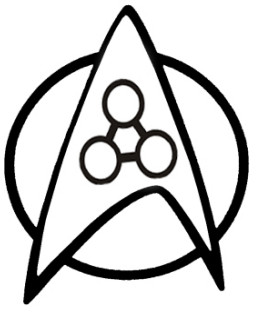
Der Weltraum. Unendliche Weiten. Wir schreiben das Jahr 2013.



So komme ich mir manchmal bei meinen Experimenten mit dem Hive vor: Allein TriOS als einfaches Mini-OS ist eine kleine Welt für sich und mental erscheint mir wie ein Paralleluniversum…

Nun ja, diese aktuelle Version von mental bezeichne ich mal als Alphaversion. Startbar unter TriOS oder auch völlig eigenständig nutzbar, indem der Chipcode im Flash gespeichert wird. Eine Minidokumentation "Der erste Außeneinsatz" ist ebenfalls verfügbar. Ohne Grafik und Sound kann man zwar noch nicht so viel spannende Sachen damit machen, aber das steht als nächstes auf dem Plan - immer schön einen Schritt nach dem anderen.

Im folgenden Text werde ich die Entstehung und inneren Mechanismen von mental ein wenig genauer beleuchten. Interessant für Neugierige, aber mit Sicherheit auch für Einsteiger in die Programmierung der Cogs in Assembler. Wer nicht so der Typ "Scotty" ist und mental nur nutzen möchte, sollte besser nicht den Maschinenraum betreten und gleich mit dem Text "Der erste Außeneinsatz" fortfahren! Alle Boardingenieure sind aber herzlich eingeladen zu einem Rundgang...

Alles begann im Jahr 2008: Nachdem der Ur-Hive mit seinen ersten drei Herzschlägen ins Leben getreten war, hatte ich schon damals mit iSpin (Artikel: Wie alles anfing) ein wenig über eine forthähnliche Programmiersprache nachgedacht. Damals demotivierte mich letztlich die resultierende Geschwindigkeit, aber als ein in Spin geschriebener "Forthemulator" war es der erste Schritt, um die Ideen hinter der Sache zu verstehen.

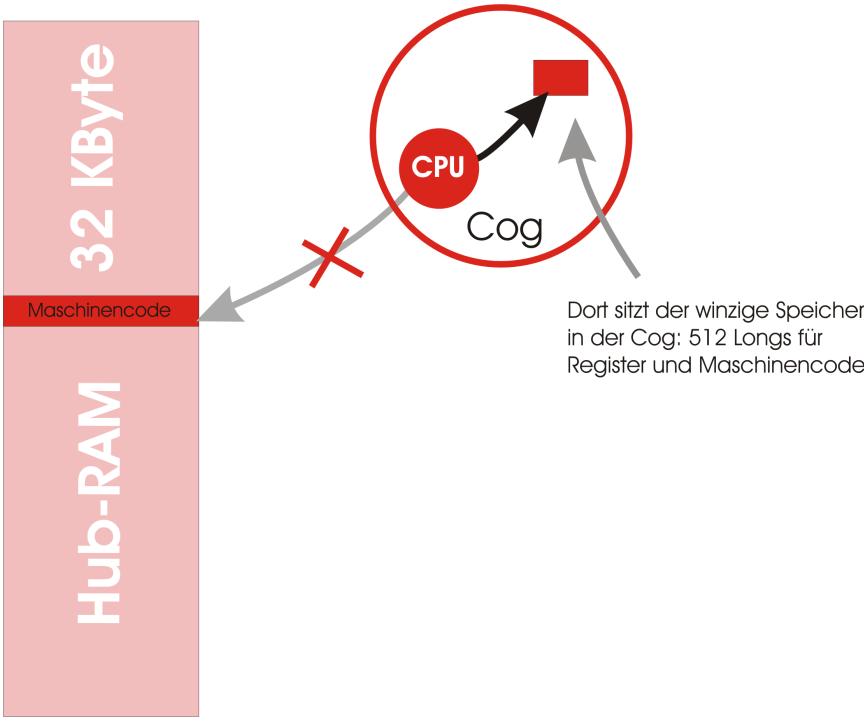
Mittlerweile bin ich einen kleinen Schritt weiter und habe mich mit der Subraumtheorie… ähm, mit PASM (Propeller-Assembler) angefreundet und mit dem mCore einen Warpkern in die Idee eines Forth für den Hive eingepflanzt. Aber fangen wir in der Geschichte etwas früher an...

Das Problem am Propellerchip sind seine zwei Gesichter, wahrscheinlich hätte dem Chip auch gut auf den Namen Janus getauft werden können.

Auf der einen Seite kann man den Prop in Spin programmieren, was sehr komfortabel für so einen Mikrocontroller ist. Spin mit seinem einfachen objektorientierten Ansatz ist wirklich sehr verführerisch. Es gibt unheimlich viele fertige Objekte für alle erdenklichen Anwendungen: Schnittstellen, Grafikausgabe, Konverter, Soundobjekte usw. Die Nachteile: Man ist bei der Erstellung, bedingt durch die Komplexität des Compilers, immer an einen Hostcomputer gebunden und die Geschwindigkeit von Spin ist etwa um den Faktor 60..100 geringer als Maschinencode auf dem blanken Metall. Zudem ist Spin primär dafür gedacht, um mit einem einzelnen Propellerchip zu arbeiten.

Und da sind wir auch schon beim zweiten Gesicht des Chips: Maschinencode. Hier ist nun wieder so eine massive Leistung in jeder Cog vorhanden, dass einem fast schwindlig wird. Aber wie immer gibt es die süßesten Früchte nicht umsonst: Maschinencode kann von den RISC-CPU's in den Cogs nur im privaten Cog-RAM (cRAM) ausgeführt werden - nicht im allgemein für alle Cogs zugänglichen Hub-RAM (hRAM) und schon gar nicht im externen RAM! Und der private RAM einer Cog beziffert sich auf ganze 496 Register! (eigentlich 512 Register, aber einige Zellen sind

mit I/O und Parametern fest belegt) mit einer Breite von 32 Bit, welche als Programmspeicher und/oder als Register genutzt werden können. Nicht gerade viel, wenn man den ganzen Weltraum damit erobern will...



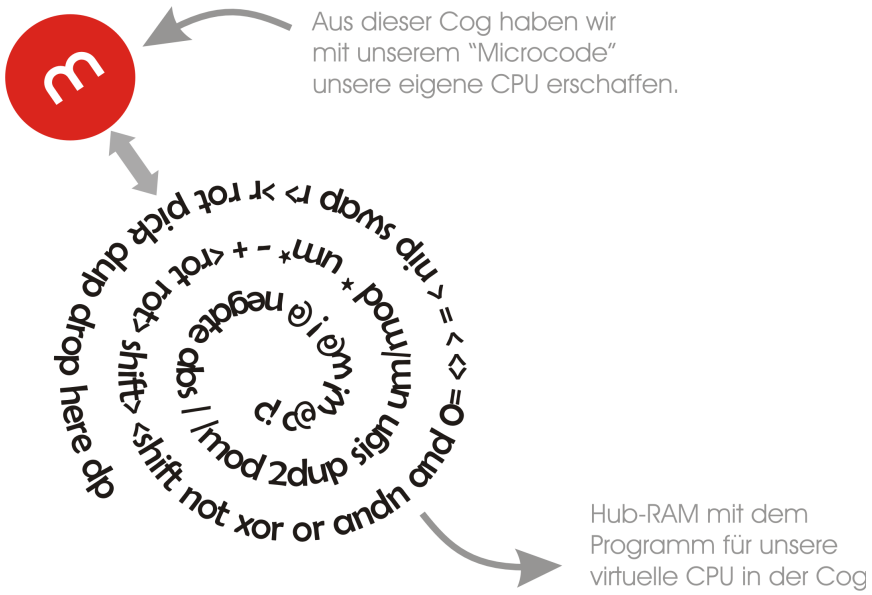
Das ist natürlich viel zu wenig Speicher, um darin ein komplettes System wie mental unterzubringen. Nun könnte man im Prinzip aber folgendes tun: Wenn man in der Cog einen Maschinenprogramm startet, welches quasi als Microcode eine Cog zu einer CPU macht, so könnte diese virtuelle CPU den viel größeren hRAM als Programmspeicher nutzen und damit aus der Enge des Cog-Subsystems mit seinen 496 nutzbaren Speicherzellen ausbrechen! Diese neu erschaffene Meta-CPU bekommt dann natürlich einen eigenen Befehlssatz, welchen wir ebenso selbst definieren können. Und in dieser selbst erdachten Maschinensprache können wir dann das mental-System programmieren... So mein Plan.

Wenn man ein wenig über diese Lösung nachdenkt, kommt schnell die Erkenntnis, dass es für dieses Prinzip für den Propellerchip schon einige praktische Realisierungen gibt:

- Spin: Spin selbst arbeite genau nach diesem Prinzip. In der Cog wird eine VM gestartet, welche den Bytecode abarbeitet, den der Spin Compiler erzeugt.
- ZiCog - ein Zilog Z80 Emulator in einer Cog
  Link: Parallax Forum - ZiCog
- MoCog - ein Motorola 6809 Emulator
  Link: Parallax Forum - MoCog
- Zog - Implementation der Open Source Zylin-CPU in einer Cog
  Link: Parallax Forum - Zog

Außer bei der Spin VM könnte man bei den drei anderen Versionen mit einem Assembler für die entsprechende virtuelle CPU Code erzeugen und im Hub RAM ausführen. Im Prinzip ginge das auch mit der Spin VM, aber durch die komplexere objektorientierte Struktur dürfte das einen ziemlich unhandlichen Assembler für Spin Bytecode ergeben...

Aber ich möchte ja letztlich nicht in Assembler programmieren, sondern in einer forthähnlichen, höheren interaktiven Sprache. Und genau das ist recht einfach zu realisieren: anstatt einen Z80 oder eine Zylin CPU in einer Cog zu realisieren, kann man ja auch gleich eine CPU gestalten, welche Forth als Maschinencode ausführt oder in seinem Maschinencode schon besser unterstützt als die Realisierungen in obigen Beispielen! Und genau das macht die neue Meta-CPU - der mCore - von mental: So besitzt er in der CPU (also in der Cog) gleich zwei getrennte Stacks für Daten und Returnadressen. Zusätzlich sind alle implementierten Befehle auf die Nutzung dieser beiden Stacks ausgerichtet - arithmetische und logische Funktionen zum Beispiel erwarten ihre Eingabeparameter auf dem Datenstack und speichern das Ergebnis auch dort. Damit entsprechen alle Maschinenbefehle des mCore auch gleichzeitig einer ganzen Palette von Forth Befehle (Worten).



Für Forth-Insider ist das natürlich nichts neues, denn sie wissen, das bei den meisten Softwareimplementierungen von Forth ein Grundwortschatz von Forthbefehlen in Maschinencode des entsprechenden Prozessors realisiert werden muß, um mit ihnen quasi eine virtuelle Forth-CPU zu erschaffen. Nichts anderes machen wir hier auch, allerdings mit einer Einschränkung: Für den Grundwortschatz an Forthbefehlen (primäre Worte) haben wir nicht mehr als unsere 496 Register zur Verfügung. Das ist Sudoku für Programmierer... ;)

Die Sonde - Entdeckung des Warpantriebs

Also los, holen wir den Assembler aus der Hosentasche und fangen an, uns eine eigene virtuelle CPU zu basteln. Aber als erstes müssen wir schauen, welche Werkzeuge uns dafür zur Verfügung stehen.

Sowohl im Spintool von Parallax, als auch im alternativen Entwicklungssystem von Brad (BST - Brad's Spin Tool), haben wir einen Assembler für die Maschinensprache der RISC-CPU's in den Cogs integriert. In beiden Fällen handelt es sich um einen sehr einfachen Assembler, da sind weder Makros noch andere hilfreiche Mechanismen integriert. Ebenso ist in dem offizielen Handbuch von

Parallax neben der Referenz der in Spin enthaltenen Befehle auch eine umfangreiche Referenz über die Programmierung des Propellerchips in Maschinencode enthalten.

Bei der Assemblerprogrammierung auf dem Hive gab es für mich am Anfang ein kleines Problem: Die Zugänglichkeit. Natürlich kann man ein Assemblerprogramm in Maschinencode mit dem Propellertool oder BST erzeugen und auch auf einem Chip im Hive zur Ausführung bringen, aber im Gegensatz zur Programmierung in Spin und mit Hilfe der Infrastruktur von TriOS kann kaum die Programmausführung beobachtet werden. Schreibt man ein einfaches Testprogramm unter Spin wie das bekannte "Hallo Welt!", so stellt das TriOS schon viele Funktionen zur Zeichenein- und Ausgabe zur Verfügung, welche man gut zum debuggen verwenden kann. Und hat man einen blanken Chip vor sich, kann immer noch über die serielle Schnittstelle kommuniziert werden, fertige Objekte liegen dem Propellertool schon bei. Möchte man aber Maschinencode nutzen, so ist das, als ob man in ein Schwarzes Loch fliegt - nachdem man den Code in die Cog "geschossen" hat, befindet er sich im übertragenen Sinne hinter dem Ereignishorizont und man kann nicht beobachten, was dort passiert: Läuft der Code richtig, ist er abgestürzt, ist er dank der rasenten Geschwindigkeit der RISC-Cores rasend schnell abgestürzt, hat er gerade in 200 ns die Lottozahlen der nächsten Woche generiert? Wir wissen es einfach nicht! Unsere Cog befindet sich sozusagen im Subraum, jenseits der normalen Dimensionen, welchen wir mit unserem Sensoren scannen können.

Dazu kommt noch erschwerend eine gewisse "Eigenartigkeit" des Maschinencodes für die RISC-CPU's hinzu. Hier mal eine kleine Auswahl:

- Es gibt keine Stacks, dennoch aber einen Call und RET. Wie gehts: Da es keinen Stack gibt, ist der Return-Befehl egentlich ein JMP-Befehl zurück, mit fester Zieladressen. Diese Zieladresse wird bei einem Unterprogrammaufruf auf die entsprechende Rücksprungandresse modifiziert... Jummy, selbstmodifizierender Code direkt in der CPU integriert, dass ist doch mal was! :) Im Prinzip handelt es sich also um einen "verteilten" Stack, der platzsparend über den gesamten Maschinencode "verschmiert" ist.
- Da selbstmodifiierender Code an der Tagesordnung ist, gibt es genau dafür spezielle Maschinencodebefehle, welche genau definierte Bereiche eines Befehls modifizieren kann.
- Jeder Maschinencodebefehl kann bedingt sein, also abhängig von den Flags. Das macht den Code verdammt kompakt.
- Für jeden Maschinencode kann bestimmt werden, ob dieser bei Ausführung die Flags setzt und ob er das Ergebnis der Operation! in die Zieladresse schreibt.
- Es gibt keine dedizierten Register, vielmehr kann jedes der 512 Speicherzellen in der Cog als Register genutzt werden, auch der Programmcode selbst. Dafür besitzt der Befehlscode immer je eine 9 Bit Ziel- und Quelladresse.

Ich bin mir nicht ganz sicher ob es Masochismus oder Wahnsinn ist, dass ich mir gerade so einen absonderlichen Prozessor für meine erste Implementation eines Forthsystems ausgesucht habe. Hmm, wahrscheinlich war es mehr Naivität und die Erkenntnis, dass ich wahrscheinlich ewig warten kann, bis mir eine kundige Person für den Hive ein Forth programmiert.

Zurück zu unserem Schwarzen Loch und seinem Ereignishorizont bzw. zu unserer Cog in ihrer Unzugänglichkeit und Absonderlichkeit: Was können wir also tun um die Situation für uns zu gewinnen? Nun, in solchen Fällen starten die Helden in ihren Sternenschiffen meist eine Sonde, welche über einen speziellen Übertragungskanal Informationen aus der unzugänglichen Region sendet. Und genau das wollen wir jetzt auch tun - wir senden eine erste Sonde in das unbekannte Terrain, einen Sputnik, der uns mit seinem Beep

Link: So war es damals - Beep Beep Beep...

ein erstes Lebenszeichen aus dem entfernten Raum sendet. Was für ein Abenteuer!

Aber gut, lasst uns zuerst unseren Maschinencode-Sputnik konstruieren:

```
'
' Spuntnik - die erste Sonde im fremden Raum
'
CON
_CLKMODE = XTAL1 + PLL16X
_XINFREQ = 5_000_000

PUB main
  cognew(@entry,0)

DAT

              ORG       0               'start im cog ram ab adresse 0

entry         mov       DIRA, PIN       'pin auf ausgabe setzen
              mov       TIME, cnt       'berechne verzögerungszeit
              add       TIME, #9        'minimalverzögerung!
:loop         waitcnt   TIME, DELAY     'warten!
              xor       OUTA, PIN       'pin umschalten
              jmp       #:loop

PIN     long  |< 24                     'pinmaske
DELAY   long  60_000_000                'verzögerungszeit

TIME    res   1
```

Wie der erste sowjetische Satellit, hat auch unsere Softwaresonde keine sehr komplizierte Funktion: Mit der Hertbeat-LED am entsprechenden Chip sendet sie uns ein Blinksignal als Lebenszeichen aus der anderen Dimension. Das ist nicht viel, aber immerhin ein Anfang und es beantwortet uns eine der wichtigsten Frage der Menschheit: Gibt es etwas jenseits unserer Welt?

Also los, starten wir die Sonde: Einfach "sputnik-0.spin" in den Compiler laden, mit F10 in eine Cog im Regnatix-Chip schießen und über eine blinkende LED staunen! :)

Aber das ist uns noch lange nicht genug, denn wir wollen ja in der Cog eine eigene virtuelle Maschine aufbauen und dazu müssen wir sie wenigstens in den Experimenten beobachten können, um zu sehen, welche Fehler wir machen. Ok, wir haben ja jetzt unsere LED-Lichtverbindung wird mancher denken, aber das wird nicht genügen. Es ist zwar nicht unmöglich, aber wahrscheinlich eine ziemlich schlechte Idee, größere Datenmengen, wie Registerbelegungen und Befehlscodes, über einen LED-Morsecode übertragen zu wollen. ;) Also brauchen wir einen leistungsfähigeren "Telemetriekanal" zu unseren Sonden. In der Propeller-Community gibt es dafür einige interessante Lösungen. Hier eine kleine Auswahl:

1. Ein allgemeiner Debugger für Spin und I/O-Werte, für PASM eher ungeeignet: Debug-Lite

von Parallax
Link: Debug-Lite
2. PASD von Andreas Schenk, eine sehr komfortable Debugger-Suite
Link: insonic - PASD
3. BMA - Bite My ASM Debugger von John Steven Denson (jazzed)
Link: Parallax Forum - BMA
4. Emulation des Maschinencodes mit dem Propelleremulator Gear
Link: sourceforge - Gear
5. Emulation mit dem Emulator pPropellerSim (Java)
Link: sourceforge - pPropellerSim

Noch weitere Debugger, Emulatoren und Tools sind im englischen Wiki aufgeführt:

Link: propeller wiki - Debuggers and Emulatoren

Mit den Emulatoren konnte ich mich nicht anfreunden, weshalb sie erstmal außen vor sind. Letztlich habe ich mich bei diesem Experiment für den BMA entschieden, bei anderen Sachen aber auch schon den sehr komfortablen PASD verwendet.

Mit dem BMA Debugger ist es möglich, eine Cog recht genau in ihrer Funktion zu beobachten. So kann man den Maschinencode im Einzelschritt abarbeiten und dabei die Veränderung der Register und Flags beobachten, man kann den Speicher in der Cog untersuchen und noch vieles mehr. Einzig drei Bedingungen müssen für die korrekte Funktion erfüllt sein:

1. Vor dem ersten Befehl müssen acht NOP-Befehle stehen. Für das Programm ohne Bedeutung, dienen sie als Platzhalter für einen Code, welchen der Debugger dort eigenständig implementiert, um das restliche Maschinenprogramm zu kontrollieren.
2. Der Maschinencode muss über das Debuggerobjekt gestartet werden. In unserem Beispiel über die Methode `deb.debug(@entry,0)`.
3. Es wird ein Terminalprogramm auf dem Host benötigt, da der Debugger über die serielle Schnittstelle nutzbar ist. Das Propellertool bringt PST mit, im BST ist ein kleines Terminalprogramm gleich integriert.

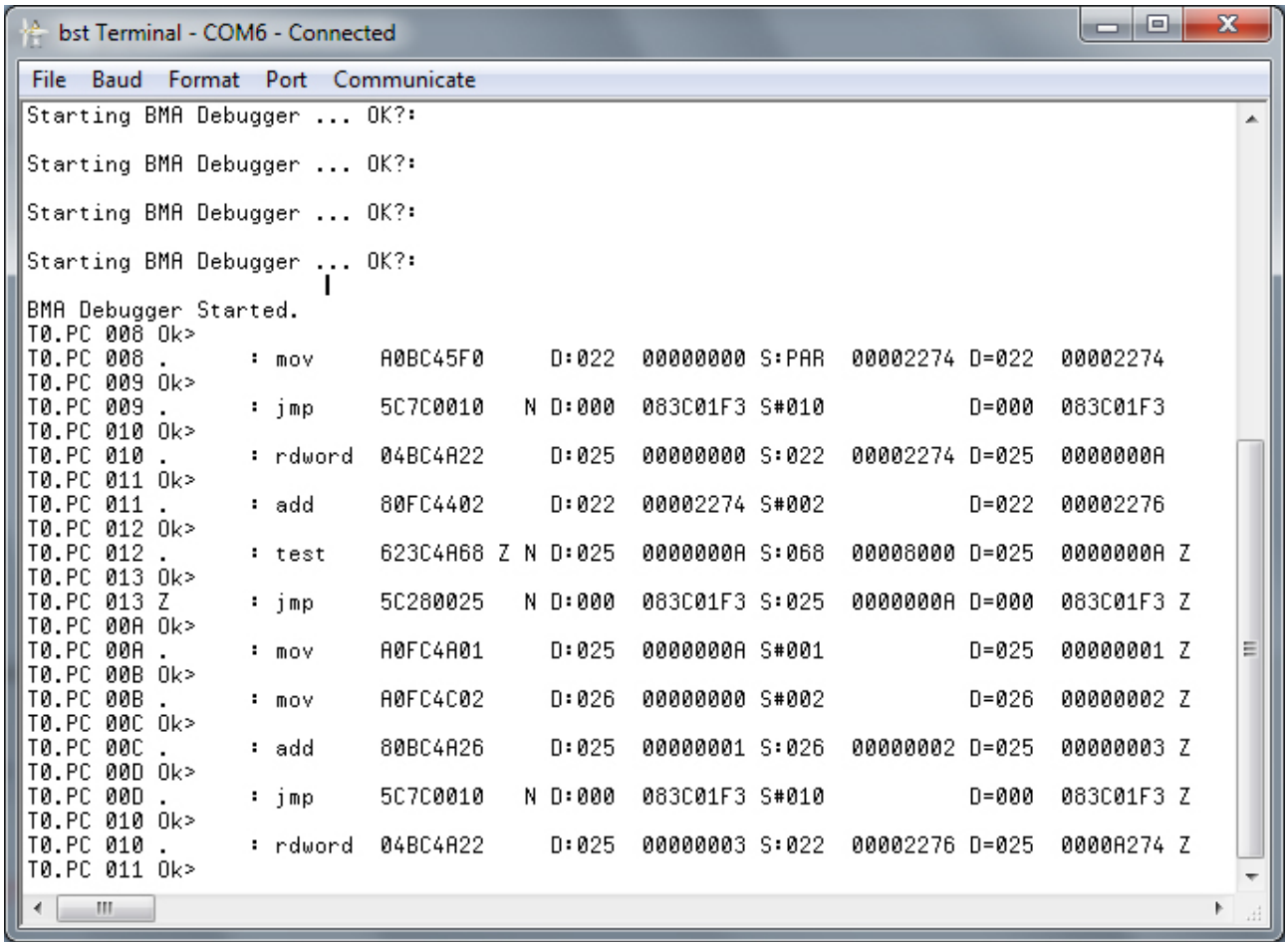
Wird der Maschinencode unter Beachtung dieser Vorgaben gestartet, injiziert der Debugger in den einleitenden acht NOP's seine eigene "Sonde" und es kann über eine serielle Konsole vom Host-PC aus beobachtet und gesteuert werden, wie die Cog arbeitet und was genau in der Cog geschieht.

Ich habe unseren Sputnik entsprechend verbessert (sonde-1), um an einem einfachen Beispiel zu zeigen, wie die Cog und der Debugger arbeitet. Und so kann man dieses Werkzeug testen:

1. sputnik-1.spin in BST laden
2. Den Code per Taste F10 in den Hive übertragen, am besten in den Chip Regnatix
3. Über den Menüpunkt "View/Seriel Terminal" die serielle Konsole öffnen.
4. Baud = 115200 einstellen
5. Format = 8 Bit, Parity None einstellen
6. Port = COM? die Schnittstelle einstellen, an welcher der Hive angeschlossen ist
7. Communicate/Connect anwählen
8. ENTER betätigen, bis die Startzeile des Debuggers erscheint

Nun sollte sich der Debugger mit der Zeile `Starting BMA Debugger ...OK?:` melden. Betätigt man mehrmals die Return Taste, so wird diese Zeile wiederholt. Erst bei Eingabe von "j" wird der Debugger gestartet und ist betriebsbereit.

Nun steht unsere Telemetrieverbindung in die "andere Welt". Mit der Taste Return kann nun Schritt für Schritt das Maschinenprogramm von Sputnik abgearbeitet werden, und an entsprechender Stelle wird auch die LED an- oder ausgeschaltet, wie das im Programm codiert ist.



Was zeigen uns aber die Werte im Einzelnen? In der ersten Zeile unseres Programms Sputnik wird folgender Befehl ausgeführt:

```
mov   dira, Pin   'Pin auf Ausgabe setzen
```

Der Debugger gibt dazu folgende Ausgabe:

```
T0.PC 008 . : mov A0BFEC0C D:DIRA 00000000 S:00C 01000000 D=DIRA 01000000
```

Ohne jetzt im Detail auf den Maschinencode einzugehen, dafür muss jeder selbst das Propeller-Handbuch bemühen, eine Erklärung der Anzeige:

`T0.PC 008`	Adresse des aktuell abgearbeiteten Maschinencodes
`mov A0BFEC0C`	Befehlsmnemonik mit 32 Bit Befehlscode
`D:DIRA 00000000`	Drain: Zieladresse und der Inhalt vor Ausführung
`S:00C 01000000`	Source: Quelladresse und der Inhalt

`D=DIRA 01000000` Zieladresse und Inhalt nach der Ausführung des Befehls

Wie vielleicht bemerkt, habe ich das Programm von Sputnik 1 ein wenig geändert: Da wir nun keinen waitcnt-Befehl zur Zeitsteuerung mehr brauchen (läuft ja alles im Einzelschritt) habe ich die entsprechenden Befehle entfernt.

Nun haben wir eine leistungsfähige Telemetrie zu unserer Cog-Sonde und können komplexere Dinge erforschen. Folgende Befehle versteht unsere Debug-Sonde nun dank BMA zur Verfügung:

```
ax          : animate with x ms delay per step
bx          : toggles breakpoint at COG address x
c           : clears all breakpoints
D           : dumps all COG values
dx n        : dumps n COG values from x
ftx n v     : fill n HUB addresses with v from x with t type = b,w,l
g           : run COG and stop at any breakpoints
htx n       : dumps n HUB values from x with t type = b,w,l
ix          : step showing result of instructions at x
L           : lists/disassembles all COG values/instructions
l           : lists/disassembles 16 instructions from PC
lx n        : list n instructions s starting at x
n           : step over jmpret and call
pr          : prints special register values PAR, etc...
px          : prints content of COG register number <hex>
r           : resets pc back to starting position
R           : restarts COG
sx v        : sets value at COG address x = v
tx          : switch to COG task x
z           : shows flags
Enter       : single-step
?           : show this help screen
```

Ein ausreichend leistungsfähiges System, um einen ersten großen Spung in eine fremde Welt zu wagen. Im späteren Verlauf der Entwicklung wird der BMA-Debugger zu klein und die Daten zu unstrukturiert, so dass wir eine neue Sonde bauen werden, um den komplexeren Status unserer virtuellen Maschine besser darstellen zu könne. Aber dazu später mehr, vorerst genügt der BMA um grundlegende Strukturen zu erschaffen.

Der Warpkern

Kommen wir zum Warpkern unseres neuen Systems, also zum fertigen mCore und seiner internen Struktur. Der mCore macht quasi aus einer Cog eine CPU, welche wesentliche Befehle der Programmiersprache Forth als Maschinencode verarbeitet. Natürlich ist nicht der gesamte Sprachumfang im mCore implementiert. Vielmehr gibt es im Forth eine bestimmten Grundwortschatz an Wörtern, mit welchen man das restliche Forthsystem in Forth selbst programmieren kann. Bezüglich des Grundwortschatzes gibt es verschiedene Meinungen, aber es ist wahrscheinlich eine gute Idee, sich den nativen Wortschatz eines wirklich in Silizium realisierten Forthprozessors anzuschauen. Dabei sind die Forth-Computer Mark-I & Mark-II sehr lehrreich, da bei diesen Geräten ein Forth-Computer aus TTL-Gattern diskter aufgebaut wurde:

- Mark-I Forth Computer

- Mark-II Forth Computer
- RTX 2000, RTX 2001 Forth CPU

Nun, wir erinnern uns: Wir haben nicht mehr als 496 Speicherzellen für unseren Code und seine Register. Zusätzlich benötigt Forth zwei getrennte Stacks. Da der Maschinencode der RISC-CPUs keine Stacks kennt, müssen wir auch diese quasi per Software realisieren und wenn das Ganze auch noch schnell sein soll, müssen diese Stacks noch mit in den Cog-RAM. Naja, wenn wir eh gerade dabei sind alles selbst zu machen, dann kommt es auf ein paar Stacks auch nicht mehr an und Speicherplatz wird einfach überbewertet… ;)

Kurze Bestandsaufnahme was unsere virtuelle Maschine alles braucht:

- Befehlssatz
- Universalregister für interne Operationen
- Return-Stack mit 32 Einträgen
- Datenstack mit 32 Einträgen

Tja, und schon waren es nur noch 432 Register in der Cog...

Programmspeicher: Eine erste Frage lautet: Wo soll der Programmcode liegen, welchen unsere virtuelle CPU abarbeitet? Nun, im Prinzip gibt es im Hive zwei Möglichkeiten: entweder liegt der Befehlscode im hRAM (32KByte) oder im eRAM (1024 KByte). Im eRAM hätten wir natürlich wesentlich mehr Ellenbogenfreiheit für unsere Programme, aber der Zugriff wäre auch deutlich langsamer (ca. Faktor 20 in PASM) und teilt sich auch noch die Bandbreite mit den anderen Kommunikationsvorgängen. Zumal es auf dem Hive bisher kaum Programme gibt, die auch nur ansatzweise den hRAM von Regnatix füllen. Also kommt der Programmcode in den hRAM. Der eRAM soll aber dennoch über Befehle unserer virtuellen CPU, also so schnell wie möglich, ansprechbar sein. Dort können dann durchaus Daten gespeichert werden.

Befehlscode: Nächste Frage: Welche Breite soll der Befehlscode haben? Ich denke ein Bytecode ist zwar auf den ersten Blick platzsparend, aber bei der zweiten Betrachtung ist es günstiger einen 16 Bit Befehlscode zu verwenden. Warum?

Dafür müssen wir die beiden Adressräume betrachten , in denen wir uns bewegen: Als erstes haben wir den cRAM in der Cog. Die dortigen 496 Register (eigentlich 512) werden mit einer 9 Bit Adresse angesprochen. In diesem Speicher befinden sich unsere primären Funktionen als nativer Maschinencode der coginternen RISC-CPUs. Die einfachste und schnellste Codierung unserer Befehle besteht darin, diese 9 Bit als Einsprungadresse in die Cog Routine zu verwenden. Damit entfällt auf der einen Seite der Wasserkopf eines Befehlsdekoders, auf der anderen Seite passt der Befehlscode aber nicht mehr in einen Bytecode.

Als zweites haben wir den hRAM, welcher mit einer 16 Bit Adresse angesprochen wird. Hier soll später das Programm unserer eigenen CPU gespeichert werden. Da nur in den unteren 32 KByte RAM bzw. Programmspeicher residiert, ergibt sich eine 15 Bit Adresse, was einen 16 Bit Befehlscode nahelegt.

Ich habe das Problem wie folgt gelöst: Im mCore ist der Befehlscode immer 16 Bit groß. Das oberste Bit (Bit 15) ist das Call-Bit. Ist dieses Bit gesetzt, handelt es sich um einen Maschinencodebefehl, welcher in den untersten 9 Bit als Adresse der Cog Routine codiert ist. Die

ungenutzten 6 Bit werden zum Beispiel für 6 Bit Literale verwendet, indem dieser Wert in den Befehlscode eingebettet wird. Ist dieses Bit nicht gesetzt handelt es sich um einen Call Befehl und die unteren 15 Bit geben so wie sie sind die Adresse der Routine im hRAM an. Hier die zwei sich aus diesen Überlegungen ergebenden Befehlsformate:

Befehlsformat 1
o - 9 Bit cRAM Adresse
n - 6 Bit optional eingebetteter Wert

Befehlsformat 2 - CALL
c - 15 Bit hRAM Adresse

Das Programmsteuerwerk unserer CPU muss also anhand des obersten Bits unterscheiden, welches Befehlsformat vorliegt um die restlichen Bits im Befehlscode korrekt interpretieren zu können.

Adressierungsart: Unsere CPU verarbeitet einen 16 Bit Befehlscode. Zugriffe auf den hRAM werden aus Geschwindigkeitsgründen immer an einer 16 Bit Grenze ausgerichtet. Die Ausrichtung ist wichtig für den Zugriff auf andere Datentypen im hRAM:

- 32 Bit Werte müssen aus zwei 16 Bit Zugriffen zusammengesetzt werden
- 8 bit Werte (z. Bsp. in Strings!) müssen immer auf die nächste 16 Bit Grenze aufgefüllt werden

Verarbeitungsbreite und Stacks: Die RISC-CPUs in den Cogs sind reinrassige 32-Bit-CPUs. Also wird unsere virtuelle CPU ebenfalls eine Verarbeitungsbreite von 32 Bit haben. Alles andere in meinen Augen keinen Sinn.

Die beiden Stacks haben eine Tiefe von je 32 Einträgen. Da bei logischen und arithmetischen Operationen immer mit 32 Bit Daten gearbeitet wird, ist das für den Datenstack sehr effizient und schnell.

Der Returnstack hingegen speichert immer nur 15 Bit hRAM Adressen, wodurch jede Speicherzellen im Returnstack nur zur Hälfte genutzt wird. Im Prinzip wäre es auch möglich, einen 16 Bit Stack zu realisieren, um den Speicher in der Cog besser zu nutzen, aber da die RISC CPUs nativ im coginternen RAM nur auf Longs zugreifen könne, müsste ein Wordzugriff umständlich mit Schiebeoperationen und allerlei Logik codiert werden, was die Sache verkompliziert und eventuell sogar den eingesparten Speicher wieder verbraucht. Eine andere Möglichkeit wäre auch die Auslagerung des Returnstacks in den hRAM unter Verwendung von 16 Bit Zugriffen. Da aber beide Stacks ein zentrales Element unserer Stackmaschine sind, würde dadurch deutlich die Performance sinken. Also bleibt es momentan bei einem Returnstack in der Cog mit dem Kompromiss der ineffizienten Speichernutzung bei maximaler Geschwindigkeit.

Befehlsdekoder/Steuerwerk: Zurück zur Befehlsverarbeitung in unserem mCore. Was für Vorgänge haben wir bei der Verarbeitung der Befehle in unserer virtuellen CPU? Ein kurzer Programmablaufplan für eine VM zur Verarbeitung eines vereinfachten Befehlscodes:

Maschinencodebefehl
o - 9 Bit cRAM Adresse
x - ungenutzte Bits

1. A = (IP) Befehlscode einlesen
2. IP = IP + 2 Befehlszähler erhöhen
3. FUNC(A) Befehlscode ausführen
4. GOTO 1

Eigentlich eine sehr einfache Sache. Kodiert in PASM sieht das wie folgt aus:

```
DAT
' ------------------------------------------------------------------------
' befehlsdekoder
' ------------------------------------------------------------------------
' 16 bit opcode:
' xxxxxxxa_aaaaaaaa     a - 9  bit cog ram adresse, primärer code


m_next                  rdword  REG_A,IP        ' reg_a = (ip)
                        add     IP,#2           ' ip = ip + 2
                        jmp     REG_A           ' opcode ausführen

' ------------------------------------------------------------------------
' befehlscodefunktionen
' ------------------------------------------------------------------------

m_f1                    nop                     ' funktion 1
                        jmp     #m_next

m_f2                    nop                     ' funktion 2
                        jmp     #m_next

m_f3                    nop                     ' funktion 3
                        jmp     #m_next
```

Gehen wir einen Schritt weiter, indem wir dem Befehlsdekoder nun beibringen, unsere erdachten zwei Befehlsformate korrekt zu interpretieren. Entsprechend erweitern wir unseren Programmablauf:

`1nnnnnnoooooooo` Befehlsformat 1
o - 9 Bit cRAM Adresse
n - 6 Bit optional eingebetteter Wert

`0ccccccccccccccc` Befehlsformat 2 - CALL
c - 15 Bit hRAM Adresse

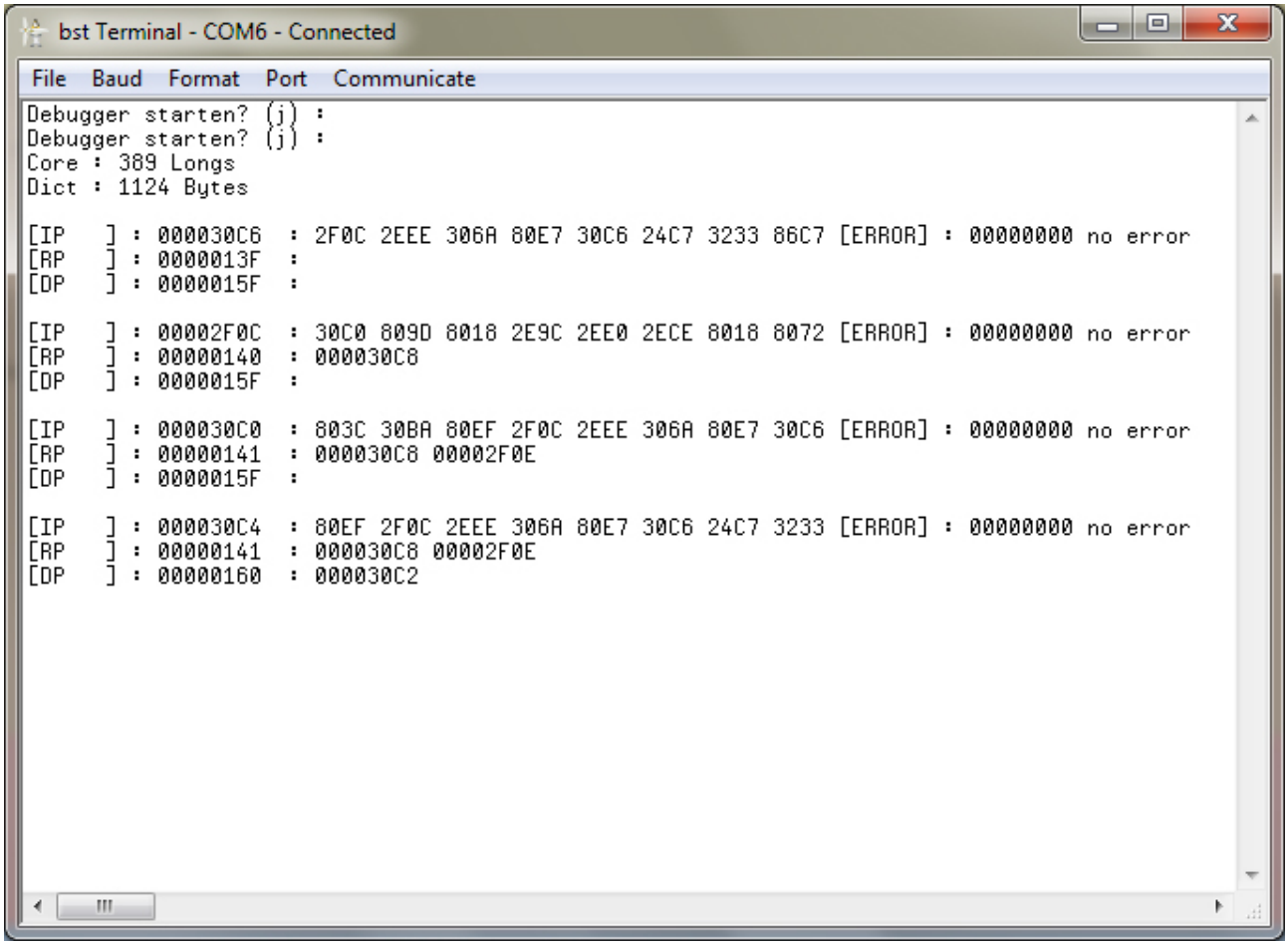
1. A = (IP) Befehlscode einlesen
2. IP = IP + 2 Befehlszähler erhöhen
3. Bit 15 = 1 Befehlsformat abfragen
4. JA: FUNC(A) Format 1 ausführen
5. NEIN: CALL(A) Format 2 ausführen
6. GOTO 1

Und nun wieder die Umsetzung in PASM:

```
DAT
' ------------------------------------------------------------------------
' befehlsdekoder
' ------------------------------------------------------------------------
' 16 bit opcode:
' 1xxxxxxa_aaaaaaaa     a - 9  bit cog ram adresse, primärer code
'                       x - 6  bit immediate literal
' 0aaaaaaa_aaaaaaaa     a - 15 bit hub ram adresse, sekundärer code


m_next                  rdword  REG_A,IP            ' reg_a = (ip)
                        add     IP,#2               ' ip = ip + 2
                        test    REG_A,M_PRI     wz  ' bit 15 = 0?
                        '-------------------------
        if_nz           jmp     REG_A               ' bit 15 = 1
                        '-------------------------
                        movd    :modify,RP          ' bit 15 = 0 --> call
                        add     RP,#1               ' | ip --> rs(rp)
  :modify               mov     IP,IP               ' |
                        mov     IP,REG_A
                        jmp     #m_exit2            ' next

' ------------------------------------------------------------------------
' befehlscodefunktionen
' ------------------------------------------------------------------------

m_f1                    nop                         ' funktion 1
                        jmp     #m_next

m_f2                    nop                         ' funktion 2
                        jmp     #m_next

m_f3                    nop                         ' funktion 3
                        jmp     #m_next
```

Das ist alles, und ich finde, dass ist verdammt klein und schick. Wenn unser mCore der Warpantrieb ist, dann ist diese Codesequenz quasi die Warpkernspule! Hier ist der zentrale Punkt, an welchem die Befehlsausführung der virtuelle CPU koordiniert wird, hier zählt jeder Befehl, jeder eingesparte Zyklus.

Mit diesem Code und dem BMA-Debugger habe ich nun schrittweise den Core vervollständigt. Im späteren Verlauf der Entwicklung von mental war dieser Debugger aber zu umständlich und ich habe mir temporär Befehle in den Core eingebaut, um die internen Funktion zu steuern. Mit einem kleinen Spincode konnte ich so Stacks und Arbeitsregister der Forthmaschine in den hRAM schreiben und über Variablen Sachen wie einen Einzelschrittmodus steuern. Im folgenden Archiv "sonde-2" ist ein Zwischenstand der Entwicklung mit diesem neuen Forth-Debugger zu finden. Die Hauptdatei ist dabei "lab-vm1-debforth.spin". Startet man diese Datei in Regnatix, so meldet sich nun der neue mCore Debugger. Ein Einzelschritt wird dabei folgendermaßen im Terminal angezeigt:



Wie man erkennen kann, ist die Ausgabe nicht mehr auf den RISC-Maschinencode bezogen, sondern zeigt nun relevante Informationen unserer virtuellen CPU:

```
IP          - Instruction Pointer, dahinter die acht nächsten Befehlscodes
RP          - Retrunstack Pointer, dahinter maximal acht Zellen im Returnstack
DP          - Datenstack Pointer, dahinter maximal acht Zellen im Datenstack
ERROR       - Fehlercode
```

Der Debugger kann über folgende Tasten gesteuert werden:

```
h - help
s - step
r - reset
x - erweiterter registersatz
d - dump forthcode und debugspeicher
e - reset errorcode
g - go
```

Bei diesem Test wird übrigens ein noch fehlerhaftes Wort getestet, um das schon implementierte Wörterbuch auf der Konsole auszugeben. Hat man den Code in Regnatix geladen und geht per Einzelschritt die Befehle durch, so erscheint schrittweise das Wörterbuch auf dem am Hive angeschlossenen Bildschirm. Allerdings ist diese Routine noch fehlerhaft, und verfehlt die Abbruchbedingung. Nun ja, es handelt sich hier um einen realen Arbeitsstand mitten aus der Wachstumsphase von mental - mit allen noch enthaltenen Fehlern und Schwächen.

Betrachtet man sich den mCore in der Datei "m-vm1-run-2.spin", so erkennt man schon viele in die virtuelle CPU implementierte Befehle. So ist im Beispiel schon der Code für die Worte zur Stackmanipulation wie dup/swap/rot und einige Dutzend anderer Befehle vorhanden. Hinter dem mCore befindet sich das Maschinencodeprogramm für die virtuelle Forth-CPU. Dieses Programm ist schon in der Form einer minimalistischen Datenbank als Wörterbuch strukturiert.

Wie schon geschrieben, handelt es sich um einen frühen Zwischenstand, aber wenn man den aktuellen Code von mental betrachte, so findet man schon hier viele grundlegende Konzepte und Strukturen.

In dieser Phase habe ich noch intensiv mit dem Forth-Debugger gearbeitet, bis der äußere Interpreter fertiggestellt war. Wenn dieser Schritt vollzogen ist, kann man Forth sehr komfortabel mit sich selbst debuggen - direkt auf dem Hive und ohne Host. Und kurz nach der Fertigstellung des äußeren Interpreters habe ich so auch sämtliche externen Debuggermechanismen aus dem Code entfernt und nur noch mit m selbst gearbeitet. Man kann also sagen, die Entwicklung hat sich in folgenden Phasen vollzogen:

1. VM erstellen (Innerer Interpreter) - Testwerkzeug: PASM-Debugger mit Host-PC
2. Äußeren Interpreter erstellen - Testwerkzeug: VM-Debugger mit Host-PC
3. Metasystem erstellen - Testwerkzeug: mental

Insgesamt umfasst der Grundwortschatz aktuell folgende Maschinencodebefehle, welche der Core direkt in der Cog ausführen kann:

```
Stackmanipulation       : drop dup pick rot >r r< swap ds
Literale                : lit6b litw litl lits
Vergleichsfunktionen    : < = > <> 0=
Logische Funktionen     : and andn or xor not <shift shift> <rot rot>
Integer Arithmetik      : + - abs negate um* um/mod
Speicherzugriffe        : @ ! w@ w! c@ c! x@ x! alignw cmove +!
Kontrollstrukturen      : i leave exit execute branch 0branch (next) (loop)
                          (+loop) perform
Fehlerbehandlung        : abort
Stringfunktionen        : =str =name
Wortdefinitionen        : (variable)
Chipmanagment           : cnt wait reset
Interchipkommunikation  : a! a@ b! b@
```

Genauere Informationen zur Funktion der Befehlen können der Referenztabelle von mental entnommen werden. Mit diesen grundlegenden Befehlen konnte ich das restliche System erstellen. Eventuell ist es aber auch möglich den Grundwortschatz noch weiter zu reduzieren, aber die Maschinencodebefehle sind natürlich sehr schnell in der Asuführen und ich wollte so viel wie möglich davon im Core unterbringen.

Allerdings habe ich den Befehlssatz des mCore nicht in einem Durchgang implementiert und mit diesem dann das restliche System erstellt. Da ich nicht genau wußte, welche Befehle ich grundlegend brauche, habe ich den Befehlssatz im weiteren Verlauf der Arbeit in mehreren Iterationen vervollständigt. Im Prinzip ist es sogar jetzt noch möglich, einen wichtigen oder zeitkritischen Befehl im Core hinzuzufügen, oder eine selten genutzten Befehl zu entfernen und als sekundäres Wort zu realisieren. Ich kann nicht einmal genau sagen, ob die Dimensionierung der Stacks mit je 32 Einträgen ausreichend ist, aber auch das kann ich zu jedem Zeitpunkt noch in den

Grenzen der verfügbaren Ressourcen anders konfigurieren.

Was ich eigentlich damit sagen will: ich habe hier einige Sachen mehr nach Bauchgefühl oder pragmatischen Gesichtspunkten gestaltet - es ist halt (wenn man iSpin mal vergißt) mein erstes Forthsystem - das nächste System wir besser… ;)

Nun gut, kommen wir nun zum Überbau unseres Warpkerns:

Das Schiff

Nachdem wir nun in einer Cog eine virtuelle CPU erschaffen haben, welche als Stackmaschine Forthbefehle direkt wie Maschinencode verarbeitet, wollen wir uns der Metaebene des Systems zuwenden.

Im Prinzip könnten wir schon jetzt mit dem mCore wie mit jeder anderen CPU Programme im hRAM speichern und abarbeiten lassen. Wir könnten uns mit dem Befehlscodes unserer virtuellen CPU ein Monitorprogramm schreiben, um Programme zu laden, zu speichern und zu starten. Weiter könnten wir einen Assembler schreiben und damit Programme übersetzen, speichern und ausführen.

Das wäre aber irgendwie unsinnig, denn wir können nun schon mit einigen weiteren Strukturen direkt in Forth programmieren, also warum noch einen Assembler bemühen? Denn letztlich wollen wir ja mehr: wir wollen einen Interpreter mit einer Kommandoeingabezeile, einen Compiler, einen Quelltexteditor und eine Speicherverwaltung, um unsere Quelltexte auf dem Datenträger zu verwalten. Kurz, wir wollen ein System, um mit dem Hive ohne Hostcomputer autark auf einer höheren Abtraktionsebene zu programmieren.

Also müssen wir den momentan möglichen Maschinencode für unseren mCore weiter entwickeln. Nun, neben der Parameterübergabe mit zwei getrennte Stacks ist die nächste gute Idee bei Forth das Wörterbuch. Im Prinzip ist die Entwicklung des Wörterbuchs gut nachzuvollziehen: Wenn man einen Computer in Maschinencode bzw. mit einem Assembler programmiert, wird man bald den Wunsch verspüren, die im Speicher befindlichen Unterprogramme nicht allein über ihre Startadresse in einem Monitorprogramm zu nutzen, sondern vielmehr über einen symbolischen Namen. Wahrscheinlich hatte man im Assembler ja schon einen sinnvollen Namen für die Funktion gefunden, kann diesen aber im Monitorprogramm nicht nutzen. Was liegt also näher, als zu sagen: Ok, dann lass uns doch einfach die Unterprogramme in einer Art Datenbank einbetten, wobei jeder Datensatz aus einem Namensfeld und einem Bereich für den Code besteht!

Und genau das ist das Wörterbuch im Forth: Es ist eine minimalistische Datenbank, welche einem Unterprogramm einen Namen und zusätzliche Attribute zuordnet. Und die einfachste Datenbank ist eine verkettete Liste wie es im Forth verwendet wird. Alle neuen Elementen werden am Ende angehängt und beziehen sich auf schon vorhandene Elemente in der Datenbank. Elemente können nur vom Ende des Wörterbuchs her gelöscht werden, wodurch es keine offenen Referenzen gibt, da sich jedes Element nur auf vorherige Elemente bezieht.

Eine solche Struktur kann man sogar manuell in einem Assembler aufbauen, was ich auch genau so getan habe. Schauen wir uns einfach den Anfang des Wörterbuchs von mental einmal an, zu finden im Verzeichnis /rom/regflash.spin:

```
dat                                                     'm-dict: wor

m_dbase
dat                                                     'm: dp
                        ' variable dp
                        word    0
DP_NFA                  byte    s02,"dp"
DP_PFA                  word    (@m_data - @m_cbase) / 4 + PRIM
                        word    @@@m_dpointer

dat                                                     'm: here
                        ' : here ( -- dp) dp w@ ;
                        word    @@@DP_NFA
HERE_NFA                byte    s04,"here"
HERE_PFA                word    @@@DP_PFA
                        word    (@m_wfetch - @m_cbase) / 4 + PRIM
                        word    (@m_exit - @m_cbase) / 4 + PRIM

dat                                                     'm: drop
                        word    @@@HERE_NFA
DROP_NFA                byte    s04 + NF_PR,"drop"
DROP_PFA                word    (@m_drop - @m_cbase) / 4 + PRIM
                        word    (@m_exit - @m_cbase) / 4 + PRIM

dat                                                     'm: dup
                        word    @@@DROP_NFA
DUP_NFA                 byte    s03 + NF_PR,"dup"
DUP_PFA                 word    (@m_dup - @m_cbase) / 4 + PRIM
                        word    (@m_exit - @m_cbase) / 4 + PRIM

dat                                                     'm: pick
                        word    @@@DUP_NFA
PICK_NFA                byte    s04 + NF_PR,"pick"
PICK_PFA                word    (@m_pick - @m_cbase) / 4 + PRIM
                        word    (@m_exit - @m_cbase) / 4 + PRIM

dat                                                     'm: rot
                        word    @@@PICK_NFA
ROT_NFA                 byte    s03 + NF_PR,"rot"
ROT_PFA                 word    (@m_rot - @m_cbase) / 4 + PRIM
                        word    (@m_exit - @m_cbase) / 4 + PRIM
```

Wir sehen hier die Definition der Worte: **`dp here drop dup pick`** und **`rot.`**

Die Bezeichner m_drop m_rot m_dup usw. sind quasi die Maschinencodes unseres mCore. Mit dem Ausdruck **`(@m_rot - @m_cbase) / 4 + PRIM`** löst der Assembler die Referenz korrekt auf und setzt mit **`PRIM`** das oberste Bit um zu kennzeichnen, dass es sich um Befehlsformat 1 (primär) handelt.

Der erste 16 Bit Wert in der Definition ist immer ein Zeiger auf das vorherige Wort - eine verkettete

Liste. Einzig bei dem ersten Element im Wörterbuch ist dieser Zeiger 0. So kann man in einfacher Weise vom Ende des Wörterbuches durch die gesamte Datenbank wandern.

Dieser erste Zeiger wird im allgemeinen als Linkfeld (LF) bezeichnet. Das folgende Feld ist das Namensfeld (NF) welches einen counted String mit dem Namen des Codes enthält. Namen können in mental maximal 16 Zeichen lang sein, weshalb das Längenfeld im ersten Byte 4 Bit belegt. Die oberen Bits werden für spezielle Attribute verwendet. mental verwendet direkt gefädelten Code, weshalb auf das Namensfeld direkt das Parameterfeld (PF) folgt, welches den Code des Wortes enthält.

Die ersten Worte habe ich von Hand in das Worterbuch geschrieben, quasi manuell compiliert. Der Assembler bietet dabei wenigstens den Komfort, die symbolischen Referenzen aufzulösen, so dass man nicht alles im Kopf wandeln muss. Dennoch wäre an dieser Stelle ein Makroasembler eine wunderbare Sache.

Der primäre Wortschatz (unsere Maschinencodebefehle) im Wörterbuch umfasst momentan (12.05.2013) 62 Worte, sekundär sind 117 Worte im Wörterbuch codiert. Insgesamt sind also 179 Worte fest verfügbar, codiert in knapp 3,5 KByte. Über 29 KByte des hRAM sind also für eigenen Code frei, und dass mit integriertem Quelltexteditor und Massenspeicherverwaltung. Zauberei? Genialität? Leider nichts davon! :(

Das Zauberwort im Hive heißt Parallelität auf zwei Ebenen! Wir haben im Hive ja drei Propellerchips mit je acht Cogs, welche alle ihre Ressourcen zur Verfügung stellen. Bei mental war ich bemüht, so viele Funktionen wie möglich vom Master in die Slaves auszulagern, so dass in Regnatix so viel Speicher wie möglich für den Benutzercode verfügbar ist. Zusätzlich wird dadurch die Nebenläufigkeit des Systems erhöht, wie wir gleich noch am Beispiel des Dot-Kommandos sehen werden.

Und damit bin ich persönlich schon wieder auf Neuland angelangt. Klassischen Forthsysteme müssen meist mit einer CPU auskommen, dafür hat das System auch Zugriff auf alle verfügbaren Ressourcen. Einige Konzepte wie der Eingabeparser und die Ausgabeformatierung gehen von der Verfügbarkeit dieser Ressourcen aus. Verlagert man diese in ein anderes parallel laufendes System, müssen einige Dinge anders gestaltet werden.

Als Beispiel soll die Ausgabeformatierung dienen: Nehmen wir das Wort "." (dot) welches einen Wert vom Datenstack als numerische String auf der Konsole ausgibt. Da ist bei einem klassischen System viel zu tun: Der Wert muss entsprechend der aktuellen Zahlenbasis in einen String gewandelt und dann zeichenweise auf der Konsole oder dem Bildschirm ausgegeben werden.

Bei mental ist diese Funktion in den Bellatrix Chip ausgelagert. Der mCore übergibt mit der Ausführung des Wortes "." (dot) nur den obersten 32 Bit Wert an Bellatrix mit der Steuersequenz, diesen als numerischen Wert auszugeben. Nun kann sich der mCore wieder seinem Programmcode zuwenden, während in Bellatrix einer der parallel laufenden Prozessoren den numerischen Wer in einen Ausgabestring wandelt, in den Bildschirmspeicher schreibt usw. Dieses Konzept spart sehr viele Ressourcen im Core und vereinfacht darüber hinaus das System.

Dennoch kann die Auslagerung nicht ohne einige grundlegende Änderungen stattfinden, aber davon mehr im nächsten Abschnitt.

Jetzt wird es bunt!

Parallelisierung ist ein spannendes Thema und an einigen Stellen nur durchzuführen, indem man alte Zöpfe abschneidet. (Das schreibe ich hier mal ganz frech, obgleich ich als Forth-Neuling ja noch nicht mal Zöpfe zum abschneiden habe! :) Diese Sache ist mir besonders bei der Auslagerung von Interpreterfunktionen aufgefallen.

Um Speicher im mCore zu sparen, war es unter anderem das Ziel, auch die Eingabezeile und alle damit zusammenhängenden Wörter auszulagern. Bei den Gedanken zum Thema wurde mir durch die räumliche Trennung der Funktionen die Uneffektivität der bisherigen Vorgehensweise bei der Interpretation einer Eingabezeile bei den klassischen Forthsystemen bewußt. Schauen wir uns einmal an was vor sich geht, wenn man in einem klassischen Forthsystem eine Eingabe tätigt:

1. Trennung eines Tokens aus dem Eingabestring
2. Suche im Wörterbuch nach Token
3. Wenn ja, dann dieses Wort ausführen und weiter bei 1.
4. Wenn nein: Ist dieser Token in eine Zahl wandelbar?
5. Wenn ja, dann Wert auf Stack ablegen
6. Wenn nein, dann Fehlermeldung

In einem System mit einer CPU mit Zugriff auf alle Recourcen ist das soweit ok. Schauen wir uns das in einem verteilten System an:

1. System 1: Trennung eines Token aus dem Eingabestring
2. System 1: Sende Token zu System 2
3. System 2: Suche im Wörterbuch nach Token
4a. System 2: Wenn Token gefunden, entsprechendes Wort ausführen
4b. System 2: Wenn Token nicht gefunden, sende Nachricht an Sytem 1
5. System 1: Ist Token in Zahl wandelbar?
6a. System 1: Wenn ja, sende Wert an System 2
7. System 2: Wert auf Stack ablegen
6b. System 1: Wenn nein, Fehlermeldung

Wie man sieht ist der gleiche Vorgang bei einer verteilten Implementierung komplexer und mit mehreren "teuren" Kommunikationsvorgängen zwischen den Chips behaftet. Als ich versucht habe nach klassischem Muster die ersten Testcodes für diese und ähnliche Funktionen zu entwerfen, sind mir sofort diese unschönen Vorgänge sichtbar geworden und ich habe auch schnell eine Lösung dafür gefunden: vorgeparster Quelltext. Und bei diesem Stichwort musste ich sofort an colorForth denken, obgleich mental nichts mit colorForth zu tun hat!

Aber langsam und nochmal zum mitschreiben: Wie funktioniert das bei einem klassichen Forthsystem? Im Prinzip versucht ein klassisches Forth, einem Token eine Bedeutung durch einen Automatismus zu geben. Es fragt: Ist dieses Toke im Wörterbuch verzeichnet? Wenn ja, haben wir ja eine Bedeutung. Wenn das Token nicht im Wörterbuch verzeichnet ist, wird geprüft, ob es sich um eine Zahl handelt. Wenn ja, wird der numerische Wert auf dem Datenstack gespeichert. Wenn es weder Wort noch Zahl ist, dann gib eine Fehlermeldung aus, soll sich doch der Benutzer darum kümmern...

Bei einem verteilten System gibt es ein verwirrendes hin und her zwischen den parallel laufenden Systemen, wenn man die gleiche Strategie anwendet, wie bei einem System mit nur einer CPU.

Zumindest konnte ich mich nicht recht mit dem aus einem solchen Konzept resultierenden Code anfreunden (ließ sich einfach nicht sauber in einfache Einzeiler zerlegen, ziemlich häßlich), weshalb ich auch weiter über dieses Thema nachdachte. Schließlich war es ja mein erklärtes Ziel, mental so einfach wie möglich zu gestalten!

Letztlich löste ich das Problem mit vorgeparstem Quelltext: Damit kann der ganze verwirrende Code eingespart werden, da der Programmierer ja schon bei der Eingabe genau weiß welche Bedeutung die einzelnen Token haben. Was fehlt ist nur eine Möglichkeit für den Programmierer, diese Bedeutung im Quelltext zum Ausdruck zu bringen. Farben sind dabei eine gute Lösung, denn sie geben zusätzlich dem Quelltext optisch eine Struktur, wie man sie teilweise in Editoren durch Syntaxhervorhebung wieder mühsam mit weiterem Code extrahiert - was in meinen Augen den Mehraufwand bei der Eingabe auch wieder zusätzlich rechtfertigt. Der farbige Code sieht einfach gut aus und durch die eingesparten Zeichen wird er auch noch kompakter.

Einmal erdacht, erwies sich die Verwendung von bedeutungsschwangeren Farben als sehr vorteilhaft: Der Code für den Interpreter ist dadurch enorm klein und es entfallen einige syntaktischer Zeichen und die farbigen Quelltexte sind schön kompakt und übersichtlich.

Natürlich gibt es auch Nachteile: So wird der Code für die Eingabezeile und den Editor etwas komplexer, und es ist kein reiner ASCII-Code mehr möglich. Würde man ein Terminalprogramm vom Host aus nutzen wollen, so kann nun keine "Software von der Stange" mehr verwendet werde - aber wer will schon noch mit einem Host arbeiten, wenn er mental auf dem Hive hatt... ;)

Technisch besteht ein Eingabetoken nun aus einem Farbzeichen und dem String selbst. Die Farbzeichen dienen auch gleichzeitig als Trennzeichen und werden auf dem Bildschirm auch als Leerzeichen dargestellt, welche aber die aktuelle Textfarbe umschalten - es sind also quasi "farbige Leerzeichen".

Hier ein Beispiel für eine Eingabezeile:

```
quad dup * ;       Darstellung auf dem Bildschirm

quad dup * ;       Zeichen im Eingabepuffer

 Colortag Rot: Create
 Colortag Grün: Compile
```

Der Parser nun kann die Eingabezeile in einzelne Token zerlegen, indem er von Colortag zu Colortag springt. Gesteuert bzw. ausgelöst wird dieser Vorgang durch den mCore: ist der Interpreter dort "hungrig" erbittet er vom Bellatrix-Chip einen Token.

Nun, mit dem Code des äußeren Interpreters möchte ich an dieser Stelle die Führung durch den Maschinenraum von mental beenden:

```
m ?cerr cold w@ if 0 dup load cold w! then begin token again ;
```

Erklärung: Nach dem Wortnamen `m` folgt ein wenig unwichtige Core-Logistik und endlich die abschließende Endlosschleife `begin token again ;` des äußeren Interpreters. Wer den Quelltext dieses Wortes im Toolset anschaut, wird dort die Marke `M_START` finden - die Startadresse des Programms für unsere anfänglich besprochene selbst gebaute virtuelle CPU, womit sich der Kreis nun schließt!