# Ein einfacher, ATmega-tauglicher G-Code-Übersetzer

Dieser Text hat noch den Charakter eines Blogs, weil sich die Überlegungen während des Programmierens noch weiterentwickeln. Er wird deshalb noch öfter durch neuere Versionen ersetzt werden ;-).

Eine hilfreiche Quelle bei diesen Überlegungen war der Artikel „An intelligent NC program processor for CNC system of machine tool" (Autor Liu,Y; et.al. Permalink: http://escholarship.org/uc/item/65r5c7tj. Bezüglich der Backus-Naur-Darstellung von formalen Grammatiken gibt es im Internet jede Menge Artikel. Besonders gut finde ich den Artikel „Bäume sind berechnenbar" aus c't 1985, Heft 12, Seite 250ff..

G-Code ist eine „Programmiersprache" für CNC-Maschinen, mit der sich alle Funktionen der Maschine (Vorschub, Drehgeschwindigkeit der Spindel usw.) steuern lassen. Einige Abläufe sind sogar automatisiert, z.B. das Verfahren entlang einer Geraden von Punkt A nach B oder das Einfügen eines Radius mit Mittelpunkt bei X,Y zwischen den Punkten A und B.

Eine G-Code-Anweisung besteht jeweils aus einem führenden Buchstaben, auf den eine Zahl, ein Trennzeichen oder die nächste Anweisung folgt. Ich bezeichne hier eine G-Code-Anweisung als „Satz". Jede Zeile stellte eine „Aktion" dar und darf mehr als einen Satz enthalten.

Der Übersetzer soll folgende Sätze (G-Code-Anweisungen) auswerten können:

| | |
|---|---|
| G... | G-Satz, legt den Type der Aktion fest (z.B. „im Schnellgang in direkter Linie vom aktuellen zum Ziel-Punkt verfahren") |
| X... | X-Satz (x-Koordinate des Zielpunktes) |
| Y... | Y-Satz (y-Koordinate des Zielpunktes) |
| Z... | Z-Satz (z-Koordinate des Zielpunktes) |
| F... | F-Satz, legt die Vorschubgeschwindigkeit fest |
| T... | T-Satz, führt einen Werkzeugwechsel durch |
| (...) | Funktions-Kommentar (K-Satz), wird an PC zurückgesandt, um den Bediener auf etwas aufmerksam zu machen oder ihn zu einer bestimmten Tätigkeit aufzufordern (z.B. das Werkzeug zu wechseln). |

Der Typ Aktion wird mit einer Cardinal-Zahl festgelegt, die im G-Satz auf den Buchstaben 'G' folgt (mit „Maschine" meine ich immer die CNC-Maschine, nicht ihren Steuerungscomputer). Alle nachfolgenden Sätze legen andere Einzelheiten für die Ausführung dieser Aktion fest, z.B. die Zielkoordinaten, die Vorschubgeschwindigkeit oder die Drehzahl der Spindel. Beispiel:

```
...
G0 Z2.0000             ; leitet eine „G0"-Aktion ein. Zielkoordinaten sind x=unverändert, y=unverändert und z=2,0
X22.5564 Y6.8951       ; noch eine „G0"-Aktion, diesmal mit Zielkoordinaten x=22,5564, y=6,8951 und z=unverändert
G1 Z0                  ; jetzt eine „G1"-Aktion mit x=unverändert, y=unverändert und z=0,0
...
```

Alle Aktionen zusammengenommen bilden den „Auftrag". Der Auftrag trifft als Datenstrom (z.B. über eine RS232- oder USB-Schnittstelle) bei dem System ein, das die Maschine steuert.

Normalerweise werden solche Datenströme zunächst von einem Scanner bearbeitet. Er tilgt alle unnötigen Zeichen aus dem Datenstrom, wie z.B. wiederholte Leerzeichen, CR, LF und dergleichen. Dadurch wird die Anzahl der Datenbytes reduziert und dem Parser wird die Arbeit erleichtert. Der Scanner soll aber nicht die Richtigkeit Syntax prüfen, das ist Sache des Parsers. - Das gilt für Programme, die auf „grossen" Computern laufen und über jede Menge Speicherplatz verfügen können. Wenn man das Ganze aber auf dem ATmega32 des Arduino laufen lassen will, muss man von der reinen Lehre Abstriche machen.

Der Speicherplatz auf dem Arduino ist so begrenzt, dass es unmöglich ist, einen ganzen Auftrag erst zu empfangen, dann zu analysieren und abschliessend auf der Maschine auszuführen. Der Auftrag muss schon abgewickelt werden, während der Datenstrom noch läuft. Wenn die Daten drohen, den verfügbaren Speicherplatz zu „überfluten", muss der Datenstrom unterbrochen werden können (z.B. mit dem X-On/X-Off-Mechanismus). Es kann also sein, dass eine Gruppe oder ein Satz noch gar nicht vollständig übermittelt ist, wenn der ATmega versucht, sie auszuwerten.

Der Datenstrom fällt im Eingangspuffer der UART-Schnittstelle an; bei mir ist das üblicherweise die „FIFO_FROM_UART". Die Organisation dieser FIFO lässt es nicht zu, ein Byte auszulesen und es später an derselben Stelle wieder in die FIFO zurückzuspeichern („Peek"-Operation). Das darf nicht sein, weil die korrekte Reihenfolge der Daten nicht dann nicht mehr gewährleistet ist: Während der Scanner/Parser das Byte ausgelesen hat und es analysiert, könnte die UART ein frisch empfangenes Byte speichern. Wenn der Parser anschliessend das Byte wieder in die FIFO zurücklegen will, kann die Stelle, an der es vormals stand, durch das neue Byte belegt worden sein. Das Byte kann also nicht an seine ursprüngliche Stelle zurückkehren. Wenn man versucht, solche Ereignisse durch eine Interruptsperre zu verhindern, dann gibt's Geschwindigkeitsproblem beim UART-Empfang. Also muss ein Speicherbereich her („TEXTPUFFER"), in dem die Daten, die die FIFO liefert, so zwischengespeichert werden, dass einzelne Bytes sich nach dem Auslesen auch wieder zurückgelegen lassen.

Es liegt nahe, den Scanner mit dem Übertragen Daten aus der FIFO in den Textpuffer zu beauftragen. Er wird dabei, definitionsgemäss, auch gleich alle überflüssigen Zeichen unter den Tisch fallen lassen. Das ist einfach. Ganz toll wär's, wenn er auch gleich sachlich zusammengehörige Daten zusammenhielte. - Da wird's kritisch: Dabei würde er auch Fehler erkennen und melden müssen, was ja traditionell dem Parser vorbehalten wäre. Zusammengehörige Daten - das wäre z.B. „G0" oder „X22.5564". Beispiele für „gute" und „falsche" Sätze sind

G-Satz: „gut" sind „G0", „G 0" oder „g000"; nicht aber „G 0 1","G0.05"
X-Satz: „gut" sind „X22.5564", „x 22.5564", „X 22", „x.5564", „x22."; nicht aber „X22 5564" oder „X 22."CR"5564"
Y-Satz: alles wie beim X-Satz, nur das „X" durch „Y" ersetzen
Z-Satz: alles wie beim X-Satz, nur das „X" durch „Z" ersetzen
F-Satz: alles wie beim X-Satz, nur das „X" durch „F" ersetzen
T-Satz: alles wie beim X-Satz, nur das „X" durch „T" ersetzen
K-Satz: „gut" ist „(eine Nachricht an den Bediener)", nicht aber „) sowas geht nicht)", „(„ ohne zugehörige geschlossene Klammer, „(ein Text, der ... länger ist als der Textpuffer)"

Und was soll der Übersetzer der Maschine sagen, wenn er einen Fehler im Datenstrom findet? Er muss die Maschine in den sicheren, d.h. antriebslosen Zustand bringen. Es hat keinen Sinn, z.B. die fehlerhafte Aktion wegzulassen und mit der nächsten weiterzumachen: Es könnte ja sein, dass die weggelassene Gruppe ein G0-Gruppe war, die das Werkzeug in eine bestimmte Position hätte bringen sollen. Wenn die nächste eine G1-Aktion ist, dann fängt das Werkzeug bei einer ganz verkehrten Anfangsposition zu werkeln an. - Das geht nicht. Es kann keine „recovery" geben; nur den „Not-Aus".

Generell hat der Parser ein Problem, weil die Sätze kein eigenes Endzeichen haben müssen. Müsste jeder Satz z.B. durch ein Leerzeichen (' ') abgeschlossen werden, dann würde der Parser den Satz genau bis inkl. des Leerzeichens verarbeiten, es wegschmeissen (das ' '!) und den nächsten Satz beginnen. Tatsächlich wird das Ende eines Satze aber oft nur durch das erste Zeichen des Folgesatzes angezeigt (der ein Buchstabe sein muss). Also muss der Parser diesen Buchstaben irgendwie zwischenspeichern können, während er den vorangehenden Satz abschliesst. Und er muss danach, wenn er weitermachen will, noch wissen, dass er zuerst in den Zwischenspeicher gucken muss, bevor er den nächsten Satz beginnt.

Eine Aktion bedeutet nicht, dass sich das Werkzeug bewegen muss. Das kann so aussehen:

```
...
G1 X24.3874 Y35.6341 Z0.0000 F50.5 ; Verfährt mit Geschwindigkeit 50,5 zu den Zielkoordinatten x=24,3874, y=45,6341, z=0
Z10                                ; Übernimmt den Aktionstyp G1, Verfahrgeschwindigkeit, x- und y-Koordinaten von der
                                   ; vorherigen Aktion - nur die z-Koordinate bekommt den neuen Wert z=10,0. Deshalb
                                   ; braucht sich das Werkzeug nur entlang der z-Achse zum neuen Zielpunkt zu bewegen.
...
```

Es kann aber auch Folgendes passieren:

```
...
G1 X24.3874 Y35.6341 Z0.0000 F50.5 ; Verfährt mit Geschwindigkeit 50,5 zu den Zielkoordinatten x=24,3874, y=45,6341, z=0
F40                                ; Übernimmt Aktionstyp G1, x- ,y- und z-Koordinaten von der vorherigen Aktion,
                                   ; ändert aber die Verfahrgeschwindigkeit auf den neuen Wert 40,0. Weil sich die Koordinaten
```

```
                                    ; gegenüber den vorherigen nicht geändert haben, bewegt sich das Werkzeug aber nicht.
    ...
```

Es ist klar, dass die nächste Aktion erst an die Maschine übermittelt werden darf, wenn diese die Aktivitäten der aktuellen Aktion abgeschlossen hat.

„Grosse" Computer können auch noch prüfen, ob die Aktionen untereinander in sinnvoller Beziehung stehen. Da muss der Arduino aber endgültig die Waffen strecken.

Auf dem Arduino soll der G-Code-Scanner auch gleichzeitig der Parser sein. D.h. prüft beim Zerlegen des Datenstroms in Sätze gleichzeitig auch die grammatikalische Richtigkeit. Um nicht jedesmal Scanner/Parser schreiben zu müssen, spreche ich ab jetzt nur noch vom Parser - es ist ja klar was ich damit meine. Ich habe mir für den Parser folgende Grammatik ausgedacht (hier in Backus-Naur-Darstellung):

```
Ziffer  ::= '0'|'1'|...|'9'
Bstabe  ::= 'a'|'b'|...|'z'|'A'|...|'Y'|'Z'
VorZchn ::= '+'|'-'
DezSep  ::= '.'
Leer     ::= {' '}[#10]
TrennZchn ::= #13
Text     ::= {Ziffer|Bstabe|Leer|#12|#10}
GanzZahl  ::= {Ziffer}
ZahlWert  ::= (DezSep GanzZahl)|(Ziffer GanzZahl[DezSep GanzZahl])
Zahl     ::= [Vorz] ZahlWert
Satz     ::= (BStabe {Leer} Zahl)| '(' {Text} ')'
Aktion   ::= {Satz {Leer}} TrennZchn
Auftrag  ::= {{Leer} (TrennZchn|Aktion)}
```

Bei der Auswertung durchläuft der Parser die Grammatik-Definitionen in umgekehrter Reihenfolge, von unten nach oben. Für ihn besteht der Auftrag aus lauter Aktionen, die Aktionen aus Sätzen, usw. Ob der aktuelle Satz in G-, X- oder sonstwas für ein Satz ist, kümmert ihn nicht weiter. Er kann den nachfolgenden Verarbeitungsschritten die Arbeit aber etwas erleichtern, indem er vor Dezimalzahlen mit führendem Dezimalen-Trennzeichen eine Null einfügt und ein alleinstehendes Dezimalen-Trennzeichen am Ende einer Dezimalzahl kann er weglassen oder eine Null nachfüttern.

Mit der Gruppierung der Sätze wird ein anderer Programmteil beauftragt, den ich „Scheduler" nenne. Er wird jedesmal aufgerufen, wenn der Scanner einen Satz fertig bearbeitet hat, damit er nachgucken kann, ob eine neue Gruppe begonnen hat. Wenn ja, dann legt er die Parameter für die zugehörige Aktion in einer „FIFO_AKTION" ab. Ein dritter Programmteil, der „Dispatcher" beobachtet die aktuelle Aktivität der Maschine und wartet, bis die aktuelle Aktion abgeschlossen ist. Dann schaut er nach, ob auf der FIFO_AKTION eine neue Aktivität ansteht. Wenn ja, dann nimmt er die notwendigen Einstellungen an den Steuerelementen der Maschine vor und startet die neue Aktivität.

Bei dieser Arbeitsteilung fängt der Parser alle „Schreibfehler" ab, der Scheduler kann also auf fehlerfrei geschriebenen Sätzen aufbauen; logische Fehler können diese Sätze immer noch enthalten. Es ist z.B. sinnlos, wenn ein G-Satz als Zahl eine Dezimalzahl enthält. - Tatsächlich gibt es bei grossen Maschinen inzwischen aber so viele G-Sätze, dass tatsächlich auch Zahlen mit Dezimalpunkten zugelassen sind. Bei unserem Arduino erlauben wir sowas aber nicht.

Der Scheduler bekommt die Daten Satz für Satz zu sehen. Er hat die Aufgabe, die Daten aus den Sätzen, vor allem die Zahlenwerte, in ein Format zu übersetzen, mit dem das Steuerprogramm für die Motoren der Maschine direkt angesprochen werden können. Dazu hat er seine eigene Grammatik, die auf den Definitionen der Parser-Grammatik aufbaut:

```
CardZhl   ::= Ziffer GanzZahl
GSatz     ::= 'G' CardZahl
XSatz     ::= 'X' Zahl
YSatz     ::= 'Y' Zahl
ZSatz     ::= 'Z' Zahl
FSatz     ::= 'F' Zahl
TSatz     ::= 'T' CardZahl
KSatz     ::= '(' {Text}
```

Die Grammatik kann einfacher sein als die des Parsers, weil der Parser schon ein fehlerfreies, leicht auszuwertendes Format liefert. Weil dem Scheduler die Sätze einzeln präsentiert werden, ist es jetzt einfach, das Satzende zu erkennen: Es ist immer identisch mit dem Ende des Textes im Textpuffer. Beim K-Satz (Kommentar) braucht man deshalb die abschliessende rechte Klammer nicht mehr. Dafür muss der Scheduler darauf achten, dass die Zahlentypen zu den Satztypen passen: Z.B. ist ein T- oder G-Satz mit einer Dezimalzahl nicht erlaubt. Bei den G-Sätzen bietet es sich an, sie in einen Satz von fortlaufenden Nummern zu übersetzen, die anschliessend als Index in eine Tabelle dienen. Die Tabelle enthält Zeiger auf die Prozeduren, die die gewünschte Bewegungsart des Werkzeugs umsetzen. Kommentare werden ohne die Klammern unverändert über die UART wieder an den PC zurückgeschickt; wenn ein LCD an den Arduino angeschlossen ist, könnten sie auch dort angezeigt werden.

Für jeden Satz werden zwei Variablen eingerichtet. Eine davon enthält den aktuell gültigen Wert, die andere nimmt neue Werte auf, die Parser und Scheduler aus dem Datenstrom herausgefiltert haben. Wenn der Dispatcher aufgerufen wird - und das tritt immer dann ein, wenn eine vollständig und fehlerfrei empfangene und ausgewertete Aktion ansteht - dann soll er die neu eingestellten Werte mit den aktuellen vergleichen. Sofern sich die Zielkoordinaten nicht geändert haben, braucht er nur die aktuellen Werte durch die neuen zu ersetzen. Wenn sich allerdings eine der Zielkoordinaten verändert hat, dann muss er darüber hinaus auch noch die entsprechende Bewegung der Maschine einleiten.

**Programmierung des Parsers**

Nun können wir uns um die Einzelheiten der drei Programmteile kümmern. Der Parser macht mir die meisten graue Haare, wenn ich an folgende Situation denke: Er stösst beim Einlesen einer Zahl plötzlich auf eine leere FIFO. Das kann zweierlei bedeuten: Entweder ist er schneller gewesen als die UART, die mit der Lieferung der nächsten Stellen der Zahl nicht nachkommt. Es kann aber auch sein, dass die UART Däumchen dreht, weil sie die letzte Stelle der Zahl schon empfangen hat. Dann hätte der Parser genau diese letzte Stelle des Auftrags gerade aus der FIFO geholt. Der Parser kann jedenfalls nicht entscheiden, vor welcher der beiden Situationen er gerade steht.

Hier gibt es einige Lösungsmöglichkeiten, die ziemlich kompliziert zu werden drohen: Man könnte dem Parser gestatten, seinen Zustand (d.h. die Inhalte aller benutzten Register, den Stackzeiger und den aktuellen Wert des Programmzählers) „einzufrieren" (Aufruf „PARSER_SUSPEND"), ins Hauptprogramm zurückzukehren und abzuwarten, bis das Hauptprogramm feststellt, dass sich neue Daten in der FIFO befinden. Dann würde das Hauptprogramm den Parser wieder aufrufen. Der würde sich wieder in den alten Zustand zurückversetzen (Aufruf „PARSER_RESUME") und an der Stelle weitermachen, an der er sich unterbrochen hatte. So etwas nennt man auf Computer-Chinesisch einen „context switch". In Echtzeit-Betriebssystemen ist das ein zentrales Programm-Bauteil. Nicht, dass das nicht einzurichten wäre - es ist aber doch mit ziemlich grossem Kaliber auf unsere arme Programmieraufgabe geschossen.

Nächster Lösungsvorschlag: Einfach abwarten. Der Parser geht in eine Dauerschleife, in der er immer wieder nachguckt, ob in der FIFO schon etwas angekommen ist. Dadurch würde „nur" das Hauptprogramm angehalten; der Empfang über die UART ist interrupt-getrieben und funktioniert weiter - es ist also sichergestellt, dass das Programm nicht ewig in der Schleife bleibt. - Es sei denn, es war tatsächlich das letzte Zeichen des Auftrags. Dann kommt nix mehr über die UART und das Programm bleibt in der Schleife. Das kann man machen, ist aber nicht schön.

Noch ein Vorschlag: Wir sagen dem Parser, er solle nur einen Satz auf einmal auswerten und nicht den ganzen Inhalt der FIFO in einem Rutsch. Das geht auch nicht so ganz einfach: Wenn zwischen zwei Sätzen kein Trennzeichen steht, dann hat der Parser das erste Zeichen des Folgesatzes schon gelesen, bevor er überhaupt bemerken kann, dass der aktuelle Satz zuende ist. Wenn es ein Trennzeichen gewesen wäre, dann wär's kein Problem, denn das überspringt er sowieso ohne es zu speichern. Da ist es einfach die Verarbeitung vorübergehend

zu beenden und geordnet aus dem Parser ins Hauptprogramm zurückzukehren. Nur, wenn ein Nicht-Trennzeichen übrig ist, dann kann man das ja nicht wieder in die FIFO zurücklegen - es würde dort (wie oben besprochen) an einem ganz verkehrten Platz landen und die Reihenfolge der Zeichen bis zur Unkenntlichkeit durcheinander bringen. Also schaffen wir ein Plätzchen im Steuerdatenblock des Parsers, an dem er es speichern kann und ein Flag, mit dem er anzeigen kann „ich habe mich unterbrochen; ich weiss aber, wo ich das unverarbeitete Zeichen wiederfinde". Das ist verhältnismässig einfach zu bewerkstelligen.

Es ist ausserdem praktisch, nur einen Satz auf einmal zu „parsen", denn dadurch kann man auch dem Scheduler immer nur einzelne Sätze zur Verarbeitung vorwerfen. Er braucht dann nicht über ganze Satz-Folgen zu grübeln. Es bleibt aber ein kleines Rest-Problem: Wenn der letzte Satz des Auftrags nicht mit einem Trennzeichen endet, dann kann der Parser ihn nie fertigstellen. D.h. er wird nicht ausgeführt. Aber das kann man ertragen, bzw. wenn man daran denkt, den Auftrag immer mit einem Leerzeichen oder einem CR zu beenden, es auch vermeiden.

Oben ist das Wort „Steuerdatensatz" gefallen - davon hatte ich bisher noch gar nicht gesprochen. Ja, jedes Programm hat ein paar Daten, die es für seinen Betrieb braucht. Diesbezüglich gibt es zwei Programmiererfraktionen:

Die eine verwendet fleissig die „.def"-Direktive des Assember-Preprozessors und hält alle Betriebsdaten in den Registern. Der Vorteil dieses Ansatzes ist die hohe Zugriffsgeschwindigkeit. Nur ist es kaum möglich, alle diese Daten ständig in den Registern zu halten, weil andere Programmaufgaben auch so ihre Ansprüche haben und irgendwann stösst man selbst bei 32 Registern an Kapazitätsgrenzen. Also muss man vor jedem Wechsel der Programmaufgabe (in unserem Fall z.B. beim Wechsel vom Parser zum Scheduler) die Betriebsdaten irgendwo zwischenspeichern. Dafür benutzt man meistens den Stack, der dann mehr Platz braucht. Oder man reserviert für jede Programmaufgabe einen Bereich im RAM, in den die Aufgabe beim Wechsel ihre Betriebsdaten abladen bzw. aus dem sie sie wieder aufladen kann; dann braucht man den Stack nicht so auszuweiten. Beides ist leicht zu einzurichten und funktioniert.

Die andere Fraktion hält die Betriebsdaten normalerweise im RAM und holt sie nur bei Bedarf und kurze Zeit in die Register. Diesen RAM-Bereich nenne ich den „Steuerdatenblock". Das kostet etwas mehr Verarbeitungszeit, ist aber bezüglich der Wartung der Programme und des Entlausens vorteilhaft. Einmal kann man den RAM-Bereich nach „Feldern" organisieren, auf die man mit vordefinierten Byte-Offsets zugreifen kann. Beispiel: Für unseren Parser könnten die Definitionen des Steuerdatenblocks so aussehen

```
; Organisation des Steuerdatenblocks GPARSCTL
; (Byte-Offsets der Felder)
;
.equ GPARSCTL_FLAGS     =  0
.equ GPARSCTL_TEXTZGRH  =  1
.equ GPARSCTL_TEXTZGRL  =  2
.equ GPARSCTL_ERRORID   =  4
.equ GPARSCTL_RESTZCHN  =  5

.equ GPARSCTL_LNG = 6

; Organisation der Flags GPARSCTL_FLAGS
; (Bit-Offsets der Flags)
;
.equ PARSER_SUSPENDED  =  0
.equ PARSER_ERROR      =  1
.equ PARSER_AKTIV      =  7
```

Am Anfang des Hauptprogramms braucht man dann nur einen Bereich im RAM zu reservieren, z.B. so

```
...
.equ ...              = ...
.equ GCODEPARSER      = ...
.equ ...              = GCODEPARSER + GPARSCTL_LNG + TEXTPUFF_LNG
```

Das ist ein bisschen umständlich, aber mit der übersichtlicheren Methode

```
.DSEG
.ORG 0x0060
...
GCODEPARSER:    .BYTE GPARSCTL_LNG + TEXTPUFF_LNG
...
```

funktioniert's leider nicht, weil bei mir die Definitionen von  GPARSCTL_LNG und TEXTPUFF_LNG erst weiter hinten im Programm (genauer gesagt in einer eingebundenen separaten Datei) stehen; und „forward references" akzeptiert der Assembler bei .BYTE-Direktiven nicht. Schade.

Wie immer man's auch macht: Wenn man sich die Möglichkeit schafft, den RAM-Inhalt über die UART an den PC schicken zu lassen, dann hat man sich mit dieser Organisation eine hervorragende Möglichkeit geschaffen, die Betriebsdaten und damit den Programmablauf „in echt" zu beobachten. Beim Entlausen ist das ein Hilfsmittel von unschätzbarem Wert. Wenn man zuätzlich die Felder im Steuerdatenblock nur mit den Anweisungen wie

```
...
    ldd zl,low(GCODEPARSER)
    ldd zh,high(GCODEPARSER)
    ...
    ldd yl,z+GPARSCTL_TEXTZGRL
    ldd yh,z+GPARSCTL_TEXTZGRH
    ld r16,y+
    std z+GPARSCTL_TEXTZGRL,yl
    std z+GPARSCTL_TEXTZGRH,yh
...
```

anspricht, dann braucht man den Zeiger auf den Steuerdatenblock nur einmal in das z- oder y-Doppelregister zu laden, um dann beliebig auf die Felder zugreifen zu können. Dann ist es auch sehr einfach, die Felder im Steuerdatenblock anders anzuordnen oder Felder hinzuzufügen: Man braucht nur die Definitionen zu ändern, der Rest erledigt sich „von selbst". Wegen dieser Flexibilität und der Entlausungs-Hilfe bin ich Mitglied der „Betriebsdaten im RAM"-Fraktion geworden.

Aber zurück zur Organisation des Parsers. Wir haben oben gesehen, dass beim Auswerten der Sätze die Einzelheiten der nächsten Aktion zusammengestellt werden. Beim Eintreffen des CR ist die Aktion komplett beschrieben und zur Ausführung bereit. Die Frage ist, ob man alle Sätze zusammenträgt und sie dann en bloc auswertet oder jeweils jeden Satz einzeln auswertet, sobald er vom Parser fertiggestellt ist. Die reine Lehre sieht die Auswertung en bloc durch den Scheduler vor. Mit dem ATmega ist es aber zweckmässig, mal wieder den „häretischen" Weg zu gehen. Wir werden also dem Parser erlauben, jeden Satz einzeln auswerten zu lassen, sobald er ihn fertiggestellt hat.

Dann gucken wir uns einmal den Scheduler an.

Die zentrale Frage ist hier, wie die Daten sinnvoll und mit vertretbarem Platzbedarf zu organisieren sind.

4.1 Lexical analysis
The major functionality of lexical analysis is to merge a sequence of characters from the input NC program into sequence of words, which is a high-level representation unit, an example is shown in Figure 5. Meanwhile, in this step, all blank and comments within the program will be deleted. After lexical analysis, a symbol table with the same information but more systematic compared to the original character-based program will be built. During analysis, all character-based error will be checked, for example whether the unacceptable address letters has been used or not. In this paper, one dictionary has been designed in this step to store all the valid address letters.

( nicht kopiert: Figure 4: Proposed NCPP structure.)

Original-G-Code:
N001 G91 G28 X0 Y0 Z0
N002 G00 X100 Y100
N003 S100. M03 M07
/N004 G90 G01 X150 Y150 F50.
N005 G17
N006 G02 X200 Y200 I50 J50
N007 #1001=300
N008 G01 X#1001
……
N110 M05 M09 M30

Übersetzter G-Code:
(N, “001”) (G, “91”) (G, “28”) (X, 0) (Y, 0) (Z, 0)
(N, “002”) (G, “00”) (X, 100) (Y, 100)
(N, “003”) (S, 100) (M, “03”) (M, “07”)
(Op, / ) …
……
(#, 1001) (Op, =) (Lit, 300)
: (G, “91”) calls token
: N/G/M means address
: Op means operator
: Lit means literals
(der Text oben stammt aus: Figure 5: Example of lexical analysis)

4.2 Syntax and semantic analysis
Syntax analysis is to determine if a sequence of words within a block is syntactically correct, it is also called intra-block check. It includes the range checking of the data portion of a word and the parameters format checking.
Semantic analysis checks the major inter-block error, which means to make sure whether the logical relationship among several blocks of NC program is correct or not, for example, the same group G/M word cannot appear more than once in a block; block/word sequence should be subject to the G/M word execution order table, spindle should be turned on before any cutting motion starts, etc. Figure 6 shows an example of syntax and semantic analysis.
(N, “001”) (G, “91”) (G, “28”) (X, 0) (Y, 0) (Z, 0)
(N, “002”) (G, “00”) (X, 100) (Y, 100)
(N, “003”) (S, 100) (M, “03”) (M, “07”)
(Op, / ) …
……
(#, 1001) (Op, =) (Lit, 300)
: (G, “91”) calls token
: N/G/M means address
: Op means operator
: Lit means literals
G

91
G
0
Word
X Y Z
28 0 0
Block
Expr Expr
Word Word Word Word
Figure 6: Example of syntax and semantic analysis.
In order to design NCSD for syntax and semantic analysis, three kinds of cases of NC blocks should be analyzed:
Case 1: Extract data expression in each word of NC program. For example, “X [1+2*3-4/5], Zsin[30], #1=2.0 F #1” should be correctly decoded as “X6.2, Z0.5, F2.0”.
9
Case 2: Check syntax relation of words within each block. For example, within block “G17 G02 X10 Y20 I-10 F15 S100 M03”, Z axis value should not appear since in XY circular interpolator plane only X&Y axes are allowed, also F word and S word are mandatory or at least should appear in advance in order to starting G02 circular motion.
Case 3: Check syntax relation among several blocks of NC program. For example, the following blocks:
“N0010 G91 G40
N0020 S100 M03
N0030 G01 G53 X20 F15”are subject to the G53 and G01 syntax rules, G91 in the first block “N0010 G91 G40” is correct, however an error will be found in block “N0030 …” because G53 does not allow G91 mode, which was given in block “N0010…”.
Aiming at processing above three cases, a systematic and theoretical representation of the NC program syntax rules, Extended Backus Naur Form (EBNF) was chosen.
4.3 Extended Backus Naur Form
EBNF is a syntactic metalanguage to describe a computer programming language formally, which was developed by John Backus and Naur in 1960 to describe the syntax of the Algol 60 language. Since then, it has become a very important tool to represent the syntax of a language in computer science. An international standard for EBNF, ISO/IEC 14977 was also published in 1997.
Following is a general definition of EBNF representation:
• Syntax consists of one or more syntax rules;
• A syntax rule is represented as an equation, left symbol of the equal sign is so-called nonterminal symbol while right symbols are a list consisting of either non-terminal symbol or terminal symbol;
• Non-terminal symbol appears at least once at the left of equal sign;
• Terms with quote symbol as header and ender are terminal symbol;
• Interpreting an expression is to apply one or more of the syntax rules.
10
NC language can be considered as one type of simple computer programming language; therefore it’s quite reasonable to use EBNF to represent the NC program syntax. Based on that, the structure of NCSD can be systematically defined.

EBNF für G-Code

Production rules (portion for real value reading EBNF)
1. Real_value=real_number|expression|parameter_value|unary_combo;
2. Expression=’[‘+real_value+{binary_operation+real_value}+’]’;
   Binary_operation=’**’|’/’|’MOD’|’*’|’AND’|’XOR’|’-‘|’OR’|’+’;
3. Parameter_value=’#’+integer_number;
4. Unary_combo=ordinary_unary_combo|arc_tangent_combo;
5. Ordinary_unary_combo=ordinary_unary_operation+expression;

Ordinary_unary_operation='abs'|'acos'|'asin'|'cos'|'exp'|'sin'|'ln'|'sqrt'|'tan';
6. Arc_tangent_combo='atan'+expression+'/'+expression;
7. Real_number=['+'|'-']+((digit+{digit}+['.']+{digit})|('.'+digit+{digit}));
8. Integer_number=digit+{digit};
9. Digit=0|1|2|3|4|5|6|7|8|9;

Meta-identifier in the syntax EBNF

| | |
|---|---|
| = | The symbol on the left is equivalent to the expression on the right |
| + | followed by |
| \| | or |
| [] | zero or one of the expression inside square brackets may occur |
| {} | zero or many of the expression inside curly braces may occur |
| () | exactly one of the expression inside parentheses must occur |
| ; | end of each rule |
| ' | first-quote-symbol |
| - | except |

B.1 Block syntax EBNF:
Block = Group1_Expr | Group0_Expr1 | Other_Expr;
Group1_Expr = G80_Expr | Other_Group1;
Group0_Expr1 = G92_Expr | Other_Group0 ;
Other_Expr = M_Expr | S_Expr | F_Expr | T_Expr ;
G80_Expr = 'g80'| G80_MODAL | ('g80' + Group0_Expr2);
Other_Group1 = G00_Expr | G01_Expr | G02_Expr |G03_Expr | G81_Expr | G82_Expr
|G83_Expr | G84_Expr | G85_Expr|G86_Expr | G87_Expr | G88_Expr
|G89_Expr ;
Group0_Expr2 = G10_Expr|G28_Expr|G30_Expr|G92_Expr;
Other_Group0 = G04_Expr|G53_Expr

B.2 G code group0 syntax EBNF
G04_Expr = 'g04' + 'p' + real_value;
G10_Expr = 'g10' + 'l2' + 'p' + (1-6) + [ Axis_Expr ];
G28_Expr = 'g28' + [ Axis_Expr] ;
G53_Expr = 'g53' + ABS_Mode + G40_Expr + 'g00'|'g01' ;

B.3 G code group1 syntax EBNF
G01_Expr=[Coord_Expr] + [Feed_Mode] + [Distance_Mode]+ [ Unit_Input ] +
[ Plane_Selection ] + ( 'g01' |G01_MODAL ) + Axis_Expr + F_Expr +
ST_Expr + [ Coolant_On ] + [ G40_Expr ] + [ G53_Expr ];
……

B.13 M code syntax EBNF
M_Expr = ( [ Spindle_Oper ] + [ Coolant_Oper ] + [ Change_Tool | ] ) | Prog_Oper;
- Spindle operation syntax EBNF
Spindle_Oper = Start_Spindle | Stop_Spindle ;
Start_Spindle = CW_Spindle | CCW_Spindle ;
CW_Spindle = 'm03' ;
CCW_Spindle = 'm04' ;
Stop_Spindle = 'm05' ;
……

B.14 F code syntax EBNF
F_Expr = (' f ' + real_value) | F_MODAL
……