Häufig kommen Fragen auf, was man denn am besten nehmen kann, wenn man sich in die Programmierung der ARM-Prozessoren einarbeiten will. Sicherlich gibt es eine Vielzahl von Evaluationskits verschiedener renommierter Hersteller für deren Produkte. Aber es geht auch einfacher und preiswerter. Die derzeit (Herbst 2012) bei Pollin angebotene Fernbedienung "Betty" (Swissbetty) ist ein recht ansehnlich ausgestattetes Handgerät mit einem ARM7TDMI Prozessor LPC2220 von NXP. Als Fernbedienung ist sie eher ein Versager, denn ihr Stromverbrauch ist viel zu hoch für Batterien, so daß sie mit Akkus betrieben und allweil in eine Ladeschale gesteckt werden muß. Aber als Evaluationsboard ist sie unschlagbar günstig, hat sie doch neben sehr viel Flashrom auch noch ein grafisches Display und einiges an eingebauter Peripherie.

1 Wozu ein Eval-Board?

Die Betty ist ursprünglich eine Fernbedienung und damit für einen Verwendungszweck gedacht, der völlig außerhalb der Sphäre des Programmierens und der Mikrocontroller-Schaltungstechnik liegt.

Ein Evaluationsboard verfolgt einen ganz anderen Zweck, nämlich jemandem zu ermöglichen, sich mit Hardware und Programmierung einer Prozessorfamilie vertraut zu machen und später dann das gewonnene Wissen bei eigenen Entwicklungen einsetzen zu können. Auch das Ausprobieren von Algorithmen und Codestücken und deren spätere Wiederverwendung ist ein wesentlicher Zweck von Evalboards.

Das Herstellen von finalen Anwendungen, also der eigentliche Gerätebau ist jedoch außerhalb des eigentlichen Zweckes von Evalboards. Manchmal geht so etwas, dann kann man froh sein darüber, aber zumeist ist es sinnvoller, für echte Anwendungen ein genau dafür entwickeltes Stück Hardware zu benutzen.

Die Software vom Evalboard kann man jedoch zu großen Teilen weiterverwenden, wenn man beim darin Herumspielen keinen Spaghetticode produziert hat.

2 Die Bettybase

Meine Idee ist nun, für die Betty eine neue Firmware aufzusetzen, die eben keine Fernbedienung ist, sondern einem Einsteiger so systematisch wie möglich die verschiedenen Aspekte der Firmwareentwicklung nahe bringt. Ich hab sie einfach ***BettyBase*** genannt. Sie kommt in den ersten der zwei Flashroms und kann Apps starten, die separat von BettyBase entwickelt und gelinkt worden sind. Dazu gibt es ein kleines API, mit dem die Apps auf Dienste der BettyBase zurückgreifen können (aber nicht müssen).

3 Was braucht man denn so?

Nun, zu allererst natürlich eine Swissbetty und einen Satz von 2 AAA-NiMH Akkus, denn die beiliegenden Akkus sind überlagert und unbrauchbar.

Als nächstes muß man sich einen oder zwei Adapter bauen. Einen für die serielle Schnittstelle und den anderen für einen JTAG-Adapter sofern man ihn hat. Keine Sorge, es geht alles auch ohne JTAG.

Auf der Rückseite der Betty sieht man bei abgenommenem Batteriedeckel einen Steckverbinder aus zwei Reihen Pinheads im üblichen 2.54 mm Raster. Das ist der Steckverbinder, für den man sich die besagten Adapter basteln muß. Im einfachsten Fall ist das ein Stück doppelreihige Buchsenleiste mit ein paar Litzen drangelötet - aber die Lötstellen lieber mit Schrumpfschlauch überziehen, sonst ist alle Nase lang irgendein Draht abgebrochen..

Wer einen gut funktionierenden JTAG-Adapter und die dazu passende Software hat, wird sich einen zweiten Adapter basteln, sein JTAG-Equipment anschließen und damit bequem programmieren können.

Zum Kennenlernen der Betty und zu deren Programmierung ist ein JTAG-Equipment jedoch nicht zwingend erforderlich.

Wichtig hingegen ist ***Literatur***. Ladet euch also nicht nur das Usermanual des Prozessors von NXP herunter, sondern auch die technischen Unterlagen von ARM.

Als nächstes ist Übersetzungssoftware (neudeutsch *Toolchain*) erforderlich. Ich selbst arbeite beruflich mit dem Original-Compiler von ARM und habe demzufolge selbigen auch für das Betty-Projekt benutzt. Aber ich habe mir Mühe gegeben, möglichst keine Spezereien zu verwenden und weitestgehend simples ANSI-C zu schreiben. Damit sollten sich eigentlich alle Quellen problemlos mit jedem anderen Compiler übersetzen lassen.

Obendrein habe ich es vermieden, irgendeine IDE zu benutzen. Stattdessen gibt es ein simples Batchfile, wo alle Tool-Aufrufe drinstehen. So kann ein jeder seinen eigenen Lieblingseditor benutzen und zur Anpassung an die eigene Toolchain die Batchdatei editieren oder sich ein Makefile machen etc.

Natürlich kann man auch eine Demoversion vom "Keil" herunterladen. Man muß dann jedoch darauf achten, mit der Codegröße innerhalb von 32 K zu bleiben. Obendrein sind die "Keil"-Demos zeitbegrenzt, d.h. nach einigen Monaten stellen sie ihren Dienst final ein. Den "Keil" herunterzuladen kann ich uneingeschränkt auch denjenigen empfehlen, die aus was für Gründen auch immer damit nicht umgehen wollen, denn dort finden sich eine Menge Sachinformationen in Form von .chm Helpfiles und Demo- bzw. Beispielprogramme, die einen Blick wert sind.

Mir ist das Nichtverwenden irgendeiner IDE ein echtes Anliegen!
Als Newcomer sollte man lernen, mit der Sache selbst umgehen zu lernen, also den Chip kennen zu lernen und den Compiler selbst richtig aufrufen zu können. Beim bloßen Knöpfchendrücken in der IDE lernt man von

der Sache selbst so gut wie nix. Zum erfolgreichen Entwickeln braucht man auch einen Debugger nicht wirklich. Weitaus besser ist es, beim Codeschreiben strategisch denken zu lernen. Und: mit der BettyBase hat man ja bereits eine funktionable Firmware auf der Maschine, auf die man sehr bequem aufsetzen kann. Ich habe beim Schreiben der BettyBase ja auch keinen Debugger gebraucht. Aber gut, verrennen wir uns lieber nicht. Ich bin einem guten Debugger nicht abgeneigt, aber bei wirklich allen Systemen, die ich kenne, gibt es keinen wirklich echten Standalone-Debugger (mal von Ollie und vom uralten Borland-Debugger abgesehen).

Stattdessen sind gerade die Debugger für Embedded Systeme immer sehr fest in eine jeweilige IDE eingebunden. Der Grund dafür ist zumeist, daß nur durch die Einbindung in eine IDE der menschlich lesbare Bezug zwischen Maschinencode und Quellzeilen hergestellt werden kann. Ein Standalone-Debugger könnte nur im echten Assemblerniveau arbeiten - und dieses ist gerade beim ARM-Assemblercode keine leichte Nuß.

Will man also partout einen Debugger, dann bedeutet dies die Festlegung auf eine bestimmte Toolchain.

Soviel zu Toolchains...

Für die Verwendung der Betty als Eval-Board sollte man mindestens noch eine kleine Hardwareänderung anbringen: Anstelle der Kontakte, mit denen man die Betty in die Ladeschale steckt, sollte man sich einen passablen Steckverbinder einbauen, damit man die Betty quasi per Nabelschnur mit Strom versorgen kann, während man sie in den Händen hält.

4 Toolz und so

Für das reine Aufsetzen und Übersetzen von Quellen reichen selbstverständlich ein Editor und die Übersetzungstools aus. Aber es gibt ja noch viel mehr im Leben eines Mikrocontrollers mit Userinterface. Da wären solche Dinge wie Fonts, Bilder und Icons, Menüs usw. bis hin zu Audiofiles.

Es wäre Unfug, die gleichen Formate wie auf dem PC zu verwenden mit Truetype-Fonts und am PC üblichen Formaten von Bildern (BMP,JPG,PNG usw). Bei den Menüs hört es dann ganz auf, da ist auf einem Ding wie der Betty was ganz anderes gefragt als am PC.

Also hat man die Wahl, entweder mit dem auszukommen, was ich in die BettyBase bereits eingebaut habe oder sich ein paar Tools selbst zu schreiben, mit denen man all die Dinge in mikrocontrollergerechte Formate wandeln kann, die man haben will.

An Fonts wird es wohl vorerst keinen Mangel haben, denn es sind derzeit folgende Fonts eingebaut:

- ein simpler 5x7 Font
- ein simpler 6x13 Font
- Helvetica 12 regular
- Helvetica 12 bold
- Helvetica 14 regular
- Helvetica 14 bold
- Lucida 20

Wer sich damit näher befassen und sich ein Font-Tool schreiben will, lese im Kapitel über die Fonts weiter.

Die Anzeige von Bildern ist derzeit noch nicht implementiert. Wer es selbst tun will, sei hiermit dazu herzlich eingeladen. Ich stelle mir das so ähnlich vor wie meine simple Kompression für "Landkarten auf dem Mikrocontroller" (siehe "http://www.mikrocontroller.net/"). Das dürfte recht platzsparend und ausreichend schnell sein. Eine erste Idee:

```
const char MyPicture[] =
{ byte dX;
  byte dY;
  bytes gepacktePixels...
};
```

Die Pixel wiederum als Stream von je 2 Bits in eine entsprechende Anzahl Bytes gepackt. Einen Kompressionsalgorithmus für 2 Bit große Pixel halte ich derzeit für eher nicht so wichtig.

Menüs braucht man eigentlich immer und das in die BettyBase eingebaute Menüsystem ist generisch, so daß man recht frei ist in der Art, es zu benutzen. Allerdings sehe ich durchaus einen Bedarf, sich ein ordentliches grafisches Tool für den PC zu schreiben, um dort das Menü bequem entwerfen zu können. Dies wäre dann ein Menü-Tool und damit das dritte Tool für den PC, was zu schreiben sich lohnt.

5 Zum eigentlichen Programmieren der Betty

Wer - wie gesagt - per JTAG klarkommt, braucht hier nicht weiterzulesen.

Alle anderen müssen den im Controller eingebauten Bootlader benutzen. Das ist allerdings eine eher strubbelige Sache: Im Gegensatz zu den anderen Chips seiner Familie hat der LPC2220 keinen internen Programmspeicher, sondern zwei externe Flashroms. Der auch im LPC2220 eingebaute Bootlader besitzt jedoch keinerlei Code, um externen Flashrom zu behandeln. Deshalb greifen auch die üblichen Tools wie FlashMagic hier nicht.

Das Einzige, was der eingebaute Bootlader für uns tun kann ist, ein Stückchen Programm (was wir ihm geben müssen) in den internen RAM zu laden und dort draufzuspringen.

Glücklicherweise hat ein freundlicher Programmierer mit dem Programm "Betty-Heaven" eine Programmiersoftware geschrieben, die genau das so tun kann.

Um in den eingebauten Bootlader zu kommen, muß man ein bestimmtes Pin des Controllers auf low halten und ihm dann einen Reset verpassen. Wenn er anläuft und das Pin auf low sieht, geht es in den Bootlader.

Normalerweise wird dieses Verzweigen in den Bootlader mittels zweier Steuersignale der V24-Schnittstelle erledigt und auch Betty-Heaven hat dies eingebaut.

Aber dummerweise ist Betty-Heaven dabei ein Schnitzer passiert, denn das Resetsignal erscheint nach kurzer Zeit wieder und die Kommunikation kommt nicht zustande.

Wir können uns jedoch sehr einfach behelfen: Wir brauchen einen kleinen Schiebeschalter und einen kleinen Tipptaster. Unseren seriellen Adapter müssen wir ja sowieso noch basteln. Die Signale am Steckverbinder der Betty sind alles TTL-Signale, also keine V24-Signale! Am besten man besorgt sich per Ebay o.ä. einen USB zu seriell Adapter, der TTL-Signale ausgibt. Einfach mal suchen nach sowas wie

"PL2303HX USB to TTL Serial pport Converter Module 5V & 3.3V Output"

Sowas kostet ca. 3 Euro. Von diesem Konverter verbinden wir Ground, TxD und RxD mit dem Steckverbinder zur Betty und sonst nix. Den Schiebeschalter verbinden wir mit Ground und BOOT. Den Taster verbinden wir mit Ground und Reset. Sinnvoll ist es, sich ein Stückchen Streifenleiterplatte zu nehmen und beides dort draufzulöten. Anschließend die Kabelenden mit der Leiterplatte per Tesa oder Schrumpfschlauch verbinden, weil einem sonst (wie mir) öfters mal ein Kabel abbricht. Wer will, kann an den Schiebeschalter noch "Betty" und "Bootlader" schreiben.

Damit hätten wir unser Programmier-Equipment. Zum Programmieren stellen wir den Schiebeschalter auf "Bootlader" und betätigen kurz den Resettaster. Anschließend starten wir Betty-Heaven und können jetzt unsere Flashroms programmieren. Es dauert ein wenig lang, denn Betty-Heaven ist kein Rennpferd. Aber es geht und das ist entscheidend.

6 ARM oder THUMB das ist hier keine Frage

Die ARM7TDMI Prozessoren verstehen 2 Befehlssätze: den 32 bittigen ARM Befehlssatz und den 16 bittigen THUMB Befehlssatz.

Für einige Programmteile ist ARM zwingend erforderlich. Dazu gehören Startupcode und Interruptprogramme. Daran sollte man sich strikt halten. Natürlich kann man mutwillig Interruptprogramme auch in THUMB Code übersetzen lassen. Damit das dann funktioniert, muß der Linker sogenannten Interworking-Code dazwischensetzen. Besser ist es deshalb, für Interrupt-Programme ARM zu nehmen und sie so kurz wie möglich zu halten.

Die Betty hat ihren Code in zwei externen Flashroms, die mit 16 Bit Datenbreite angekoppelt sind. Für einen ARM Befehl sind also 2 Codezugriffe zu je 16 Bit nötig. Im THUMB Mode braucht ea hingegen nur einen Codezugriff.

Deshalb ist es gut, für alles andere durchgängig nur THUMB Code zu erzeugen. Für sowas gibt es eingebaute Pragmas:

#pragma arm

und

#pragma thumb

Alles was nach solchen Pragmas kommt, wird in den angesagten Code compiliert.

Interruptprogramme, also die Funktionen, die einen Interrupt bedienen, müssen speziell gekennzeichnet werden, weil ihr Start- und Ende-Code anders ist als der von gewöhnlichen Funktionen. Dazu dient das Wörtchen ***__irq*** Ein Beispiel:

```
#pragma arm
void __irq Uart0TxService (void)
{ volatile long IIR;
  int si, so;
  si = v24sin;
  so = v24sout;
  VICSoftIntClr = (1<<6)  /* Soft Int löschen */
  IIR = U0IIR;            /* Ident Reg lesen  */
  if ((si!=so) && (U0LSR & (1<<5))
  {  U0THR = V24puffer[v24sout];
     v24sout = (so+1) & (v24bufsize-1);
  }
  VICVectAddr = 0;        /* clear Interrupt  */
}
#pragma thumb
/* ab hier weiter im Thumb Modus */
```

Diese drei etwas über den gewöhnlichen ANSI-C Gebrauch hinaus reichenden Ausdrücke sind schon fast alles, was es in den Quellen an Besonderheiten gibt.

Eine Besonderheit wäre noch zu nennen: Software-Interrupts. Klingt gewaltig, ist aber ganz einfach. Neuerdings wird das Ganze "SVC" (Supervisor-Call) genannt, ist aber dasselbe. Das API - also das Interface zwischen BettyBase und einer App besteht aus solchen Software-Interrupts bzw. Supervisor-Calls. Für solche Calls gibt es im Maschinencode des Prozessors einen speziellen Befehl, der vom Compiler anstelle des sonst dort stehenden Unterprogramm-Aufrufes eingebaut wird. In diesem speziellen Maschinenbefehl gibt es (im THUMB-Mode) ein Feld, wo eine Nummer 0..255 eingebaut werden kann. Anhand dieser Nummer kann dann das zuständige Programm innerhalb der BettyBase feststellen, welchen Systemdienst man beanspruchen möchte.

Ein Beispiel: so steht es in der Headerdatei:

```
__swi(19) int CgStr_at (int x, int y,
                        char* P, word FontMode);
```

und man benutzt es im Programm so wie eine normale Funktion:

```
CgStr_at (20, 10,
     "Hallo Betty", black | idf_Helv_14B);
```

Damit schreibt man auf den Bildschirm "Hallo Betty" mit dem Font Helvetica 14 Bold und mit schwarzer Farbe.. Easy, wat?

Daraus macht der Compiler dann

```
MOVS    R3, #0x43        ; Param 4 = Fontmode
MOVS    R1, #10          ; Param 2 = Y Koordinate
MOVS    R0, #2           ; Param 1 = X Koordinate
ADR     R2, aHallo       ; param 3 = "Hallo"
SVC     19               ; Supervisor Call
```

Damit das so funktioniert, muß der Compiler natürlich sowas wie SWI oder SVC kennen und beim Compilieren den speziellen Maschinenbefehl zu erzeugen.

7 Nochmal zu den Interrupts und States

Die ARM7TDMI Prozessoren kennen mehrere Systemzustände. Fast alle Systemzustände haben ihren eigenen Stack und ihre eigenen Flagregister. Die verschiedenen Stacks werden im Startupcode aufgesetzt. Einige

Zustände haben ihre eigenen Register, so vor allem der FIQ-Zustand. Und sie haben - wie gesagt - fast alle ihre eigenen Stacks! Daher kommt es beispielsweise auch, daß man bei Funktionen, die man per SWI (Software-Interrupt) aufruft, nur maximal 4 Argumente verwenden darf und daß die Arguments sich mit 4 Registern der CPU darstellen lassen müssen. Beim SWI schaltet der Prozessor nämlich auf einen anderen Stack um und alle Argumente, die auf dem Userstack liegen, sind für das aufgerufene Programm nicht zugänglich.

Der UNDEF State: Wenn man auf irgendeinen Bereich im 4 GB Adreßvolumen des Prozessors zugreift, der nicht durch irgendetwas ***definiert*** ist, also weder zum RAM, noch zum ROM noch zu irgendwelchen Registern gehört, dann wird man vom Prozessor abgewürgt. Das heißt, es gibt einen Interrupt, der Stack wird auf den UndefStack umgeschaltet und was weiter passiert, hängt vom Undef-Service-Programm ab, das daraufhin gestartet wird.

Man kann sich das mal praktisch ansehen, indem man sich eben mal per eingebautem Hintergrundmonitor den Bereich 0..FF anzeigen läßt:

```
> D 0 FF
```

Bingo, nach den "geremappten" Bytes kommt das Aus.

Kurzum, es ist gut für's grundlegende Verständnis, sich die Doku von ARM zu den diversen Systemzuständen gut durchzulesen.

Das Normale ist der **Usermode**. Hier sind alle Interrupts prinzipiell freigegeben und die F- und I-Bits nicht mehr veränderbar. Das ist gut und richtig und im Grundzustand läuft die BettyBase in diesem Usermode. Gibt es jedoch einen Interrupt, dann schaltet der Prozessor auf den entsprechenden Mode um. Mit der Rückkehr aus dem Interrupt geht es wieder zurück in den Usermode.

Eine Besonderheit gibt es: den FIQ-Mode: Wenn ein FIQ (**F**ast **I**nterrupt Re**Q**uest) passiert, dann schaltet die CPU in den FIQ-Mode, der nicht nur seinen eigenen winzigen Stack hat (normalerweise nur 4 Byte für die geretteten Flags), sondern auch seine eigenen Register von R8 bis R14. Gedacht ist der FIQ für Interrupts, die schnell und häufig kommen und deshalb so sparsam wie möglich behandelt werden müssen. Wenn man nur R8..R14 benutzt, dann braucht man kein einziges Register auf den Stack zu retten. Daher der nur 4 Byte große Stack. Deshalb ist beim FIQ pure Assemblerprogrammierung angesagt. Erbsenzählen mit jedem Maschinenbefehl ist hier eine Tugend.. Ein typisches Beispiel ist eine Quasi-DMA mithilfe des FIQ.

Bei der Programmierung der ARM-Prozessoren ist es eher unüblich und auch verpönt, für irgendwelche schlecht durchdachten Algorithmen mal kurz die Interrupts zu disablen, wie das viele Umsteiger von kleineren Controllern gern tun. Wesentlich besser ist es, sich vernünftige Algorithmen auszudenken und die notwendigen Interrupts geeignet zu priorisieren.

Ein kleines Negativ-Beispiel sei hier genannt:

```
int some_value;

bool done;

void __int I_did_it (void)
{ if (done==false)
  { theOutput = some_value;
    done = true;
  }
}
function doof (void)
{ int i;
  i = 99;
  while(i)
  { disableint();
    some_value = my_Result(i--);
    done = false;
    enableint();
    while(done==false);
  }
}
```

Hier hacken 2 voneinander unabhängige Funktionen auf derselben Variablen 'done' herum. Grundfalsch! Ich habe so einen Mist schon so oft in anderer Leute Quellen gesehen! Oft dünkt man sich besonders schlau, wenn man die kritische Sequenz in DisableInt und EnableInt einfaßt.

Dabei geht es doch viel einfacher: Man gebe jeder Funktion eine Variable, auf die sie das exclusive ***Schreib***recht hat und fertig. Lesen dürfen beide, schreiben jeder nur auf seines:

```
int some_value;
int ich, du;

void __int I_did_it (void)
{ if (ich!=du)
  { theOutput = some_value;
    ich = du;
  }
}

function schlau (void)
{ int i;
  i = 99;
  du = ich;       /* wenn gleich, passiert nix */
  while(i)
  { some_value = my_Result(i--);
    ++du;           /* jetzt wird's ungleich */
    while(ich!=du); /* bis Irq nachgezogen hat */
  }
}
```

Hier kommen sich die beiden Funktionen nicht in die Quere. Mit solchen Konstrukten spart man sich all die Verrenkungen, die die Konstrukteure von ARM den Dummköpfen in den Weg gelegt haben. Wer es dennoch tun will, der muß eine Menge bedenken: um an die entscheidenden Bits heranzukommen, muß man sich bereits in einem privilegierten Zustand befinden. Obendrein muß man beachten, daß man es ja je nach Zustand mit unterschiedlichen Stacks zu tun hat und daß der aufrufende Code ja in ARM oder THUMB vorliegen kann und daß man sich sogenannte 'spurious' Interrupts einhandeln kann, die fachgerecht abzuhandeln sind. Kurzum: besser Hände weg, bis man die recht umfänglichen Dokus von ARM im Detail verstanden hat und genau weiß, was man tut - bevor man es tut.

Ein typisches Beispiel für Puffer zwischen zwei miteinander kooperierenden Funktionen sind I/O-Puffer für UART's:

```
char V24puffer[v24bufsize];  /* Sendepuffer */
volatile int v24sin;   /* Index im Sendepuffer */
volatile int v24sout;  /* Index im Sendepuffer */
```

Auch hier hat jeder **seinen** Zeiger: Das Programm, was den Puffer füllt, hat **v24sin** und das Interrupt-Programm, was ihn wieder leert, hat **v24sout**. Beide Programme können ihre Sache machen ohne sich gegenseitig mit irgendwelchen DisableInt's verriegeln zu müssen. Es ist etwa so wie bei 2 Leuten, die sich gegenseitig auf die Finger schauen. Jeder kann seine Finger und die des Anderen betrachten und seine Schlüsse daraus ziehen, aber jeder kann nur das verändern, was er in seinen eigenen Fingern hat. Da braucht es keinen gemeinsamen Topf, wo mehrere zugleich hineingreifen, um etwas zu verändern - und wo sie sich tendenziell immer in die Quere kommen.

8 Was es so alles in der BettyBase gibt

Die BettyBase hat einen Hintergrundmonitor, der über den seriellen Kanal UART0, also über unseren Programmieradapter zu erreichen ist. Die Baudrate kann man nach eigenem Gusto wählen, derzeit habe ich sie auf gemütliche 9600 Bd gestellt. Der Monitor ist in ***cmd.c*** zu finden und nach eigenem Belieben mit Kommandos anreicherbar. Wie das geht, sollte aus der Quelle ersichtlich sein. Hier ein Beispiel:

```
if (match("D",&Cpt))      { DoDisplay(); return;}
if (match("BLABLA",&Cpt)){ DoBlablabla();
                           return;
                         }
```

Für diverse Ein- und Ausgaben gibt es ein reichhaltiges Sortiment von Funktionen in ***conv.c*** wie z.B. Hexa, Dezimal und Float Input und Output. Braucht man z.B. nach dem Kommandowort noch einen Integerparameter, dann schreibt man

```
if (match("BLABLA",&Cpt))
{ DoBlablabla(Long_In(&Cpt));
  return;
}
```

Auch ganz easy, nicht wahr? Wie man sieht, ist eigentlich der ganze Programmteil cmd.c genauso wie conv.c fast völlig hardwareunabhängig - wenn man mal davon absieht, daß es ja gerade die Kommandos sind, die auf der Hardware etwas bewirken sollen und deshalb mit der konkreten Hardware was tun sollen.

Die UART0 Bedienung findet sich in serial.c

Dem Setup ist eine separate Quelle ***setup.c*** gewidmet. Ich habe dort eine Menge Preprozessor-Akrobatik geschrieben, damit man es bei Änderungen leicht hat und nicht aus Versehen was anderes einreißt. Es finden sich ja noch viele Teile der Betty, die ich in BettyBase noch gar nicht berücksichtigt und mit einem Treiber versehen habe.

Die Abfrage der Tasten auf der Betty findet sich in ***tasten.c*** Man kann sowohl entprellte Tastendrücke (mit dort einstellbarer Repetierzeit) abfragen als auch den momentanen Zustand aller Tasten abfragen. Für die Repetierzeit gibt es 2 Zeiten: Die Zeit bis zum 1. Repetieren hab ich auf 0.9 Sekunden gestellt und die Zeit nach dem ersten Repetieren auf 0.1 Sekunden.

Die komplette LCD-Bedienung findet sich in ***gdi.c*** Allerdings sind die 2 Lanes des Videopuffers im Startupcode festgelegt. Das hat den Vorteil, daß man auch außerhalb von BettyBase mit diesen festen Bereichen arbeiten kann.

Der Timer 0 ist als Systemtick eingerichtet und die damit verbundenen Aktionen werden per Interrupt veranlaßt. Das Einzige, was derzeit dort erledigt wird, ist die Entprellung der Tasten, das Sekundensignal für das Menü und derzeit noch ein 200 ms Signal für das Anzeigen der ADC's.

9 Menü und Events

Man kann ein Menüsystem auf 1000 verschiedene Arten konstruieren. Das, was ich hier implementiert habe, ist ein ganz generisches Menü, mit dem man alles machen kann: Listen, Icons, Gemischt, es ist alles drin. Auch Eigabezeilen, Schalter usw. sind machbar. Ich habe vorsorglich eine Variable

```
bool isEditing;
```

vorgesehen, die man benutzen kann, wenn man sich eine Editierzeile o.ä. ausdenkt und dabei zwischen Editieren und Navigieren unterscheiden muß.

Am Anfang stehen Events und ein zugehöriger Puffer. Verschiedene Programmteile können Ereignisse generieren, so z.B. der Systemtick:

```
void __irq Timer0Service (void)
{ T0IR  = 0xFF;      /* IRQ im Timer löschen */
  T0MR0 = T0TC + 10; /* Matchreg 10 ms weiter */
  Tasten_Bonze();
  ++Ticks;
  if ((Ticks==20)||
      (Ticks==40)||
      (Ticks==60)||
      (Ticks==80))  Add_Event(id_200_milli);
  if (Ticks >= 100)
  { Ticks = 0;
    Add_Event(idSekundeUm);
  }
  VICVectAddr = 0;  /* clear Interrupt */
}
```

Hier wird alle Sekunde ein Ereignis kreiert, das dem Menü anzeigt, daß mal wieder eine Sekunde um ist. Ähnlich macht es die Grundschleife in main.c

```
if (IsKeyAvailable())
 { i = TranslateKeynum(GetKey());
   if (i) Add_Event(tastMin + i);
 }
```

Hier wird ein Tastenereignis erzeugt und in den Eventpuffer gebracht.

Davon völlig losgelöst ist die Verarbeitung der Events. Sie werden nacheinander dem Menü quasi in den Rachen geworfen und das Menü kann sich dann aussuchen, was es damit macht:

```
if (Event_avail()) DispatchEvent(Get_Event());
```

Der DispatchEvent ist praktisch der einzige Input-Kanal zu den Gefilden des Menüs. Dieses besteht im Wesentlichen aus einem hierarchisch verknüpften Satz von Objekten.

Objekte .. in C ? Hmm, ja. Man muß sie zwar zu Fuß deklarieren, aber dafür funktioniert es. Bitte zu diesem Zweck menu.h anschauen und vestehen lernen. Das ist wichtig!

In C sind diese Objekte gewöhnliche struct's, die aber jeweils bis zu 3 Funktionen enthalten: eine für Tastaturereignisse, eine für andere Ereignisse und eine, die das Objekt zeichnet. Normalerweise nennt man dies **Methoden**.

Die simplen kleinen Menüs in der BettyBase hab ich per Script gemacht, entsprechend formal sieht es von innen aus. Es wäre wesentlich besser, sich dafür einen grafischen Menü-Editor zu schreiben.

Soweit Menü-Objekte nur formale Aufgaben erledigen, sind ihre Methoden bereits in ***menu.c*** enthalten.

Aber darüberhinaus will man ja mit dem Menü etwas bewirken. Die dazu benötigten Methoden finden sich in ***BettyWorkMenu.c***.

Was man treiben muß, um dort prozedural zu programmieren und trotzdem sich nicht gänzlich selbst zu blokkieren sieht man in "Betty_Tastentest_OnKey".

Im Detail sieht ein Menüelement so aus:

```
#define RCST  const struct TMenuItem
struct TMenuItem
{ struct TRect R;
  const struct TMenuItem *davor;
  const struct TMenuItem *danach;
  const struct TMenuItem *Owner;
  const struct TMenuItem *Members;
  dword Flags;
  void  *Data;
  void  (*OnKey)  (RCST* self, word *aKey);
  void  (*OnEvent)(RCST* self, word *aEvent);
  void  (*Draw)   (RCST* self);
};
```

Das erste Element ist natürlich ein Rechteck, was die Lage bestimmt. Aber das Rechteck hat keine Bildschirmkoordinaten, sondern seine Koordinaten beziehen sich auf den Clientbereich des **Owners**.

Die nächsten zwei Elemente sind zwei Zeiger (davor und danach), die zur Verlinkung des Menüelementes in einer doppelt verlinkten Kette dienen. Ob man wirklich doppelt verlinken muß, oder ob es einfach auch ausreicht, kann mal diskutiert werden.

Die nächsten zwei Elemente sind der Owner und die Members. Jedes Menüelement (außer dem Menü selbst) hat einen Owner, in dessen Clientbereich es sich befindet und von dem es seine Nachrichten erhält.

Das Menü selbst erhält seine Nachrichten von DispatchEvent.

Jedes Menüelement kann seinerseits beliebig viele Members enthalten. Hat es welche, dann muß es sich natürlich seinerseits darum kümmern, daß auch seine Members Nachrichten erhalten.

Als Nächstes hat es das Element Flags. Hier sind alle möglichen Flagbits angesiedelt. So z.B. ***opaque*** (das Element soll nicht durchsichtig sein) oder ***canFocus*** (das Element kann beim Navigieren im Menü den Fokus erhalten). Das ist was Wichtiges, denn nicht alles, was man darstellt, ist ein Knopf zum drauf drücken.

Das nächste Element ist ein **`void* Data`** und auch dies ist was Wichtiges. Die struct's sind ja im ROM angesiedelt und manche Elemente benötigen zusätzliche Daten im RAM oder ebenfalls im ROM. Zu diesen Daten soll der Zeiger hinführen.

Den Schluß bilden die drei Methoden des Menüelementes. Die Draw-Methode braucht wohl jedes Element, damit es sich darstellen kann. Ob das in Form eines Textes, eines nett gestalteten Buttons, eines Icons oder sonstwas ist, ist allein Obliegenheit des Menüelementes. Damit hat man alle Freiheiten für die eigene Menügestaltung.

Die Ereignisse, von denen das Menü über DispatchEvent informiert wird, sind in 2 Gruppen aufgeteilt: *Events* und *Keys*. Dieser Unterschied ist wichtig, denn *Events* können alle Menüelemente angehen und werden deshalb an alle verteilt. *Key-Ereignisse* hingegen müssen anders behandelt werden. Sie müssen zu allererst an das gerade fokussierte Menüelement gereicht werden, denn das ist es ja gerade, was zu allererst darauf reagieren soll. Erst wenn es nix damit anfängt (und das Ereignis löscht), soll sich sein Owner darum kümmern.

So im Prinzip funktioniert das Menüsystem.

10 Fonts

Fonts haben natürlich sehr eng zu tun mit dem Rasterizer im GDI, aber sie sind dennoch eine extra Kategorie. Ich habe hier einmal einige Fonts unterschiedlicher Größe und unterschiedlichen Zeichenvorrates eingebaut. Grundsätzlich haben sie variable Zeichenbreiten. Im Prinzip ist es so, daß jedes Zeichen aus einem Bitstream besteht, ohne dabei auf Byte- oder Word-Grenzen auszurichten. Ein Zeichen 5x7 Pixel braucht da eben nur 35 Bit und fertig.

Ein Font besteht zu allererst aus einem Fontheader, der die grundlegenden Eigenschaften beschreibt:

```
struct Fontheader
{ byte Capitals; /* Hoehe der Grossbuchstaben */
  byte Height;   /* Font-Gesamthoehe */
  byte Ascent;   /* Hoehe ueber Null-Linie */
  byte Descent;  /* Tiefe unter Null-Linie */
  char ChMin; /* kleinstes vorhandenes Zeichen */
  char ChMax  /* groesstes vorhandenes Zeichen */
};
```

Danach schließt sich ein Block von Zeichenheadern an:

```
struct CharEntry
{ byte DispHi; /* Hi Displacement ab Fontbeginn */
  byte DispLo; /* Lo Displacement ab Fontbeginn */
  byte dX;     /* Breite in Pixeln */
  byte dY;     /* Hoehe in Pixeln */
};
```

Für jedes Zeichen von ChMin bis ChMax gibt es hier ein solches struct. Anschließend kommen die Pixel.

`(DispHi<<8)|DispLo` ergibt den Offset der Pixel des jeweiligen Zeichens ab Beginn des Fontheaders. Die Pixel muß man sich als einen Bitstream vorstellen. Ein Zeichen mit 22 Pixel Höhe und 16 Pixel Breite hat 22*16 = (ähh, wo ist mein HP32 abgeblieben?) 352 Pixel, untergebracht in 44 Byte. Einfach und effizient.

Die Eigenschaften Capitals, Ascent und Descent kennt vielleicht nicht jeder. Height gibt die Höhe des Fonts insgesamt an. Irgendwo innerhalb der Höhe ist eine gedachte Schreiblinie, unterhalb derer sich die Unterlängen

(Descents) befinden, denn Kleinbuchstaben reichen ja gelegentlich unter diese Schreiblinie, wie z.B. j,g,y und so weiter. Die Großbuchstaben gehen für gewöhnlich nicht bis hinauf zur vollen Fonthöhe, sondern bleiben etwas darunter. Capitals ist ihre Höhe über der Schreiblinie . Ascent ist nun alles, was von der Fonthöhe nach Abzug von Descent noch übrig bleibt.

11 Apps für BettyBase

Eine Anwendung für BettyBase zu schreiben, ist eigentlich ganz einfach: man braucht nur den mitgelieferten Startupcode ***Appstartup.asm*** zu verwenden und ***StdTypes.h*** sowie ***BettyApp.h*** einzubinden und man kann schon loslegen.

Hier der Code des Startups:

```
  AREA    Startup, CODE, READONLY
  ENTRY
  PRESERVE8     ; 8-Byte aligned Stack
  ARM

  IMPORT  main
  EXPORT  AppStart
  EXPORT  AppIcon
  EXPORT  AppName
  EXPORT  AppDescription

Signatur DCB     "BETTYAPP"
Flags    DCD     0
Alloc    DCD     0x82000000
Start    DCD     AppStart
Icon     DCD     AppIcon
Name     DCD     AppName
Descr    DCD     AppDescriptio

       EXPORT _main
       EXPORT __main
_main
__main
AppStart
        LDR     R7, =main
        BX      R7

AppIcon DCW     0

AppName DCB     "App-Vorlage",0

AppDescription
        DCB     "diese Vorlage macht nichts!",0

        ALIGN
        LTORG
        END
```

Dies ist sozusagen die Steilvorlage. Für eine echte App sollte man eine Kopie dieses Startupcodes verwenden, die man folgendermaßen editiert:

- bei Alloc trägt man die gerade verwendete absolute Anfangsadresse dieser App ein, wo die App auch wirklich drauf gelinkt ist. Der zweite Flashrom beginnt ja mit 0x82000000 und der Appfinder sucht nach Apps auf allen Anfängen der 32 K Blöcke im Flashrom.
- wenn man (künftig) ein Icon für diese App hat, dann baut man es bei AppIcon ein
- für AppName gibt man an, wie diese App tatsächlich heißen soll
- bei AppDescription gibt man eine Erläuterung nach eigenem Gusto.

Der Appfinder testet im Flashrom an den 32 K Grenzen, ob sich dort eine Signatur befindet und ob der Alloc-Wert mit der tatsächlichen Adresse übereinstimmt. Ist beides in Ordnung, dann nimmt er die App zur Kenntnis. Man kann sie dann per OK-Taste starten.

Im Appfinder sieht das so aus:

```
#define PAPP struct TAppHeader
struct TAppHeader
{ dword  Signatur1;
  dword  Signatur2;
  dword  Flags;
  PAPP*  Alloc;
  long  (*Start) (void);
  dword  Icon;
  char*  Name;
  char*  Descr;
};

PAPP* Appliste[appmaxnum];

... nach Apps suchen:
    if ((P->Signatur1 == AppSignatur1) &&
        (P->Signatur2 == AppSignatur2))
    { if (P == P->Alloc)
      { Appliste[i] = P;
        ...
```

und zum Aufruf:

```
   retval = Appliste[FocussedApp]->Start();
```

So eine App kann auch wie ein gewöhnliches PC-Programm sich beenden, indem int main (void) mit einem Returnwert zurückkehrt. Wer es brutal will, kann auch mit einem SWI(8) über das API einen Warmstart der Betty erzwingen.

Zum RAM-Bedarf: Die Apps benutzen den gleichen UserStack wie die BettyBase selbst.

BettyBase selbst ist mit

```
--rw-base 0x4000C000
```

gelinkt, belegt also dan RAM ab 0x4000C000. Dort sind die Variablen von BettyBase, die Stacks und der Displaypuffer angesiedelt. Für Apps stehen deshalb 48 K RAM ab 0x40000000 zur Verfügung. Um es nochmal genau zu sagen: Die Apps kommen in den Flash und sie haben als RAM die genannten 48 K zur Verfügung. Wenn nun jemand sich eine App ausdenkt, die mit anderen Apps mitläuft, dann muß er deren RAM-Bedarf so planen, daß sie sich nicht mit anderen Apps im RAM 'beißt'. Einen Anfang habe ich beim App-Startup-Code vorgesehen: irgendwann kann man bei den dortigen Flags ein Bit "Autostart" oder so ähnlich vorsehen und in die BettyBase einen Scanner, der beim Systemstart alle Apps startet, die so ein Bit gesetzt haben. Ist momentan aber noch nicht implementiert. Ähnlich sieht es mit den freigehaltenen 8 SWI's aus. Die kann man später mal als ladbare Systemcalls ausbilden, um von einer App aus spezielle SWI's kreieren zu können. Ich denke da an spezielle Treiber für Infrarot, Radio, Audio und so weiter.

In BettyBase ist derzeit kein Applader eingebaut. Um Apps in den Flashrom zu bekommen, muß man also sein Programmierequipment bemühen. Wenn man mehrere Apps im Rom haben will, sollte man sich ein Tool schreiben, das verschiedene Apps zu einem Rom-Image zusammenführt.

12 RTOS und andere Spezereien

Ob es wirklich einen Sinn ergibt, auf der Betty einen Realtime-Scheduler unterzubringen, vermag ich derzeit nicht zu sagen. Das Einzige, was ich bislang gemacht habe ist, die ersten 8 SWI's für so etwas freizuhalten, denn in den Dokus von Keil steht, daß deren Realtime-Kernel selbige belegt.

Den FIQ habe ich im Startup-Code zwar vorgesehen, aber nur mit einem Handler, der nix tut. Wer den FIQ also benutzen will, muß dort editieren.

13 ein paar wirre Ideen zum Schluß

All das, was bisher beschrieben wurde, kann man machen, ohne die Betty auseinander zunehmen oder irgendwelche Veränderungen vorzunehmen. Aber zum Basteln gehört ja auch, zumindest ein paar Pins des Controllers mit eigener Hardware zu verbinden. Geeignete Pins gibt es im Prinzip genug, man muß bloß gut löten können. Was mir dabei in den Sinn kommt:

- die Betty als kontaktloser Drehzahlmesser. Einen richtigen Fototransistor und einen aus einem Laserpointer ausgebauten Laser vorn einbauen und damit die Drehzahl von irgendwelchen drehenden Teilen (Modellbau?) messen
- die Betty als Beleuchtungsmesser: Eine richtige Fotodiode und einen logarithmierenden Verstärker vorn dranbauen und einen der freien ADC-Kanäle nutzen
- die Betty als Lärmpegelmesser: ein kleines Elektretmikro und einen Verstärker vorn anbauen und wiederum einen freien ADC-Kanal benutzen
- die Betty als HF-Monitor

... Ach Leute, Ideen gibt es sicherlich viele

So, genug für heute, es bleibt weiter interessant.

Frohes Basteln wünscht

W.S.