Ein Wort zuvor

ST baut zwar in alle seine STM32 Controller einen residenten Bootlader ein, doch um dessen Unterstützung durch eine passende PC-Software ist es schlecht bestellt. Deren „STMFlashLoaderDemo“ ***ist*** lausig. Punkt. Deshalb habe ich mir mal selbst so etwas zurecht geschrieben, was mittlerweile benutzbar ist. Besonders schön und elegant ist es nicht und diverse Einzelheiten gehen aus prinzipiellen Gründen nicht. Dazu zählt vor allem eine anfangs von mir ins Auge gefaßte automatische Chip-Bestimmung. Ich hatte die Map-Dateien des STMFlashLoaders durchgesehen, um zu einer Chipauswahl für mein Programm zu kommen und mußte dabei feststellen, daß offenbar die gleiche PID an verschiedene Chips vergeben wurde. Es möge also ein jeder Interessent die Datei „targets.cfi“ nach seinem Gusto bearbeiten, um seinen Lieblings-Chip einzutragen und nicht im Gewusel von ST den Überblick zu verlieren.

Was ist das Ganze überhaupt?

Das vorliegende Programm dient zum Programmieren von STM32Fxxx Controllern unter Verwendung von deren fest eingebautem Bootlader über eine serielle Schnittstelle. Das Programm läuft unter Windows. Ich habe es ursprünglich mit Delphi 6 geschrieben und bin dann auf Delphi 10 umgestiegen. Der Umstieg war nötig, weil das Delphi6-Programm unter Windows 7 zwar funktioniert wie gewollt, aber sein Aussehen ist grausam, wenn Fonts und Oberfläche von Windows 7 anders als genau auf 100% skaliert sind. Falls jemand ***ernsthaft*** das Programm nach Lazarus portieren will, kann er von mir die Quellen kriegen.

Probleme mit STM32Prog

Beim Ausprobieren mit einem sicherheits-fanatisch vernagelten Windows 7 ist mir noch etwas aufgefallen: sobald man einen Scan der COM-Ports vornimmt, kommt irgend ein dralles Sicherheits-Tool des Weges, was meint man sei ein schlimmes Schadprogramm und löscht die EXE kommentarlos weg.

Wegen solcher Albernheit habe ich das COM-Port-Scannen ab Programmstart ausgeschaltet. Man kann es im Port-Setup einschalten und danach die Portliste aktualisieren. Dann werden die Ports wieder gescannt und die Portliste enthält danach nur noch die Ports, die momentan im System verfügbar sind.

Wenn man von vornherein weiß, auf welchem Port der Programmieradapter erreichbar ist, dann kann man ihn auch direkt eingeben. Das Programm merkt sich dies in der aktuellen Konfigurationsdatei.

Was braucht man und wie funktioniert es?

Zunächst braucht man eine serielle Schnittstelle, die mit TTL-Pegeln bzw. 3.3 Volt Pegeln auf der seriellen Seite arbeitet. Im Zweifelsfalle nachmessen! Wir können hier keine echten V.24 Pegel gebrauchen.

Ich benutze einen simplen USB-Adapter, auf dem ein FT232BL von FTDI verbaut ist. Dessen TxD und RxD Signale gehen beim STM32 an dessen Default-Pins für USART1_RXD bzw. _TXD (immer TxD von einer Seite mit RxD der anderen Seite verbinden, wer Angst hat, schleift in die Leitungen einen 470 Ohm Widerstand ein).

Dazu werden aber auch DTR und RTS des FTDI-Chips zum Reset des STM32 und zum Auswählen des Bootlader-Starts benutzt.

Natürlich kann man auch andere Chips nehmen, solange man dort an DTR und RTS herankommt. Viele billige China-Adapter mit einem Prolific-Chip haben diese Signale leider ***nicht*** herausgeführt.

Mit einem dieser Signale wird der Reset-Eingang des STM32 verbunden und mit dem anderen wird der Umschalteingang (BOOT_0) verbunden – welcher wohin führt, kann man im Programm einstellen.

Manche Chips haben dazu noch einen weiteren Boot-Pin (BOOT_1). In diesen Fällen muß man im zugehörigen Referenzmanual nachlesen, welche Kombination aus BOOT_0 und BOOT_1 zu welchem Startverhalten führt. Zum Programmieren muß der Controller nach dem Reset in den Bootlader gehen und nicht zum Flash-Speicher.

Wichtig: Man kann am STM32 die Boot-Pins auch per Jumper o.ä. festlegen, aber den Zugang zum Reset-Pin braucht man in jedem Falle, weil der chipinterne Bootlader bei einigen Löschvorgängen den Chip per Software-Reset zurücksetzt. Das klappt nach meiner Erfahrung nur manchmal richtig und deshalb ist es einfach sauberer und zuverlässiger, vom PC-Programm aus den echten Hardware-Reset zu betätigen.

Noch wichtiger: Man sollte seinem seriellen Adapter **wirklich** den Zugriff auf zumindest den Reset-Pin des Comtrollers gönnen – man erspart sich damit eine Menge Frust und Ärger, denn damit kann das Programm den Chip jederzeit von sich aus in einen bekannten Zustand versetzen. Immerhin ist zu bedenken, daß für den Zugriff auf COM-Ports (egal ob virtuell oder real) das Betriebssystem immer zunächst den Port mit den Default-Werten öffnet. Anschließend kann/muß man sowas wie Parität und Baudrate per DCB und später per Escape-Funktionen ändern. Dabei kann (*kann* – muß aber nicht) schon mal ein Glitch am Port auftreten und im blödesten Falle erkennt der Bootlader dies als sein Synchronisier-

Signal. Die Baudrate des Bootladers ist nach unten hin begrenzt auf ca. 2400 Bd, aber nach obenhin ist m.W. keine Grenze außer der schieren Hardware vorhanden. Das kann 100x gut gehen, muß aber nicht wirklich.

Weswegen ich hier darauf herumreite? Nun, es haben Leute versucht, ohne Reset und Boot den Chip nur per TxD und RxD zu programmieren – das geht manchmal, manchmal aber eben auch nicht. Deshalb der gut gemeinte Rat, zumindest Reset anzuschließen.

Natürlich muß man sein Board mit dem Chip drauf zum Programmieren mit Strom versorgen – das sei nur der Vollständigkeit halber erwähnt.

Zum Programmieren braucht man eine Datei im Intel Hex Format (*.hex) oder im Motorola S19 Format (*.S19) oder als simples Binärformat (*.bin). ELF-Dateien (*.elf, *.axf usw.) gehen ***nicht***. Diese muß man zuvor mittels FROMELF o.ä. konvertieren.

Diese Datei muß die Firmware enthalten, und zwar auf die tatsächlichen Ausführungsadressen gelinkt. Klingt kompliziert, ist es aber gar nicht. Normale Cortexe greifen ab Reset auf Adressen ab 0 (null) zu. Also muß dort wenigstens die Vektortabelle stehen und der Code schließt sich dann normalerweise dort hinten an. Aber der programmierbare Flash steht bei allen mir bekannten STM32 ab Adresse 0800'0000h an. Deswegen portiert mein Programm allen Code im Hexfile auf den Bereich ab Flash-Anfang. Später beim normalen Programmstart wird dieser Bereich dann auf Adresse 0 gespiegelt und die gewöhnliche Abarbeitung kann dann ab dort stattfinden.

Obendrein habe ich für Leute, die partout nix außer RxD+TxD verwenden wollen, eine Art Überflieger-Modus eingebaut. Dabei gibt sich das Programm auch mit einem NAK auf das Synchronisieren zufrieden – in der Hoffnung, daß ein sauber erkanntes NAK ($1F) anstelle $79 ausreichend ist, um anzunehmen, daß der Bootlader nun arbeitet. Das klappte bei mir mit dem erwähnten FTDI-Chip und einem STM32F302RBT6 durchaus zuverlässig, aber mir ist *unwohl* dabei.

Zur Installation

Ganz einfach: man kreiert sich ein Verzeichnis eigener Wahl – aber außerhalb der üblichen Default-Verzeichnisse. Dort hinein kopiert man die EXE und wenigstens die Datei „targets.cfi“. Die Datei „default.cfp“ mag man auch dort hineinkopieren, tut man es nicht, dann legt das Programm sich selbst eine solche an und füllt sie mit Defaultwerten.

Solange man ohne Kommandozeilenparameter arbeitet, erzeugt das Programm ***NICHTS*** außerhalb seines Installationsverzeichnisses. Es legt also keinerlei Regi-

stry-Werte an und benutzt auch keine anderen Verzeichnisse. Der Preis dafür besteht darin, daß es sich seine Einstellungen in der Datei „default.cfp“ merkt – und die liegt in seinem eigenen Installationsverzeichnis. Wer das Programm also unter „Programme…“ oder „Programme (X86)…“ installieren will, wird anschließend vom System genau deswegen belästigt. Deshalb der Rat, es außerhalb der Default-Verzeichnisse zu installieren.

..es geht auch anders:

Nämlich indem man die Datei „default.cfp“ einmal manuell editiert und dort unter „INI“ den vollständigen Pfad zu einer anderen Datei (EigenerPfad\EigenerName.cfp) angibt.

Dann lädt STM32Prog eben diese Datei als seine Default-Datei und führt keine Schreiboperationen in der Datei in seinem eigenen Verzeichnis durch.

Schlampereien von ST

Hier geht es vor allem erst einmal um das Kommando 0x44, also das Extended Erase Memory Kommando. Laut AN3155, Kapitel 3.8 sollte ein Bootlader, der das Kommando 0x44 zu verstehen vorgibt, selbiges auch beherrschen. Aber dem ist nicht so, Zitat aus AN2606:

„Some products don’t support Mass erase operation. To perform a mass erase operation using bootloader, two options are available:

– Erase all sectors one by one using the Erase command

– Set protection level to Level 1. Then, set it to Level 0 (using the Read protect command and then the Read Unprotect command). This operation results in a mass erase of the internal Flash memory.“

Der Knackpunkt ist, daß der verbaute Bootlader das Kommando annimmt, quittiert und ausführt – aber der Chip hängt sich hardwaremäßig dabei auf (Latchup oder sowas) und kommt erst nach dem Wegnehmen von Vcc wieder zur Besinnung, Reset reicht ***nicht*** aus! Es ist also ***kein*** Software-Bug.

Eigentlich sollte der Bootlader laut Kapitel 3.12 beim Ausführen des Kommandos 0x92 (Readout Unprotect Command) ***immer*** den gesamten Flash ablöschen – auch ohne daß man zuvor ein Kommando 0x82 (Readout Protect Command) gegeben hat. Aber auch dies gilt offenbar nicht wirklich.

Kurzum: Einige essentielle Dinge sind in manchen Chips dezent verbuggt und die zugehörige Dokumentation von ST ist in diversen Details uneindeutig oder falsch. Also auf zum fröhlichen Ausprobieren.

Die Programmbeschreibung

Zunächst der Aufruf: normalerweise ohne Parameter. Wird ein Parameter angegeben, dann erwartet das Programm entweder eine Konfigurationsdatei (*.cfp) oder eine Hexdatei (*.hex oder *.s19) oder eine Binärdatei (*.bin) – jeweils mit vollem Pfad.

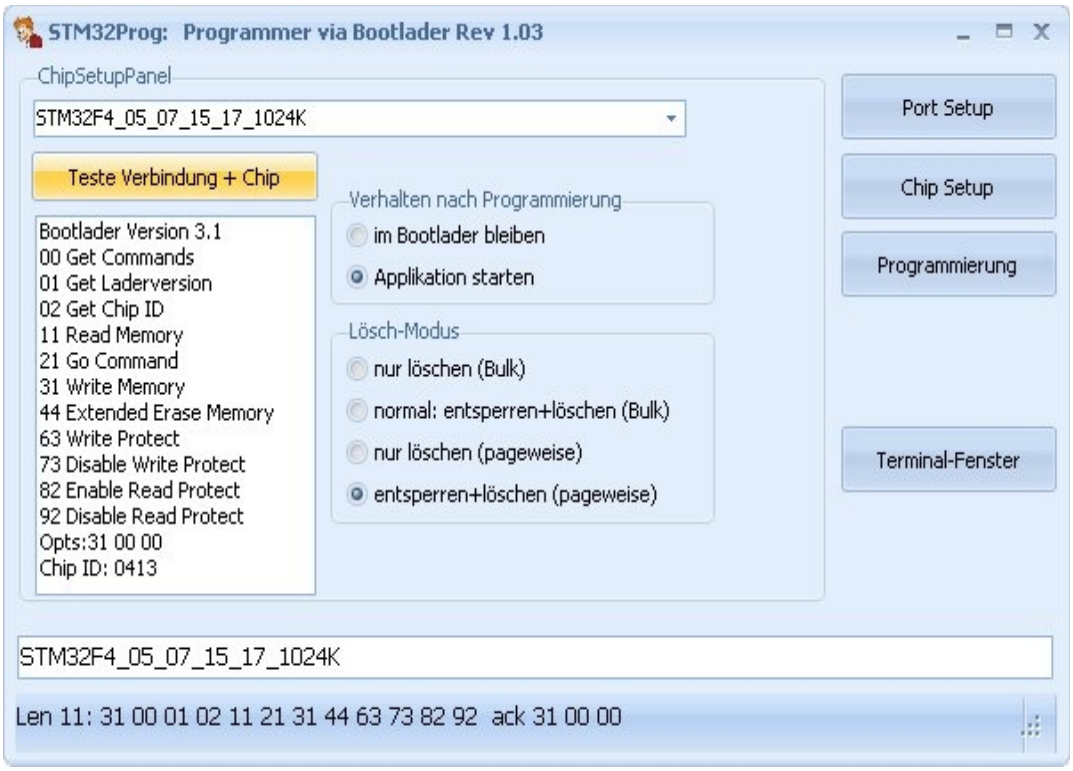
Hier das Programm, wenn man den Port Setup Knopf gedrückt hat:

Hier sucht man sich den passenden COM-Port aus, unter dem man seinen Programmieradapter erreicht. Beim Programmstart füllt das Programm die Portliste mit COM1 bis COM99. Aktiviert man „COM's scannen“ und klickt dann auf „aktualisieren“, dann trägt das Programm nur die Ports in die Liste ein, die es im System momentan auch findet. Steckt man also seinen Adapter erst nach dem Programmstart ein, kann man die Liste mit dem darunter liegenden Knopf aktualisieren lassen. *Hinweis:* Das ändert aber nichts an der Eingabezeile der Dropdown-Box. Also Box ausklappen und Port auswählen.

Bei den Baudraten werden nur die üblichen Raten angezeigt. Man kann jedoch von Hand eine andere Baudrate eintragen, die anschließend vom Programm zum Programmieren benutzt wird.

Die Signale für den Chip-Reset und für den Boot-Modus können rechts davon ausgewählt werden. Die darunterliegenden Knöpfe dienen zum Testen mittels Multimeter auf der Leiterplatte, wo sich der zu programmierende Chip befindet.

Hier sieht man das Programm, wenn der Chip Setup Knopf und danach der „Teste Verbindung...“ Knopf gedrückt wurde:



Oben ist die Auswahl-Liste für den gewünschten STM32-Typ. Diese enthält derzeit die Einträge, die ich von der „STMFlashLoaderDemo“ aus deren Setup-Dateien extrahiert habe. Wenn der gewünschte Typ nicht dabei ist, dann muß man per Texteditor die Datei „targets.cfi“ bearbeiten und seinen STM32 dort eintragen. Zum Inhalt dieser Datei siehe Anhang. Mit dem „Teste Verbindung...“ Knopf kann man testen, ob die Verbindung vom PC zum STM32 funktioniert, ob der Bootlader richtig startet und was er für Kommandos verträgt. Wenn man im Log-Fenster das sieht, was in obigem Bild zu sehen ist, dann ist die Verbindung OK.

Mit den Radiobuttons rechts neben dem Log-Fenster kann man festlegen, ob nach dem Programmieren oder Verifizieren das Programm einen Reset macht und anschließend den Chip vom Flash booten läßt oder nicht – das gilt aber nicht für Rücksetzvorgänge, die vom Bootlader ausgehen. Wenn man das Terminalfenster dieses Programms benutzen will, um mit der soeben einprogrammierten Firmware zu kommunizieren, dann sollte man die untere Option (Applikation starten) auswählen.

Mit den Radiobuttons im Feld „Lösch-Modus“ kann man jetzt auswählen, ob das Programm wie bisher den Bulk-Erase zum Löschen benutzt oder bei Chips, wo das nicht geht (siehe ST-Schlampereien weiter oben) die Sektoren einzeln löscht. Letzteres dauert ***lange*** im Vergleich zum Bulk-Erase. Beim Ausprobieren ist mir

aufgefallen, daß das Löschen einzelner Pages/Sektoren unterschiedlich lange dauert: bei dem alten Discovery, was ich dazu benutzt habe, löschten die ersten 4 Sektoren binnen weniger Zehntelsekunden, aber die letzten Sektoren benötigten jeweils fast 4 Sekunden zum Löschen.

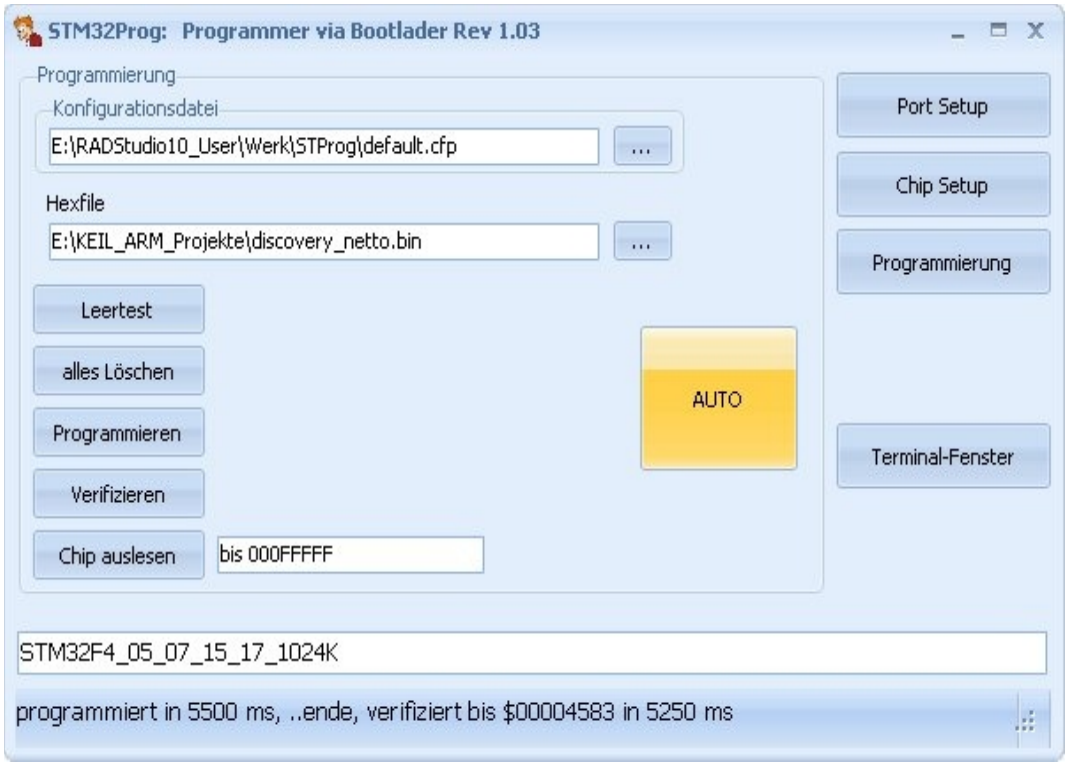
Wozu all die Setup-Menüs?

Ich habe all diese Testmöglichkeiten im Port-Setup und im Chip-Setup mit Bedacht vorgesehen, damit man sein Programmiergeschirre möglichst problemlos in Gang bekommt - das grottenschlechte Beispiel des „STMFlashLoaders“ vor Augen. Der kaut bloß stumm auf einem etwaigen Problem herum – und mit der finalen Meldung „es geht nicht“ kann man fast gar nichts anfangen.

Also habe ich es anders gemacht: Man kann Port und Baudrate nach eigenem Gusto festlegen und man kann mit Oszillograf oder Multimeter testen, ob die Zuordnung der Reset- und Boot-Signale so erfolgt, wie man es sich gedacht hat. Obendrein kann man mit dem Terminalfenster testen, ob die Kommnikation als solche geht – auch ohne einen Controller angeschlossen zu haben. Dazu verbindet man lediglich TxD und RxD am Adapterausgang und man sollte im Terminalfenster alle Zeichen sehen, die man auf der Tastatur eintippt.

Das eigentliche Programmieren

Kommen wir nun zur Haupt-Ansicht des Programms:



Hier sieht man, welche Konfigurationsdatei aktiv ist, welches Hexfile zu programmieren ist und welcher konkrete Chip ausgewählt ist. Für dieses Bild hatte ich ein gewöhnliches älteres STM-Discovery (ohne Display) ausgelesen, das Binärfile auf den Netto-Inhalt gekürzt, den Chip gelöscht (und zur Kontrolle nochmal ausgelesen) und dann per 'AUTO' neu programmiert. Es geht also.

Ein Wort zur Codegröße: Ich habe Pufferspeicher für maximal $200000 Bytes, also 2 MB vorgesehen. Ich denke, mit diesen maximal 2 MB Code wird man hier erst einmal auskommen.

Eine durchlaufende Automatik wählt man mit dem großen Knopf „AUTO“ an. Dabei wird zunächst das Hexfile frisch eingelesen, dann der Bootlader nach seinen Kommandos befragt (insbesondere nach Cmd $43 oder $44), sodann der Chip ggf. entriegelt und gelöscht (Bulk oder einzeln), dann programmiert und zum Schluß verifiziert.

Man kann jedoch auch alles in Einzelschritten tun: löschen, programmieren, verifizieren.

Eine Sonderstellung nimmt das Chip-Auslesen ein. Es wird zum Programmieren nicht benötigt und es liest per default den ***gesamten*** Flash des Chips aus und speichert dessen Inhalt in einer Binärdatei. In dem Eingabefeld rechts neben dem Auslese-Knopf kann man allerdings angeben, bis wohin der Chip auszulesen ist.

Vorsicht: Das Auslesen und der Leertest können recht lange dauern, da alle Daten in nur 256 Byte kleinen Stückchen ausgelesen werden können.

Ich lasse bislang die Arbeits-Funktionen des Programmes noch im Haupt-Thread laufen und habe sie nicht in einen separaten Thread ausgelagert. Das bedeutet, daß das Programm während der Arbeit (hier: beim Auslesen) keine weiteren Kommandos annimmt.

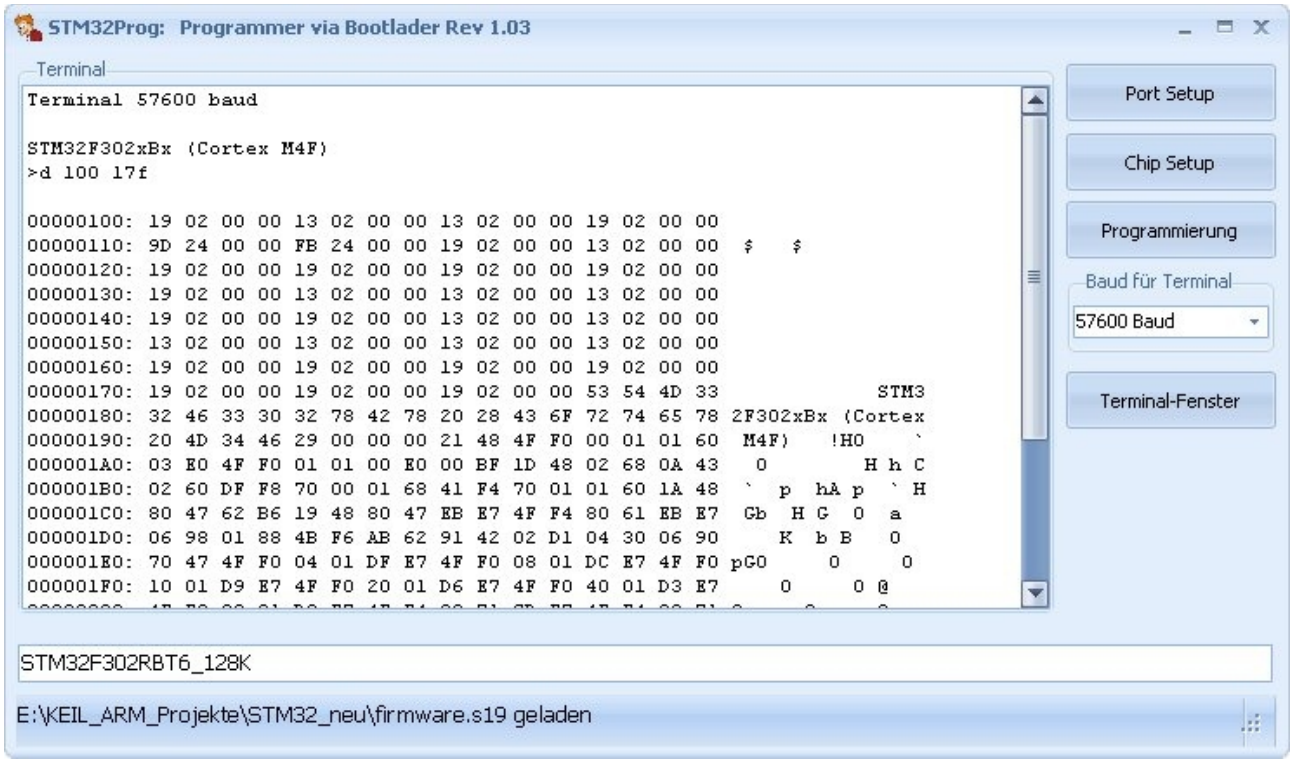
Man muß also warten, bis es fertig ist.

Auch den Versuch, Chips mittels Read-Protect+Read-Unprotect zu löschen, habe ich noch nicht unternommen – in diesem Punkt traue ich ST nicht über den Weg.

Kurzum: Es haben sich bereits einige Ideen angesammelt, die auf Realisierung warten. Ich werde damit aber nur dann anfangen, wenn genügend Resonanz von Anderen vorhanden ist, denn für meine Bedürfnisse reicht das Programm allemal.

Zum Terminalfenster

Hier sieht man das Terminalfenster in Aktion. Es ist derzeit nur ein ganz simples Terminal ohne Sonderzeichen, Escapesequenzen, Emulationen usw. Aber für eine einfache Kommunikation mit dem eigenen Programm im Controller reicht es erst einmal aus. Es kommuniziert 8N2 und die Baudrate kann man getrennt vom Bootlader separat festlegen.



Seltsamerweise haben sich bei mir die USART's im STM32F302 mit dem FTDI-Chip nicht vertragen, wenn ich lediglich 8N1 kommunizieren lasse. Aber da dies eher eine Nebensache hier ist und es für die Kommandogabe egal ist, ob nach einem Tastendruck am PC nun 1 oder 2 Stopbits gesendet werden, verschiebe ich das Nachschauen auf später.

Noch ein abschließendes Wort zu den verwendbaren Baudraten: Ich habe den Bootlader bis zu 160 kBaud ausprobiert, der Betrieb damit ging problemlos.

Allerdings habe ich keinen Geschwindigkeitsvorteil gegenüber 115200 Baud gesehen, weswegen ich erst einmal alles über 115200 für unnütz halte.

Viel Spaß beim Basteln wünscht

W.S.

Anhang 1: Der gegenwärtige Inhalt der Datei „targets.cfi“

Jede Zeile enthält einen Chip nach dem Muster:

```
Chipname; PID; Flash-Anfang; Sektorgröße; Flash_Ende; Optionadresse; Page-Anzahl; 'P' oder 'S'
```

wobei alle Adressen in HEX geschrieben ist und alle Spalten durch Semikolon getrennt sind. Die Page-Anzahl ist dezimal und 'P' oder 'S' sollen anzeigen, ob es sich lt. 'FlashLoaderDemo' um Pages oder Sektoren handelt.

Die Sektorgröße und Optionadresse verwende ich derzeit nicht. Es gibt also bislang keine Auslese-Sperre usw.

```
; Name                                PID   Flash     Sektor    Flash end  Options Pages/Sectors Typ
STM32F0_3x_16K;                     0444;  08000000; 00000400; 08003FFF; 1FFFF800;    16; P
STM32F0_3x_32K;                     0444;  08000000; 00000400; 08007FFF; 1FFFF800;    32; P
STM32F0_4x_16K;                     0445;  08000000; 00000400; 08003FFF; 1FFFF800;    16; P
STM32F0_4x_32K;                     0445;  08000000; 00000400; 08007FFF; 1FFFF800;    32; P
STM32F0_5x_3x_16K;                  0440;  08000000; 00000400; 08003FFF; 1FFFF800;    16; P
STM32F0_5x_3x_32K;                  0440;  08000000; 00000400; 08007FFF; 1FFFF800;    32; P
STM32F0_5x_3x_64K;                  0440;  08000000; 00000400; 0800FFFF; 1FFFF800;    64; P
STM32F0_7x_64K;                     0448;  08000000; 00000800; 0800FFFF; 1FFFF800;    32; P
STM32F0_7x_128K;                    0448;  08000000; 00000800; 0801FFFF; 1FFFF800;    64; P
STM32F0_9x_256K;                    0442;  08000000; 00000800; 0803FFFF; 1FFFF800;   128; P
STM32_Connectivity-line_64K;        0418;  08000000; 00000800; 0800FFFF;       - ;    32; P
STM32_Connectivity-line_128K;       0418;  08000000; 00000800; 0801FFFF;       - ;    64; P
STM32_Connectivity-line_256K;       0418;  08000000; 00000800; 0803FFFF;       - ;   128; P
STM32_High-density_256K;            0414;  08000000; 00000800; 0803FFFF;       - ;   128; P
STM32_High-density_384K;            0414;  08000000; 00000800; 0805FFFF;       - ;   192; P
STM32_High-density_512K;            0414;  08000000; 00000800; 0807FFFF;       - ;   256; P
STM32_High-density-value_256K;      0428;  08000000; 00000800; 0803FFFF;       - ;   128; P
STM32_High-density-value_384K;      0428;  08000000; 00000800; 0805FFFF;       - ;   192; P
STM32_High-density-value_512K;      0428;  08000000; 00000800; 0807FFFF;       - ;   256; P
STM32_Low-density_16K;              0412;  08000000; 00000400; 08003FFF;       - ;    16; P
STM32_Low-density_32K;              0412;  08000000; 00000400; 08007FFF;       - ;    32; P
STM32_Low-density-value_32K;        0420;  08000000; 00000400; 08003FFF;       - ;    16; P
STM32_Low-density-value_32K;        0420;  08000000; 00000400; 08007FFF;       - ;    32; P
STM32_Med-density_64K.STmap;        0410;  08000000; 00000400; 0800FFFF;       - ;    64; P
STM32_Med-density_128K;             0410;  08000000; 00000400; 0801FFFF;       - ;   128; P
STM32_Med-density-value_64K;        0420;  08000000; 00000400; 0800FFFF;       - ;    64; P
STM32_Med-density-value_128K;       0420;  08000000; 00000400; 0801FFFF;       - ;   128; P
STM32_XL-density_768K;              0430;  08000000; 00000400; 080BFFFF;       - ;   768; P
STM32_XL-density_1024K;             0430;  08000000; 00000800; 080FFFFF;       - ;   512; P
STM32F2_128K;                       0411;  08000000; 00010000; 0801FFFF;       - ;     5; S
STM32F2_256K;                       0411;  08000000; 00020000; 0803FFFF;       - ;     6; S
STM32F2_512K;                       0411;  08000000; 00020000; 0807FFFF;       - ;     8; S
STM32F2_768K;                       0411;  08000000; 00020000; 080BFFFF;       - ;    10; S
STM32F2_1024K;                      0411;  08000000; 00020000; 080FFFFF;       - ;    12; S
STM32F3_02_01_64K;                  0439;  08000000; 00000800; 0800FFFF;       - ;    32; P
STM32F302RBT6_128K;                 0422;  08000000; 00000800; 0801FFFF; 1FFFF800;    64; P
STM32F3_03_02_256K;                 0422;  08000000; 00000800; 0803FFFF;       - ;   128; P
STM32F3_03_02_512K;                 0446;  08000000; 00000800; 0807FFFF;       - ;   256; P
STM32F3_7x_8x_256K;                 0432;  08000000; 00000800; 0803FFFF; 1FFFF800;   128; P
STM32F3_34_03_64K;                  0438;  08000000; 00000800; 0800FFFF;       - ;    32; P
STM32F4_01_256K;                    0423;  08000000; 00020000; 0803FFFF; 1FFFC000;     6; S
STM32F4_01_512K;                    0433;  08000000; 00020000; 0807FFFF; 1FFFC000;     8; S
STM32F4_05_07_15_17_1024K;          0413;  08000000; 00020000; 080FFFFF; 1FFFC000;    12; S
STM32F4_10_128K;                    0458;  08000000; 00010000; 0801FFFF; 1FFFC000;     5; S
```

```
STM32F4_11_512K;                0431;  08000000; 00020000; 0807FFFF; 1FFFC000;     8; S
STM32F4_12_1024K;               0441;  08000000; 00020000; 080FFFFF; 1FFFC000;    12; S
STM32F4_27_37_29_39_2048K;      0419;  08000000; 00020000; 081FFFFF;       - ;    24; S
STM32F4_46_512K;                0421;  08000000; 00020000; 0807FFFF; 1FFFC000;     8; S
STM32F4_69_79_2048K;            0434;  08000000; 00020000; 081FFFFF;       - ;    24; S
STM32F7_4x_5x_1024K;            0449;  08000000; 00040000; 080FFFFF; 1FFF0000;     8; S
STM32L0_x3_x2_x1_64K;           0417;  08000000; 00000080; 0800FFFF; 1FF80000;   512; P
STM32L0_x3_x2_x1_192K;          0447;  08000000; 00000080; 0802FFFF; 1FF80000;  1536; P
STM32L1_Cat1-32K;               0416;  08000000; 00000100; 08007FFF;       - ;   128; P
STM32L1_Cat1-64K;               0416;  08000000; 00000100; 0800FFFF;       - ;   256; P
STM32L1_Cat1-128K;              0416;  08000000; 00000100; 0801FFFF;       - ;   512; P
STM32L1_Cat2-32K;               0429;  08000000; 00000100; 08007FFF;       - ;   128; P
STM32L1_Cat2-64K;               0429;  08000000; 00000100; 0800FFFF;       - ;   256; P
STM32L1_Cat2-128K;              0429;  08000000; 00000100; 0801FFFF;       - ;   512; P
STM32L1_Cat3-256K;              0427;  08000000; 00000100; 0803FFFF;       - ;  1024; P
STM32L1_Cat4-256K;              0436;  08000000; 00000100; 0803FFFF;       - ;  1024; P
STM32L1_Cat4-384K;              0436;  08000000; 00000100; 0805FFFF;       - ;  1536; P
STM32L1_Cat5-512K;              0437;  08000000; 00000100; 0807FFFF;       - ;  2048; P
STM32L4x_6_1024K;               0415;  08000000; 00000800; 080FFFFF;       - ;   512; P
```

Diese Liste ist ersichtlichermaßen ziemlich unübersichtlich, aber das ist man von ST ja gewöhnt: Statt einer konkreten Chip-Bezeichnung muß man sich erstmal übelegen, ob der aktuelle Chip nun der Connectivity- oder Low-density- oder Med-density- oder XL-density oder weiß der Geier was für einer Reihe angehört.

Mein Rat: Ein jeder möge sich diese Liste zusammenkürzen und nur die Chips hineinschreiben, die er selbst benutzt.

Anhang 2: der prinzipielle Inhalt der Datei „default.cfp“

```
BAUD    115200
COM     5
RESET   1
BOOT    4
APSTART 1
UNLOCK  3
CHIPNAM STM32F302RBT6_128K
FLASH   134217728
FLENG   131071
PAGES   64
PAGETYP P
INI     e:\Delphi10_User\STM32Prog\default.cfp
HEX     E:\KEIL_ARM_Projekte\STM32_neu\firmware.hex
LABER   57600
```

Das sollte eigentlich selbsterklärend sein. Findet STM32Prog in seinem eigenen Verzeichnis keine „default.cfp“, dann legt es sich eine solche Datei an und füllt sie mit Default-Werten. Jaja, das bedeutet *Schreiben ins Programmverzeichnis*.

Da ich grundsätzlich C: **nicht** für eigene Anwendungen aller Art benutze, sondern auf einer zweite Platte/Partition anordne (und Daten auf einer dritten), stört mich auch nicht die Bevormundung in den Default-Programm-Verzeichnissen.