch einen Ethernet PHY von Davicom, ein 4 MB Dataflash (ideal als großer Datenspeicher für logging Aufgaben, Webseiten, User Daten usw.), einer Batterie gepufferten Echtzeituhr (RTC) sowie einem 3.3V Schaltregler, einem Mini-USB Connector sowie einem Micro-SD Slot (Noch mehr Speicher für große Datenmengen). Und das alles auf der Fläche eines DIL64 ICs.



Viele Mikrocontroller Boards und Evaluations-Plattformen sind als eigenständiges Gerät konzipiert und schränken die Nutzbarkeit damit auf den gegebenen Form-Faktor sowie die vorhandene Peripherie ein. Das eNet-sam7X nicht! Es ist explizit als Drop-In Lösung zur Entwicklung eigener Steuerungssysteme und Web-Appliances gedacht und bietet dem Entwickler daher auf seinen zwei Steckerleisten alle freien Signale der SAM7 CPU, das Ethernet Interface, USB, MMC, JTAG sowie einen Spannungsausgang mit 3.3V, der zur Versorgung des Baseboards verwendet werden kann. Da das Board einen eigenen Schaltregler besitzt kann so die gesamte Schaltung einfach eine eine externe 5-24V Spannungsquelle angeschlossen oder z.B. über den USB Bus mit Strom versorgt werden.

Der Schaltplan des eNet-sam7X ist unter [1] zu finden.

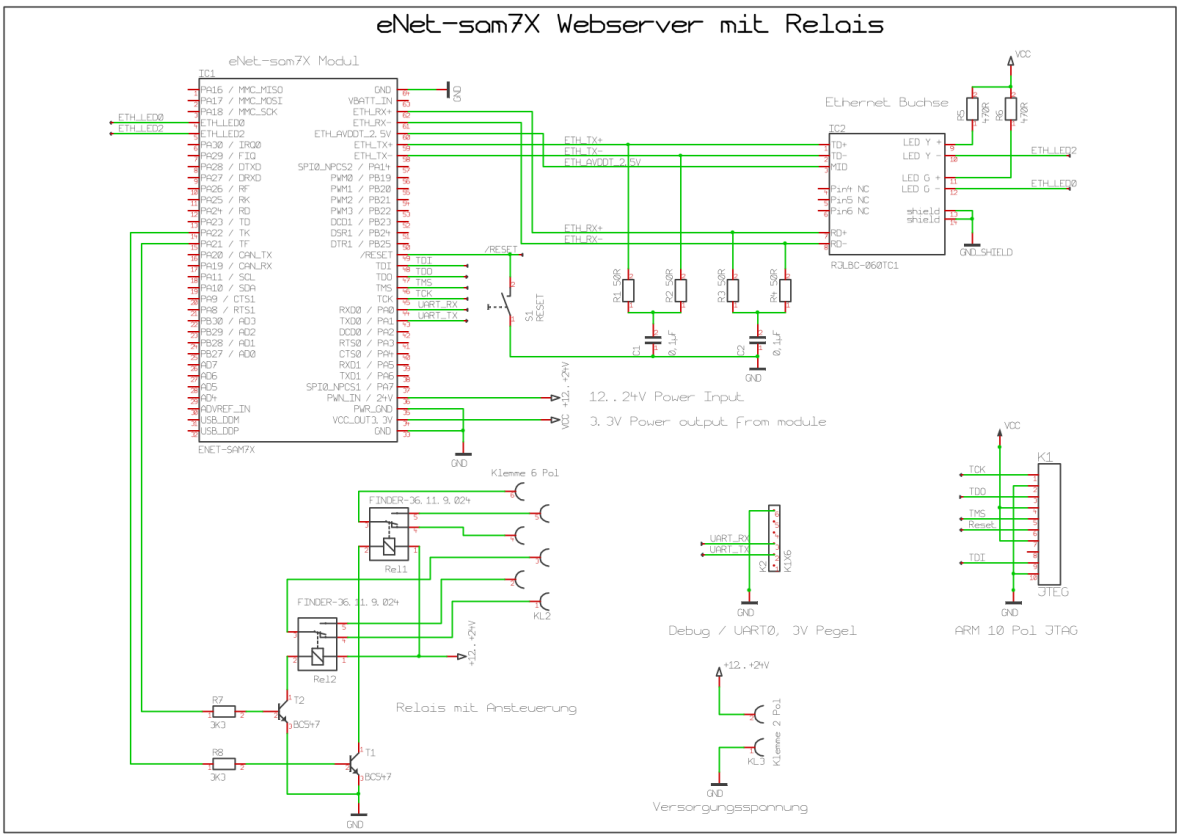
Die Schaltung

Wie einfach es ist, mit dem eNet-sam7X Modul zum Beispiel einen „Internet-Schalter“ zu bauen, zeigt das folgende Beispiel.

Achtung: Zu beachten ist, dass diese Schaltung lediglich ein Schaltungsbeispiel ist und bei der Verwendung von Netzspannung in eigenen Schaltungen diverse Vorschriften zu beachten sind, auf die an dieser Stelle nicht näher eingegangen wird.

Das Herz der Schaltung ist natürlich das eNet-sam7X Modul. Als externe Beschaltung ergibt sich lediglich noch die Ethernet Buchse (Hier eine mit Auto MDIX Funktionalität. D.h. automatischer Kreuzung des Ethernet Kabels) mit den Terminierungs-Widerständen, ein Reset-Taster, zwei Relais mit Ansteuertransistoren sowie ein JTAG Stecker (10 Polige ARM Belegung) und eine Stiftleiste um zu debug Zwecken auf die RX / TX Signale der UART0 zugreifen zu können. Die Belegung dieser Stiftleiste erlaubt direkt das Aufstecken des Sparkfun „FTDI Basic Breakout – 3.3V“ Moduls, einem einfachen 3.3V RS232 => USB Converter.

Die Schaltung kann direkt mit 12V – 24V betrieben werden, je nach der Betriebsspannung der gewählten Relais. Die 3.3V Versorgungsspannung für das Mikrocontroller Modul wird direkt auf dem Modul mit einem Schaltregler erzeugt. Das Modul stellt die 3.3V auf Pin 34 zur Verfügung, so dass weitere Hardware auf dem Baseboard gleich mit versorgt werden könnte.



Die beiden Relais sind über Ansteuertransistoren an die GPIO Pins PA21 und PA22 des AT91SAM7XC512 angeschlossen. Die Transistoren werden benötigt, da die CPU nicht genügend Strom liefern kann, um die Relais direkt zu schalten. Die Relais sind in diesem Beispiel als Wechsler ausgelegt und können über die Ausgänge an Klemme KL2 je einen Verbraucher (z.B. eine Lampe) schalten. Im Folgenden möchte ich zeigen, wie einfach es ist diese Relais über eine Webseite zu steuern.

Nut/OS

Ein modulares OpenSource Echtzeit-Betriebsystem mit TCP/IP Stack für Mikrocontroller

Eine komplexe Hardware wie diese ganz ohne Betriebssystem programmieren? Das ist aufwändig, kostet viel Zeit für die Einarbeitung und bietet lauter Fehlerquellen. Warum also nicht nach einer Lösung suchen, die uns das Entwickeln von Treibern weitestgehend abnimmt und ein Programmiermodell bietet, wie wir es vom PC her kennen? Das soll gehen? Auf einem Mikrocontroller?

Gesucht, gefunden! Das Ethernut Projekt [2] mit seinem OpenSource Echtzeit-Betriebssystem Nut/OS [3] ist genau die richtige Lösung. Das Ethernut Projekt wurde 2001 gegründet und seit dem wurde das Nut/OS zu einem sehr modularen und leicht konfigurierbaren Multitasking-Betriebssystem für Mikrocontroller entwickelt. Ursprünglich für 8 Bit Architekturen (in erster Linie Atmel AVR) entwickelt, unterstützt es heute eine ganze Reihe unterschiedlicher CPUs. Unter anderem den hier verwendeten AT91SAM7XC512 von Atmel. Nut/OS steht unter der BSD Lizenz und ist damit ohne Probleme für Closed- wie auch für OpenSource Projekte verwendbar.

Das Betriebssystem skaliert hervorragend mit den Möglichkeiten der jeweiligen Architektur. Durch den modularen Aufbau des Betriebssystem werden nur die Teile in eigene Programme eingebunden, die von der Anwendung tatsächlich benötigt werden. Das hält den Code klein und spart Ressourcen, von denen man auf einem Mikrocontroller bekanntlich nie genug hat.

Die Anpassung an das Zielsystem erfolgt in der Regel automatisch. Zur Feinabstimmung steht mit dem Nut/OS Konfigurator eine grafische Oberfläche unter Linux, Windows und OS X zur Verfügung.
Nut/OS bietet echtes Multitasking und ist dabei sicher und einfach anzuwenden. Sein kooperatives Multitasking garantiert, dass ein Thread die Kontrolle nur an eindeutig definierten Punkten abgibt. In den meisten Fällen kann so ohne zusätzliche Absicherung auf gemeinsam genutzte Ressourcen zugegriffen werden. Daraus ergibt sich kleiner und einfacher Code für die Anwendung, sowie ein geringes Risiko von Race-Conditions und Deadlocks. Deterministische Interrupt Latenzzeiten bieten hartes Echtzeitverhalten innerhalb festgelegter Zeitgrenzen, unabhängig von aktuell verfügbaren Ressourcen.
Aber das Beste zum Schluss, Nut/OS bring mit Nut/NET einen vollständigen TCP/IP Stack mit BSD artigem Socket Interface mit. Neben TCP und UDP Sockets sind diverse High-Level Protokolle implementiert. z.B. ARP, ICMP, PPP, DHCP, DNS, SNTP, FTP, TFTP, SYSLOG, HTTP, WINS (Subset) usw.

Das gesamte API Design von Nut/OS ist an den POSIX Standard angelehnt und verwendet bekannte Treiberkonzepte wie z.B. Character- und Block-Devices, Bus-Abstraktionen und der gleichen mehr. Das macht den Einstieg einfach und mittels der diversen Beispiele findet sich auch der Einsteiger schnell zurecht.

Erste Schritte: Die Vorbereitungen

Damit wir loslegen können, braucht es natürlich noch einen Cross-Compiler bzw. eine Toolchain für das Board. Das Nut/OS ist auf die Verwendung der GNU ARM Compiler Suite ausgelegt. Diese gibt es sowohl für Windows als auch für Linux und den Mac. Für Windows und Linux bietet sich zum Beispiel die eCross ARM Toolchain [4] oder die Yagarto Toolchain [5] an. Beide sind OpenSource und natürlich frei verfügbar. Sie enthalten den GNU C Compiler, Debugger sowie die Standard C Library (Newlib). Um an dieser Stelle nicht auszuufern, gehe ich nicht weiter auf die Installation ein und möchte auf unsere Webseite [6] verweisen.

Neben der Toolchain benötigen wir jetzt natürlich noch das Nut/OS Betriebssystem. Ab der Ethernut Version 4.9.11 wird die Unterstützung für das eNet-sam7X Board in der offiziellen Ethernut Distribution enthalten sein. Bis dahin kann man sich ein angepasstes Ethernut Paket auf der Webseite zum eNet-sam7X Board Support Package [6] herunter laden. Hier findet sich auch eine entsprechende Installationsanleitung.

Jetzt kann es losgehen: Der eigene Webserver

Alles ist soweit vorbereitet, jetzt kann es losgehen! Der eigene Webserver soll erstellt werden. Was auf den ersten Blick furchtbar kompliziert klingen mag, ist den zweiten Blick eigentlich ganz einfach, da das meiste bereits von den Nut/OS Bibliotheken abgenommen wird.

Unser Programm muss dafür mehrere Dinge tun:

1. Hardware initialisieren (GPIO Pins Konfigurieren, UART Treiber laden für Debug-Zwecke, Netzwerktreiber initialisieren, MMC Treiber initialisieren.
2. Micro-SD Karte mounten (die MMC Karte dient uns als Speicherort für die Webseiten)
3. Netzwerk konfigurieren (IP Adresse einstellen)
4. CGI Funktionen registrieren und Webserver Threads starten.
5. Auf eingehende Verbindungen von einem Webbrowser warten und diese bearbeiten. Relais setzen, wenn die korrekte Anforderung vom Webbrowser geschickt wurde.

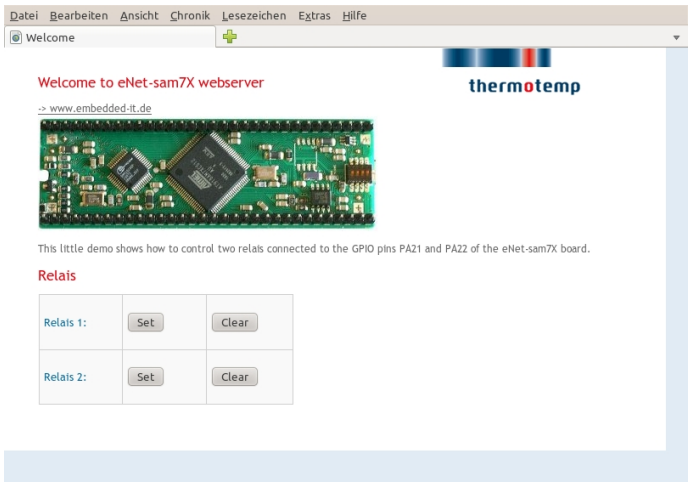
All das passt in ca. 250 Zeilen Code! Das meiste erledigen die Nut/OS Webserver Funktionen bereits im Hintergrund. Der Übersichtlichkeit halber möchte ich an dieser Stelle den Programmcode nur ausschnittsweise besprechen. Der vollständige Sourcecode kann unter [7] heruntergeladen werden.

Zunächst wollen jedoch kurz das Benutzer Interface unseres „Internet-Schalters“ vorstellen...

Die Webseite

Da wir ja ein embedded Gerät bauen, hat unsere Anwendung / bzw. unser Gerät natürlich kein klassisches grafisches Benutzerinterface. Es ist ja gerade unsere Intention ein Gerät zu bauen, welches von überall auf der Welt gesteuert werden kann. Also verlagern wir unser Benutzerinterface einfach auf eine Webseite. Auf dieser Webseite können wir natürlich wie auf einer klassischen Webseite diverse grafische Gestaltungselemente, Info- und Hilfs-Texte unterbringen und zusätzliche Informationen ablegen. Das eigentlich Interessante jedoch ist, dass wir Betriebszustände unseres Gerätes, Messwerte und diverses mehr darstellen können. Hierzu eignen sich hervorragend die Web-2.0 Technologien. Ein Schlagwort „Ajax“ ist sicher jedem ein Begriff. An dieser Stelle wollen wir jedoch bei einem einfachen Beispiel bleiben und über einfache Web-Formulare mit unserer Anwendung kommunizieren. Wer jedoch selber mit Ajax und anderen Web-2.0 Technologien experimentieren möchte, dem kann ich an dieser Stelle nur die diversen freien und kommerziellen Javascript Bibliotheken wie z.B. JQuery, ExtJS etc. empfehlen. Ein nettes Beispiel für eine ähnliche Anwendung mit dynamischer Ajax Unterstützung findet man unter [9]. Es handelt sich dabei um die eNet-sam7X basierte Sollaranlagen Steuerung meines Kollegen.

Die Webseite für unseren Webserver könnte so aussehen:



Der eigentliche funktionale Kern der Seite ist hier ein einfaches Formular. Die vier „Set“ und „Clear“ Buttons (je einer pro Relais) sind dabei als einfache Submit-Buttons mit einem eindeutigen Namen (set1, clear1, set2, clear2) markiert.

Der HTML Code für das Formular sieht dabei wie folgt aus:

```
    <form action="/cgi-bin/relais.cgi" method="post">

        <table  bgcolor="#E0E0E0" height="130" width="300">
            <tr>
                <TD height="29" width="100">Relais 1:</TD>
                <TD height="29" width="100"><input type = "submit" name="set1" value="Set" size="8"></TD>
                <TD height="29" width="100"><input type = "submit" name="clear1" value="Clear" size="8"></TD>
            </tr>
            <tr>
                <TD height="29" width="100">Relais 2:</TD>
                <TD height="29" width="100"><input type = "submit" name="set2" value="Set" size="8"></TD>
                <TD height="29" width="100"><input type = "submit" name="clear2" value="Clear" size="8"></TD>
            </tr>
        </table>

    </form>
```

Klickt der Benutzer nun z.B. auf den Button „Set“ in der Zeile Relais 1, so sendet der Webbrowser einen HTTP Post Request an unseren Webserver und überträgt dabei den Namen des Buttons an die CGI Funktion („`cgi-bin/relais.cgi`“). Diese wertet die übertragenen Parameter aus und schaltet anhand der Button-Namen das entsprechende Relais entweder ein oder aus.

So einfach lässt sich ein „Internet-Schalter“ realisieren. Die oben abgebildete Webseite liegt unserem Beispiel natürlich bei.

Das Programm

Nachdem wir jetzt bereits unser Benutzerinterface (die Webseite) vorgestellt haben, fehlt noch das eigentliche Programm, welches wir auf das eNet-sam7X Board laden und dort ausführen können.

Fangen wir also zunächst mit dem Hauptprogramm an. Natürlich besitzt ein Nut/OS Programm auch eine `main()` Funktion! Wie bei jedem anderen Programm auch, ist dies der Einsprungpunkt und der Start der eigentlichen Anwendung. Nut/OS selbst erledigt zuvor bereits einiges an Initialisierung, darum muss sich der Anwender jedoch nicht kümmern. Eine Besonderheit gibt es dennoch: wie bereits erwähnt ist Nut/OS ein Multitasking Betriebssystem und setzt dabei auf kooperatives Multitasking. Das bedeutet, dass das eigentliche Hauptprogramm (also die `main()` Funktion) selbst bereits ein eigenständiger Thread ist. Dies wird am Ende der `main()` Funktion deutlich, wo das Programm in eine Endlosschleife läuft. Zuvor wird die Priorität des main-Threads auf die niedrigste Stufe gesetzt. Die eigentliche Arbeit erledigen später die Webserver Threads.

Das Hauptprogramm

```
int main(void)
{
    uint32_t baud = 115200;
    uint8_t  i;
    int      rc;

    uint8_t  mac[] = DEFAULT_MAC; /* MAC address that shall be assigned to the ethernet device */


    /* Initialize the GPIOs used to control the relais */
    GpioPinConfigSet(PORT_RELAIS, PIN_RELAIS1, GPIO_CFG_OUTPUT);
    GpioPinSet(PORT_RELAIS, PIN_RELAIS1, 0);

    GpioPinConfigSet(PORT_RELAIS, PIN_RELAIS2, GPIO_CFG_OUTPUT);
    GpioPinSet(PORT_RELAIS, PIN_RELAIS2, 0);
```

Die `main()` Funktion unter Nut/OS bekommt keine Parameter übergeben und ist daher als void deklariert. Im Folgenden werden einige Variablen definiert, die später benötigt werden. Interessant sind hier nur die Baudrate sowie die MAC Adresse. Diese ist im Programmkopf als Konstante hinterlegt. Dieses Beispiel verwendet im übrigen eine MAC Adresse, bei der das „lokal verwaltet“ Flag gesetzt ist. Das bedeutet, dass diese Adresse nicht weltweit eindeutig ist und wir selber sicherstellen müssen, dass die MAC Adresse in unserem Netzwerk eindeutig ist. Wenn man vor hat ein Produkt mit einem Netzwerk Interface zu vertreiben, sollte man sich auf jeden Fall einen entsprechenden Adressraum eindeutiger MAC Adressen registrieren lassen.

Nach der Variablendefinition werde als erstes die GPIO Pins, die mit unseren Relais verbunden sind initialisiert und auf low-Pegel gesetzt, so dass die Relais nach dem Einschalten zunächst einmal abfallen. Im nächsten Schritt werden die Peripherie-Schnittstellen Initialisiert...

```
    /*
     * Initialize the uart device.
     */
    NutRegisterDevice(&DEV_UART0, 0, 0);
    freopen(DEV_UART0_NAME, "w", stdout);
    freopen(DEV_UART0_NAME, "r", stdin);
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);

    printf("\n\neNet-sam7X Webserver demo... Nut/OS Version %s\r\n", NutVersionString());
```

Treiber werden unter Nut/OS im Allgemeinen mit der Funktion `NutRegisterDevice()` registriert. Diese Funktion initialisiert die Schnittstelle und stellt sicher, dass unser Programm über den Device Descriptor bzw. den Device Namen (hier die konstante `DEV_UART0_NAME`) mit der Schnittstelle kommunizieren kann. In diesem Fall initialisieren wir die UART0, öffnen die Standard I/O File-Descriptoren auf dieser Schnittstelle und weisen Ihr per `_ioctl()` noch die Baudrate (hier 115200 Baud) zu. Und schon kann per `printf()` beliebiger Text ausgeben werden. So kann man leicht per Hyperterminal (Windows) oder Minicom (Linux) die Ausgaben des Programms verfolgen oder das Programm durch Eingaben steuern. Debugging wird so zum Kinderspiel.

An dieser Stelle wird es Zeit die MMC Karte zu initialisieren und das Dateisystem zu mounten.

```
    /* Initialize the MMC card and mount the filesystem */
    init_mmc();
```

Das Initialisieren der MMC Karte sowie das Mounten des FAT Datei Systems erledigt die Funktion `init_mmc()`. Diese Funktion stelle ich aus Platzgründen an dieser Stelle nicht im Detail dar, sie ist natürlich im Sourcecode der Demo vorhanden. Kurz zusammen gefasst greifen hier drei Komponenten ineinander. Der Dateisystem Treiber für das FAT Dateisystem (im Nut/OS Jargon PHAT Filesystem), der SPI-Bus Treiber für den physikalischen Zugriff auf die MMC Karte sowie schließlich der MMC Karten Treiber für den logischen Zugriff auf die Karte. Zunächst werden nun die Treiber initialisiert und miteinander verknüpft so dass anschließend das Dateisystem gemountet werden kann. Danach kann auf das Dateiesystem zugegriffen werden. Auch hier bietet Nut/OS eine sehr komfortable API, so dass wir einfach mit den Funktionen open / fopen / read / write usw. auf die Dateien auf der MMC Karte zugreifen können. Ganz so wie wir es vom PC gewöhnt sind.

Da Nut/OS mehrere unterschiedliche Dateisysteme gleichzeitig mounten kann, müssen wir beim Zugriff auf eine Datei den Pfad der Datei immer inklusive des Mountpoints angeben. So öffnet ein `fopen("PHAT0:/index.html", "r+");` die Datei „`index.html`“ Im Wurzelverzeichnis der MMC Karte.

Im nächsten Schritt wird das Ethernet Interface initialisiert.

```
    do {
        rc = NutRegisterDevice(&DEV_ETHER, 0, 0);
        if (rc != 0) {
            puts("Registering ethernet device failed! Is a cable connected? Retrying...\r\n");
        }
    } while (rc != 0);

    printf("Configuring %s... ", DEV_ETHER_NAME);

    if (NutNetIfConfig2(DEV_ETHER_NAME, mac, inet_addr(DEFAULT_IPADDR),
                    inet_addr(DEFAULT_NETMASK), inet_addr(DEFAULT_GATEWAY)) == 0) {
        puts("OK\r\n");
    } else {
        puts("Failed\r\n");
        NutSleep(500);
        while(1);
    }
```

Nachdem das Ethernet Device erfolgreich mit `NutRegisterDevice()` registriert wurde, wird per `NutNetIfConfig2()` eine statische IP Adresse, eine Subnetz-Maske sowie ein Standard Gateway festgelegt. In unserem Beispiel verwenden wir folgende Einstellungen:

```
IP:          192.168.1.200
Netmask:     255.255.255.0
Gateway:     192.168.2.254
```

Die Einstellungen sind im Programm-Kopf als Konstante hinterlegt und können den eigenen Bedürfnissen angepasst werden. Natürlich könnte man die IP Adresse auch dynamisch per DHCP abfragen. Hier sei auf die diversen Beispiele verwiesen, die in dem Ethernut / Nut/OS Paket enthalten sind.

Damit unser Webserver jetzt auch weiß, wo die Webseiten zu finden sind, müssen wir den „Webroot“, also das Wurzelverzeichnis unseres Webservers registrieren. Die Konstante `HTTP_ROOT` zeigt dabei wieder auf unsere MMC Karte: `PHAT0:/`

```
    /* Register root path. */
    printf("Registering HTTP root '" HTTP_ROOT "'...");
    if (NutRegisterHttpRoot(HTTP_ROOT)) {
        puts("failed");
        for (;;);
    }
    puts("OK");
```

Damit das Programm auf Eingaben / Steueranweisungen durch den Webbrowser des Benutzers reagieren kann, bedarf es noch einer Kommunikations-Schnittstelle mit dem Benutzer. In der Welt der Web-Anwendungen wird dies im Allgemeinen über CGI Funktionen erledigt. Auch unser Programm enthält ein CGI welches die beiden an das eNet-sam7X Modul angeschlossenen Relais steuert.

Im Gegensatz zu einem PC Webserver stellt ein CGI hier jedoch kein eigenes Programm dar sondern eine Funktion. Diese wird vom Webserver aufgerufen, wenn der User die entsprechende URL des CGIs im Webbrowser eingibt oder der Webbrowser Formulardaten an ein CGI schickt. Diese CGI Funktion muss vor Verwendung ebenfalls registriert werden. Registrieren bedeutet hier, dass der Funktion ein virtueller Name bzw. eine URL zugewiesen wird. Zuvor wird jedoch noch festgelegt, in welchem unter-Ordner des Webroots dieses CGI zur Verfügung gestellt werden soll.

```
    /*
     * Register the path where CGIs shall be located. This is a virtual path only
     * as CGIs are virtual files too
     */

    NutRegisterCgiBinPath("cgi-bin/");

    /*
     * Register some CGI samples, which display interesting
     * system informations.
     */
    NutRegisterCgi("relais.cgi", CGI_Relais);
```

Unser CGI lässt sich jetzt also über die URL `http://192.168.1.200/cgi-bin/relais.cgi` ansprechen.

Als letztes bleibt nur noch den eigentlichen Webserver zu starten und anschließend das Hauptprogramm in einer Endlosschleife enden zu lassen. Aber **Achtung:** Wie bereits erwähnt nutzt Nut/OS ein kooperatives Multitasking. Beim kooperativen Multitasking findet ein Thread-Wechsel immer nur an definierten Punkten statt und es muss jederzeit sichergestellt sein, dass die CPU an einen anderen Thread abgegeben werden kann. Eine einfache Endlosschleife würde die Programmausführung „einfangen“ und die CPU nie wieder für andere Threads freigeben. Ein einfaches `NutSleep()` hilft hier weiter und lägt den Haupt-Thread an dieser Stelle schlafen.

Aber zurück zur eigentlichen Webserver Funktion. Diese ist als Thread implementiert. Damit wir mehrere HTTP Verbindungen auf einmal bearbeiten können starten wir diesen Thread gleich mehrfach.

```
    /*
     * Start the server threads.
     */
    for (i = 0; i <= 8; i++) {
        char thname[] = "httpd0";

        thname[5] = '0' + i;
        NutThreadCreate(thname, Service, (void *) (uintptr_t) i, 4096);
    }

    printf("Web server on %s:80 ready\r\n", DEFAULT_IPADDR);

    /*
     * We could do something useful here. In this case we are just setting
     * the main thread to the lowest priority and are just waiting.
     */
    NutThreadSetPriority(254);
    for (;;) {
        NutSleep(60000);
    }
    return 0;
}
```

Der Server-Thread

Der eigentliche Webserver besteht aus dem Server-Thread. Dieser sorgt dafür, dass ein TCP Socket geöffnet wird und auf Port 80 auf eine eingehende Verbindung wartet. Dabei schläft der Thread so lange, bis der Webbrowser eine entsprechende Anfrage an den Webserver sendet.

Sobald eine Verbindung zustande gekommen ist, unser Webserver also ein „accept“ auf dem Socket ausgeführt hat, wird dem Socket ein Filedescriptor zugewiesen, der anschließend der Funktion „`NutHttpProcessRequest()`“ übergeben wird.

Diese Funktion ist das eigentliche Herzstück unseres Webservers. Sie ist eine fertig implementierte API Funktion von Nut/OS und sorgt dafür, dass die HTTP Anforderungen des Webbrowsers ausgewertet und die angeforderten Daten (Dateien oder dynamische Inhalte) an den Webbrowser ausgeliefert werden.

Zusammengefasst sieht der Code dazu wie folgt aus:

```
THREAD(Service, arg)
{
    TCPSOCKET *sock;
    FILE *stream;
    uint8_t id = (uint8_t) ((uintptr_t) arg);

    for (;;) {
        if ((sock = NutTcpCreateSocket()) == 0) {
            ...
        }

        NutTcpAccept(sock, 80);

        ...

        if ((stream = _fdopen((int) ((uintptr_t) sock), "r+b")) == 0) {
            printf("[%u] Creating stream device failed\n", id);
        } else {
            NutHttpProcessRequest(stream);
            fclose(stream);
        }

        NutTcpCloseSocket(sock);
    }
}
```

Aber wie kommen jetzt die Steueranweisungen des Benutzers zu den Relais?

Wir erinnern uns, dass im Hauptprogramm ein CGI registriert wurde. Dieses CGI wurde der URL `/cgi-bin/relais.cgi` zugewiesen. Die URL ist relativ zum Webroot des Servers zu lesen.

Hat der Webbrowser nun einen GET oder POST Request mit dieser URL an unseren Webserver gesendet, so erkennt die Funktion `NutHttpProcessRequest()`, dass es sich dabei um unser CGI handelt und ruft dieses auf. Dieses braucht die Anfrage dann nur noch auszuwerten und die Relais entsprechend zu setzen. So schließt sich der Kreis...

Das Relais CGI

Bei einem Webserver wird immer zwischen statischem Inhalt (HTML Dateien, Bildern etc.) sowie dynamischen Inhalt unterschieden. Ein CGI stellt die einfachste Form von dynamischem Inhalt dar. Es kann dabei entweder nur auf Eingaben reagieren oder eben auch vollständige HTML Seiten dynamisch generieren und an den Webbrowser ausliefern. In unserem Beispiel wollen wir jedoch nur auf Benutzereingaben reagieren und keine Ausgaben produzieren. Wie wir im folgenden Code Ausschnitt sehen, wird trotzdem mindestens ein HTTP Header zurück an den Webbrowser geschickt. Normalerweise mit dem Status Code „200 OK“. In unserem Fall senden wir jedoch ein „204 No content“ und signalisieren dem Browser damit, dass er auf keine Antwort vom CGI warten (und diese ggf. darstellen) soll.

Das CGI sieht (in gekürzter Form) wie folgt aus:

```
int CGI_Relais(FILE * stream, REQUEST * req)
{
    NutHttpSendHeaderTop(stream, req, 204, "No content");
    NutHttpSendHeaderBottom(stream, req, "", -1);

    if (req->req_method == METHOD_POST) {
        char *name;
        char *value;
        int i;
        int count;

        NutHttpProcessPostQuery(stream, req);
        count = NutHttpGetParameterCount(req);
        /* Extract count parameters. */
        for (i = 0; i < count; i++) {
            name = NutHttpGetParameterName(req, i);
            value = NutHttpGetParameterValue(req, i);

            if (strcmp(name, "set1") == 0) {
                GpioPinSet(PORT_RELAIS, PIN_RELAIS1, 1);
                printf("=> Set relais 1\n");
            } else
            if (strcmp(name, "clear1") == 0) {
                GpioPinSet(PORT_RELAIS, PIN_RELAIS1, 0);
                printf("=> Clear relais 1\n");
            } else

            ...

        }
    }

    fflush(stream);

    return 0;
}
```

Zunächst wird der HTTP Header zurück an den Browser geschickt. Anschließend überprüfen wir, ob es sich bei der Anfrage um einen POST Request handelt (GET Requests werden ignoriert) und werten ggf. die übergeben Parameter aus. Dabei wird nur der Parameter Name betrachtet. Klickt der Benutzer auf der Webseite zum Beispiel auf den „Set“ Button von Relais 1, so wird der Parameter „`set1`“ übertragen. Ein Einfacher `strcmp()` reicht zur Auswertung aus. Anschließend wird das entsprechende GPIO Pin entweder auf high (set) oder auf low (clear) gesetzt.

Und fertig ist unser Internet-Schalter!

Compilieren / Installieren

Jetzt bleibt nur noch übrig das Programm zu compilieren und anschließend auf das Board zu übertragen. Dem zum Download bereit stehenden Code liegt neben den Webseiten natürlich auch ein passendes Makefile bereit. Die Pfade im Makefile müssen ggf. der eigenen Nut/OS Installation angepasst werden. Anschließend reicht jedoch ein

```
make
```

auf der Kommandozeile um das Programm zu übersetzen. Es wird eine Datei „`webserver.bin`“ erzeugt, die anschließend z.B. per JTAG in das Flash des Mikrocontrollers geschrieben werden kann. Hierfür eignet sich das OpenSource JTAG Debugging Tool „OpenOCD“ hervorragend. Passende JTAG Adapter sind günstig (ab ca. 30 EUR) z.B. von Olimex (z.B. ARM-USB-Tiny) oder Egnite (Turtelizer 2) zu bekommen. Alternativ kann man natürlich auch das SAM-BA Tool von Atmel zum Programmieren des eNet-sam7X Boards über den USB Port dafür verwenden. Beide Methoden werden im eNet-sam7X Handbuch [8] detailliert anhand von Beispielen erklärt.

Wird das eNet-PLC Baseboard für die Entwicklung verwendet, so ist das USB-JTAG Interface bereits vollständig auf dem Baseboard vorhanden und kann direkt mit OpenOCD eingesetzt werden.

Zur Inbetriebnahme muss man nur noch eine Micro-SD Karte (Max 2GB, Fat formatiert) mit der Webseite und den zugehörigen Dateien bespielen und in den Sockel auf dem eNet-sam7X Modul einlegen, und schon ist der „Internet-Schalter“ bereit seine Arbeit aufzunehmen.

Fazit / Ausblick

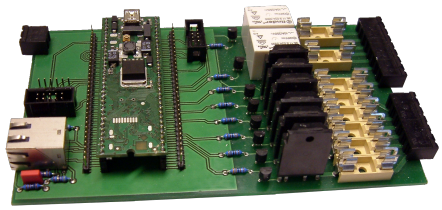
Im Rahmen dieses Artikels konnten wir natürlich nur ein ganz einfaches Beispiel konstruieren. Nichts desto trotz ist schon dieses einfache Beispiel eine vollständige und benutzbare Applikation. Ein über das Internet steuerbarer Schalter mit zwei eigenständigen Umschaltern...

Eine realistische Applikation könnte leicht ein deutlich umfangreicheres Benutzer Interface bieten, dynamische Webseiteninhalte einbinden und natürlich komplexe Steuerungs- und Regelungsanwendungen implementieren. Die Webserver Funktionalität alleine lastet die AT91SAM7XC512 CPU nicht einmal annähernd aus. Die meiste Zeit wartet das Programm ja schließlich auf Anfragen aus dem Internet. So bleibt hinreichend Rechenkapazität für eigene Ideen und Regelungs-Algorithmen übrig.

Ideen für weitere und ggf. komplexere Applikationen finden sich schnell und bleiben der Kreativität der Leser überlassen ;-)

Ein Beispiel für eine komplexere Steuerung (Steuerung für eine Solar-Thermie Anlage) mit näheren Infos inkl. Einer Live-Anzeige der aktuellen Betriebsdaten findet sich unter:

http://solar.thermotemp.de/



Ein guter Startpunkt um vielleicht Ideen für eigene Projekte rund um das eNet-sam7X Modul zu entwickeln.

Eine weitere Idee wäre die Einbindung von Lua als embedded Scriptsprache. Da embedded Lua bereits Bestandteil der Nut/OS Umgebung ist lässt sich ein einfaches webbasiertes Programmier-Modell entwickeln, so dass auch Anwender Programme für das eNet-sam7X Modul entwickeln können, die bislang wenig Erfahrungen in der Mikrocontroller Welt haben und trotzdem alle Vorzüge eines embedded Systems nutzen wollen.

Quellenverzeichnis:

[1] http://www.embedded-it.de/pdf_free/eNet-sam7X-schematic.pdf

[2] http://www.ethernut.de/

[3] http://www.ethernut.de/de/firmware/index.html

[4] http://www.embedded-it.de/bsp/eCross.php

[5] http://www.yagarto.de

[6] http://www.embedded-it.de/bsp/eNet-sam7X_bsp.php

[7] http://www.embedded-it.de/bsp/demos/webserver_relais_demo.tar.gz

[8] http://www.embedded-it.de/pdf_free/eNet-sam7X-hardware-manual.pdf

[9] http://solar.thermotemp.de/