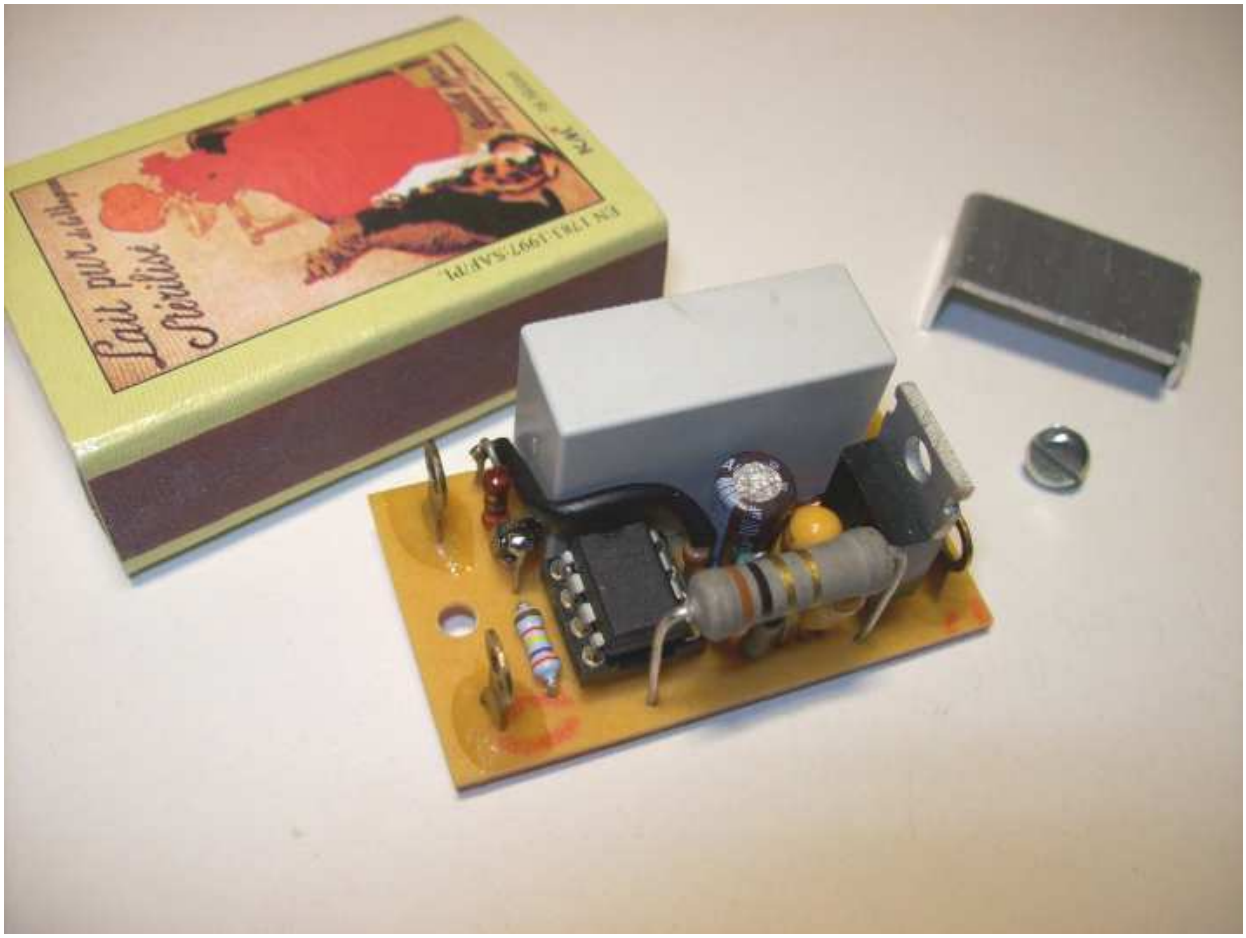
Mini(malistischer)-Frequenzumrichter



Vorwort *3*
Problemstellung *4*
Irrweg 1: Phasenanschnittsteuerung **5**
Irrweg 2: Generatorbetrieb durch Remanenz **6**
Irrweg 3: Käuflicher Frequenzumrichter **7**
Lösungsidee **8**
Patentlage **10**
Realisierung *11*
Gesamtschaltbild **11**
Schaltungsdetails **12**
Software **14**
Aufbau des Frequenzumrichters **20**
Einbau in die Bohrmaschine **21**
Betriebserfahrungen *22*

Abbildungsverzeichnis

Abbildung 1: Daten Labormuster *3*
Abbildung 2 : Drehmoment Kondensatormotor *4*
Abbildung 3: Motor an Phasenanschnittdimmer *5*
Abbildung 4: Ferrograph *6*
Abbildung 5: Hysteresisschleife *6*
Abbildung 6: Käuflicher Frequenzumrichter *7*
Abbildung 7: Motor und verfügbarer Einbauraum *7*
Abbildung 8: Erzeugung 16,7 Hz *8*
Abbildung 9: Prozedur 16,7 Hz *9*
Abbildung 10: Gesamtschaltbild *11*
Abbildung 11: Triac-Treiber und Strommessung *12*
Abbildung 12: Komparatorabfrage *13*
Abbildung 13: Programmlisting (1) *14*
Abbildung 14: Programmlisting (2) *15*
Abbildung 15: Programmlisting (3) *16*
Abbildung 16: Programmlisting (4) *17*
Abbildung 17: Programmlisting (5) *18*
Abbildung 18: Header-Auszug "allpic.h" *18*
Abbildung 19: Header-Auszug "timeloop.h" *19*
Abbildung 20: Compiler-Ergebnis *19*
Abbildung 21: Platinen-Layout *20*
Abbildung 22: Platine mit abgeschraubter Triac-Kühllasche *20*
Abbildung 23: Umrichter, Drehgeber und Haube *21*
Abbildung 24: Endmontage *21*
Abbildung 25: Bohrmaschine betriebsbereit *22*

Doku-Versionen

11.04.2019	Erstdruck
14.04.2019	Tippfehler, Platinen-Layout, Betriebserfahrungen
17.04.2019	Schaltbild korrigiert (Triac = BT136-600D)

Vorwort

Viele elektrische Maschinen sind mit Drehstrom-Asynchronmotoren ausgestattet. Um den Betrieb an nur einer Wechselstromphase zu ermöglichen, wird oft mittels Kondensatoren ein phasenverschobenes Hilfsfeld erzeugt, das den Anlauf und Betrieb ermöglicht.

Um die Drehzahl einer mit einem derartigen Motor ausgerüsteten Säulenbohrmaschine zu steuern, wurde ein Mini-Frequenzumrichter mit folgenden Eigenschaften entwickelt:

+ 16 Standard-Bauteile im Streichholzschachtel-kleinen Modul.
+ Einbau in der Maschine. Ersetzt den Ein-Aus-Schalter.
+ Hoher Wirkungsgrad, kaum Erwärmung der Elektronik bei Betrieb an 0,25 kW-Motor.
+ Kosten des Umrichters: < 5,-- €.
+ Einknopfbedienung inkl. Notaus-Taster.
+ Weitgehend lastunabhängige Drehzahl.
+ Schneller Hochlauf und aktive Bremsung.
+ Da das Motorgebläse drehzahlabhängig arbeitet, kann die Leistungsaufnahme bei geringen Drehzahlen reduziert werden.
- Nur Drehzahlherabsetzung möglich.
- Verschlechterter Motor-Wirkungsgrad durch erhöhte Stromspitzen.
- Keine PFC. HF-Störungen.

Folgende exemplarische Daten wurden am ersten Labormuster gemessen.

Frequenz [Hz]	Anlaufaufnahme [W]	Leerlaufaufnahme [W]	Drehzahl [U/min]
50	250	130	1480
40	110	95	1200
33	130	110	980
25	165	110	720
16	165	115	460
0	0	0	0

Abbildung 1: Daten Labormuster

Problemstellung

Beim Anlauf wird folgende typische Motorkennlinie von links nach rechts durchlaufen.

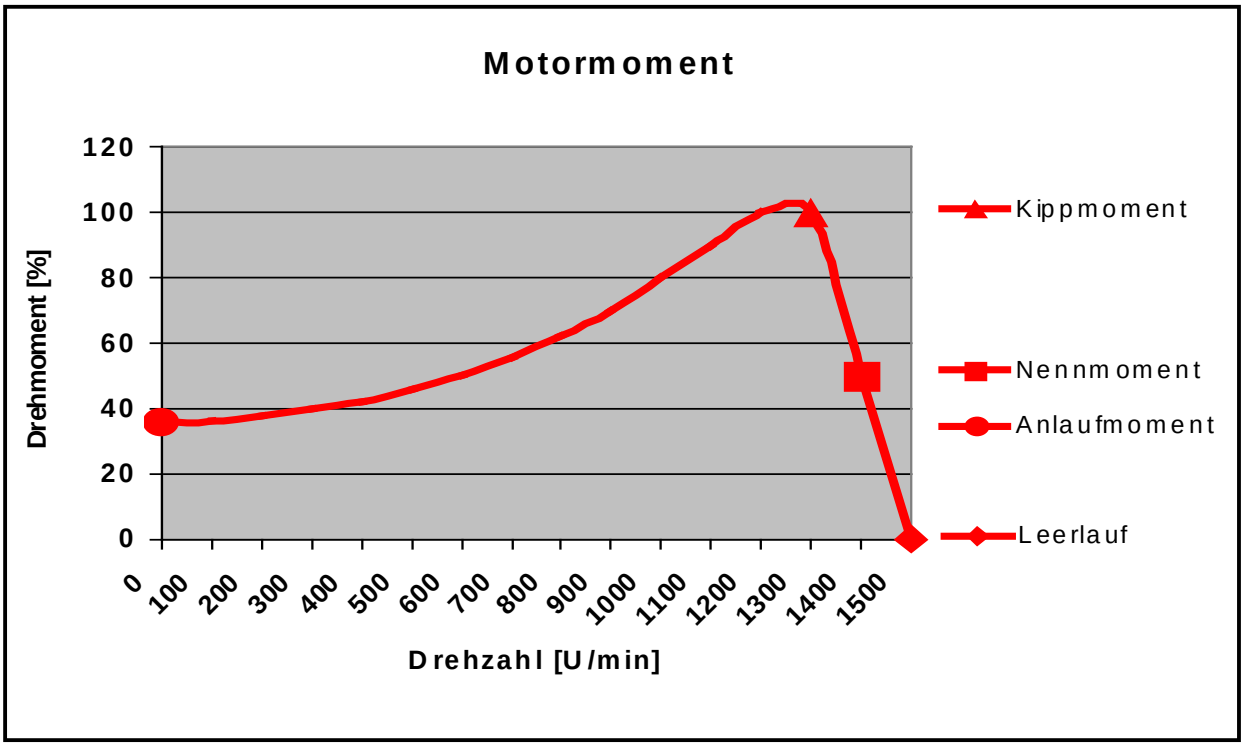


Abbildung 2 : Drehmoment Kondensatormotor

Der Motor startet mit einem geringen Anlaufmoment. Mit steigender Drehzahl steigt auch das Drehmoment. Nach Überschreitung des Kippmoments erreicht der Motor seinen Arbeitsbereich. Maximal kann der Motor eine Leerlaufdrehzahl erzielen, die sich aus der Netzfrequenz pro Minute geteilt durch die Motor-Polpaarzahl (= 2) berechnet.

Man sollte derartige Motoren lastfrei hochlaufen lassen und erst nach Überschreiten des Kippmoments mit dem zwischen Kippmoment und Leerlauf liegenden Nennmoment belasten. Die Drehzahl ist in diesem Bereich nur sehr wenig von der Belastung abhängig.

Erst wenn der Motor stärker als mit dem Kippmoment belastet wird, wird die Kennlinie nach links durchlaufen und der Motor bleibt stehen.

Irrweg 1: Phasenanschnittsteuerung

Stromreduzierende Vorrichtungen, wie zum Beispiel Phasenanschnittdimmer, flachen die gezeigte Kennlinie lediglich ab.

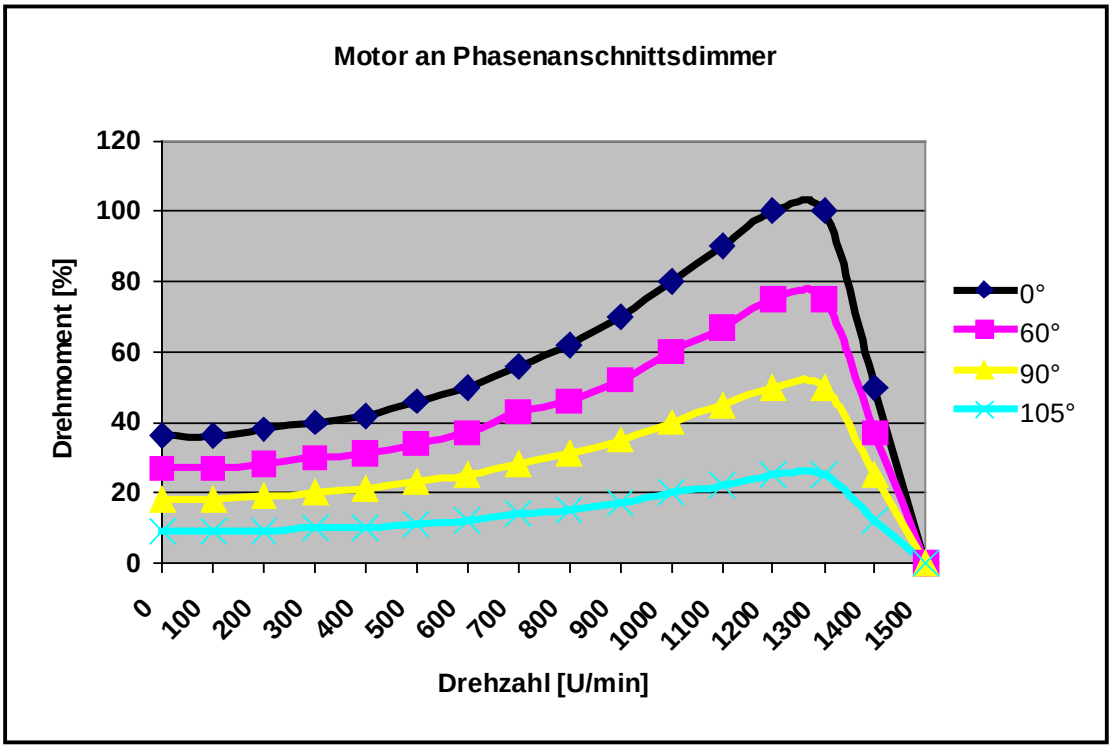


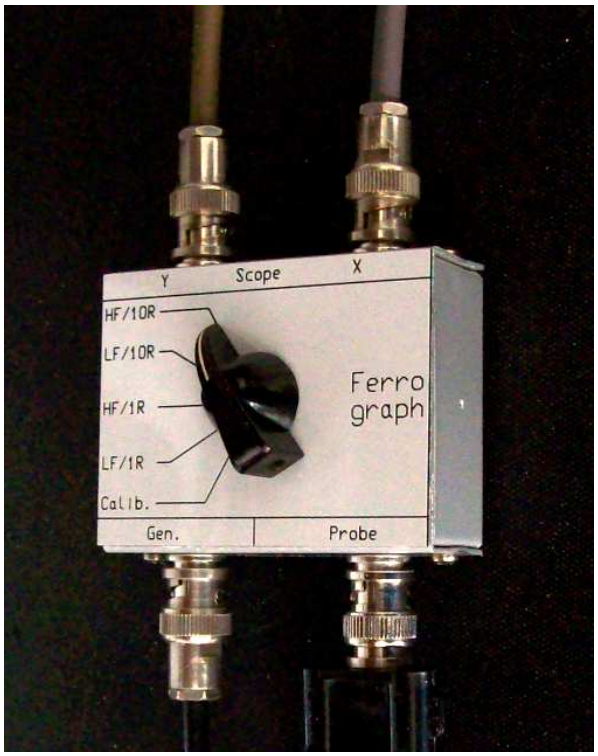
Abbildung 3: Motor an Phasenanschnittdimmer

Mit einem Dimmer kann man die Drehzahl des Motors nur im Zusammenspiel mit einer Last beeinflussen. Ohne Last würde der Motor bis zur normalen Leerlaufdrehzahl hochdrehen.

Eine hohe Leerlaufdrehzahl bei gleichzeitig vermindertem Drehmoment ist in einer Bohrmaschine mit Kondensatormotor nicht brauchbar.

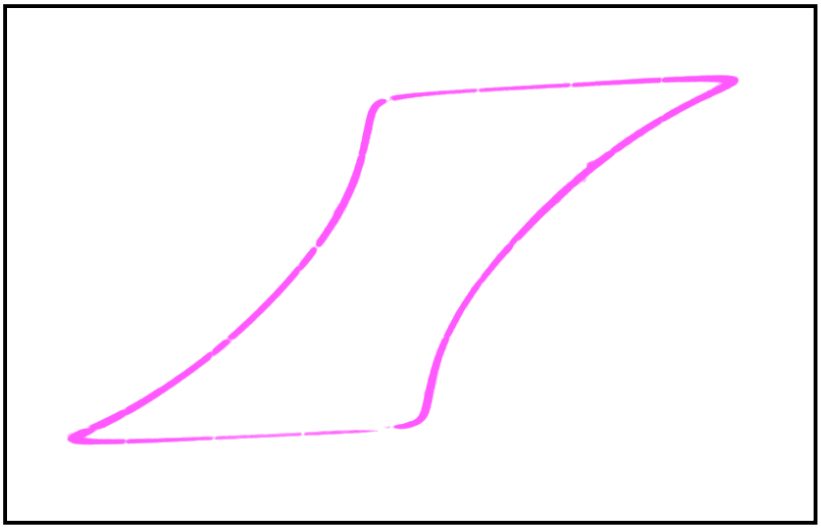
Irrweg 2: Generatorbetrieb durch Remanenz

Ebenso scheiterten Versuche, die EMK des Motors für Regelungszwecke auszunutzen.



Die Vermessung des Motors wurde mit einem Ferrographen durchgeführt.

Abbildung 4: Ferrograph



Es zeigte sich eine rechteckige Hysteresisschleife mit hohen Remanzen.

Das gab Hoffnung per Generatorbetrieb eine Regelspannung zu gewinnen, ohne die Feldspulen bestromen zu müssen.

Abbildung 5: Hysteresisschleife

Leider waren so nur wenige hundert Millivolt und wenige Milliampere nachweisbar. Eine gesonderte Bestromung der Feldspulen hätte den Aufwand hochgetrieben und einen Umbau des Motors erfordert.

Irrweg 3: Käuflicher Frequenzumrichter

Geeignete China-Frequenzumrichter sind schon ab € 60,-- (04/2019) zu bekommen.

Sie erzeugen aus der Netzspannung per PFC eine Zwischenkreisgleichspannung, die zur Versorgung einer mit vielen Kilohertz getakteten PWM-Vollbrücke dient. Hierdurch sind beliebige Sinusfrequenzen und Spannungen in idealer Weise erzeugbar. Durch die hohe Anzahl stromdurchflossener Halbleiter benötigen die Geräte jedoch komplizierte Steuerelektroniken und relativ große Gehäuse (z.B. 165 * 115 * 100 mm) mit Kühlkörpern.

Abbildung 6: Käuflicher Frequenzumrichter

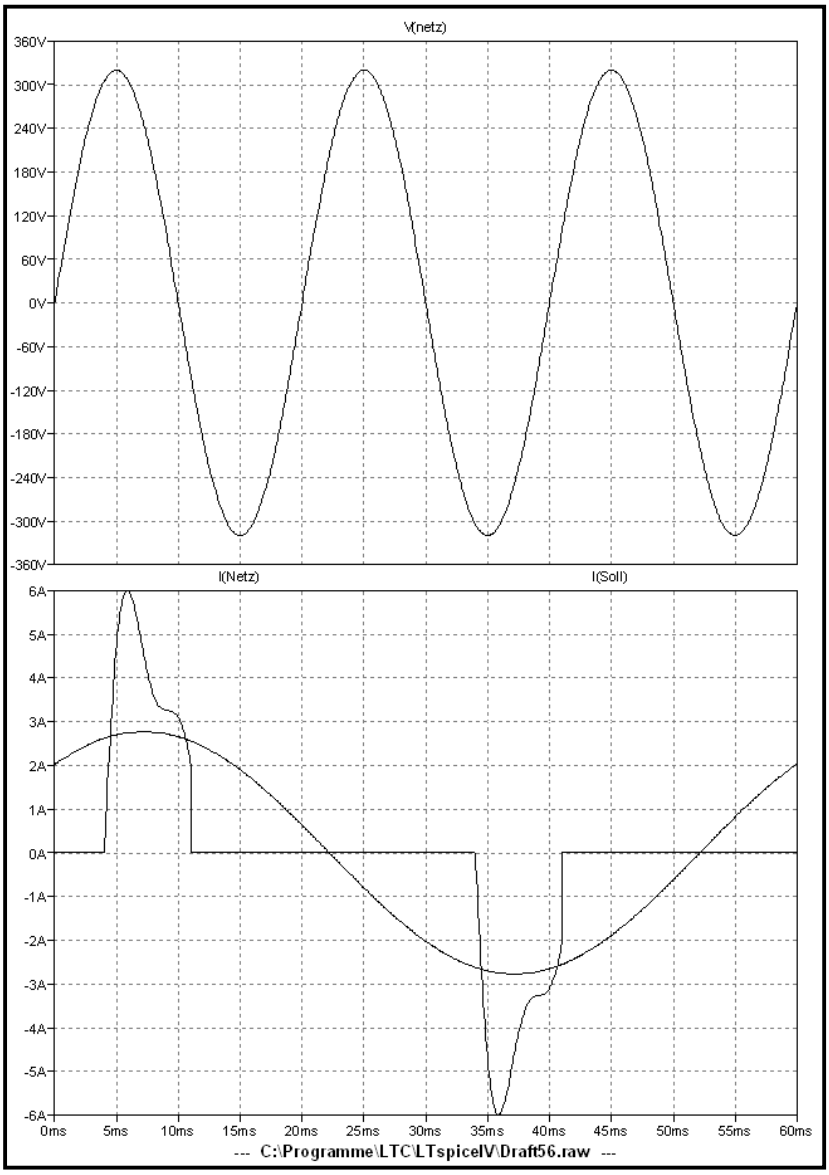
Ein Einbau direkt im Maschinengehäuse an der Stelle, an der sich bisher der Ein-Aus-Schaltkasten befand, ist dadurch verhindert.

Abbildung 7: Motor und verfügbarer Einbauraum

Lösungsidee

Es ist ein einfacher Frequenzumrichter zu entwerfen, der die eingespeiste 50Hz-Netzfrequenz in andere Frequenzen umwandeln und zusätzlich den Motorstrom beeinflussen kann.

Ein Triac kann mehrere aufeinanderfolgende Halbwellen der 50Hz Netzspannung derart anschneiden, dass die gewünschte Motorstromperiode gut angenähert wird. Es soll beispielsweise eine 16,7 Hz-Frequenz erzeugt werden.



In eine Periode der gewünschten Ausgangsfrequenz fallen sechs Netzhalbwellen.

Davon kann man die 1. und die 4. Netzhalbwelle anschneiden und so den Sollstrom annähern.

Durch die Länge der beiden Halbwellen kann der Motorstrom beeinflusst werden.

Durch Anschnitt nur einer Halbwelle kann ein eventueller Motor-Gleichstromanteil kompensiert werden.

Abbildung 8: Erzeugung 16,7 Hz

Es ist sinnvoll, eine gesamte Periode der Sollfrequenz mit einer Makrosprache in folgender Form zu programmieren:

```
void hz16(void)
{
        WAIT_PHASE_UP;                  // warte auf 1. halbwelle (pos)
        triac_fire(symmetrie);          // stromsymmetrierung

        WAIT_PHASE_DOWN;                // warte auf 2.halbwelle (neg)
        WAIT_PHASE_UP;                  // warte auf 3.halbwelle (pos)

        WAIT_PHASE_DOWN;                // warte auf 4.halbwelle (neg)
        if(startup < STARTUP_MS / (6 * 10)) triac_fire(0);          // viel strom
        else triac_fire(40);            // wenig strom nach hochlaufen

        WAIT_PHASE_UP;                  // warte auf 5.halbwelle (pos)
        WAIT_PHASE_DOWN;                // warte auf 6.halbwelle (neg)
}
```

Abbildung 9: Prozedur 16,7 Hz

Zuerst wird auf den positiven Nulldurchgang der speisenden Netzspannung gewartet (1. Halbwelle). Nach einer von der Stromunsymmetrie abhängigen Zeit (in 100 us-Schritten) wird der Triac gezündet.

Danach wird auf den negativen Nulldurchgang der 2. Netzhalbwelle gewartet, ohne dass eine Zündung erfolgt. Danach wird auf den positiven Nulldurchgang der 3. Halbwelle gewartet, ohne dass eine Zündung erfolgt.

Danach wird auf den negativen Nulldurchgang der 4. Halbwelle gewartet..Die Zündung des Triacs erfolgt entweder unverzüglich, um beim Hochlaufen den maximal möglichen Motorstrom zu erreichen. Oder die Zündung erfolgt um 4 ms (40 * 100 us) verzögert.

Zum Schluss wird auf den positiven und danach auf den negativen Nulldurchgang der 5. bzw. 6. Netzhalbwelle gewartet und damit die Prozedur abgeschlossen.

Nach diesem Prinzp lassen sich nahezu beliebige Frequenzen annähern. Es wurden folgende Prozeduren erdacht:

- 0 Hz (= Stillstand)
- 16,7 Hz
- 25 Hz
- 33,3 Hz
- 40 Hz
- 50 Hz (= Vollgas)

Diese Frequenzvielfalt ist angesichts des einfachen Schaltverfahrens mit nur einem Triac in Reihe mit der Netzfrequenz wirklich erstaunlich.

Allerdings ist das Drehmoment bei geringen Drehzahlen – prinzipiell - reduziert.

Und es muss beachtet werden, dass der Frequenzumrichter dem Motor erhöhte Spitzenströme zumutet, was die Motorverluste hochtreibt und zu verstärkten Motor-Geräuschen führt.

Patentlage

Nach Abschluss dieser Vorüberlegungen wurde die Patentsituation recherchiert.

Dabei wurde u.a. das Patent „EP0989666B9“ entdeckt, das offensichtlich den gleichen Ansatz verfolgt. Auch die Erweiterungen zur Anlaufstromerhöhung und zur Gleichstromkompensation werden in Folgepatenten erwähnt. Die Technologie scheint hart umkämpft zu sein, ermöglicht sie doch die Integration eines Frequenzumrichters mit Schnittstellen z.B. direkt im Motorgehäuse. Die Patentnehmer bestehen daher fast ausschließlich aus „global playern“.

Diese Dokumentation nimmt keine Rücksicht auf Patente, da sie sich an Bastler richtet.

Realisierung

Gesamtschaltbild

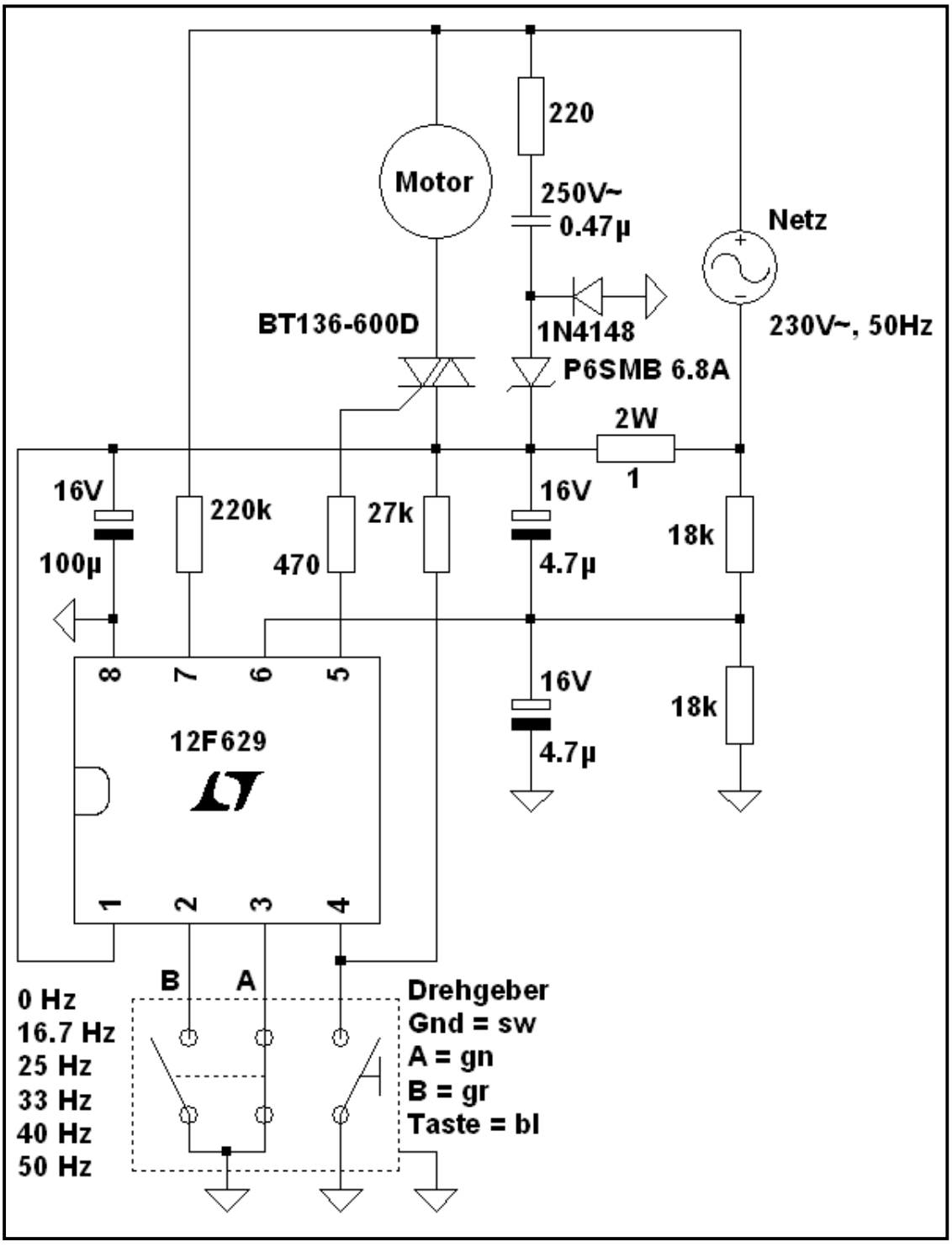


Abbildung 10: Gesamtschaltbild

Als Controller dient wegen einer besonderen Comparatoreigenschaft ein Microchip 12F629. Die Bedienung erfolgt mit einem Drehgeber. Die Stromversorgung erfolgt per Kondensatornetzteil. Gezündet wird der Triac mit negativem Gatestrom. In den so genutzen 3. und 4. Quadranten sind je nur -5mA Gategleichstrom erforderlich.

Schaltungsdetails

Auf ein Snubbernetzwerk (z.B. 220 Ohm und 100nF parallel zum Triac) zur Reduzierung der bei induktiven Lasten auftretenden Spannungsanstiegs kann verzichtet werden (dieser Spannungsanstieg könnte kapazitiv aufs Gate koppeln und eine Fehlzündung bewirken).

- Im Gegensatz zu Transistoren kann ein Triac nur bei Stromlosigkeit löschen. Es kann also keine Rückschlagspannung bei induktiven Lasten entstehen.
- Der Motor ist keine ideale Induktivität. Bei einem typischen Wirkfaktor von cos φ = 0,85 steigt die Spannung am Triac nach Stromlosigkeit auf maximal 150V. So lange dieses in mehr als 3 us geschieht, kann der Triac selbst bei offenem Gate nicht fehlzünden.
- Im Motor befindet sich ein zu dem Zeitpunkt auf 150V geladener 8 uF-Motorkondensator, dessen Energie über die Wicklungswiderstände in die beiden Feldspulen umgeladen werden muss. Die Resonanzfrequenz dieses Kreises liegt weit unterhalb eines Kilohertzes.
- Beim Triac-Sperren ist das Gate nicht offen, sondern wird über den dann leitenden oberen PMOS des Controllerausgangs mit R6 (470 Ohm) nach MT1 bidirektional verbunden.

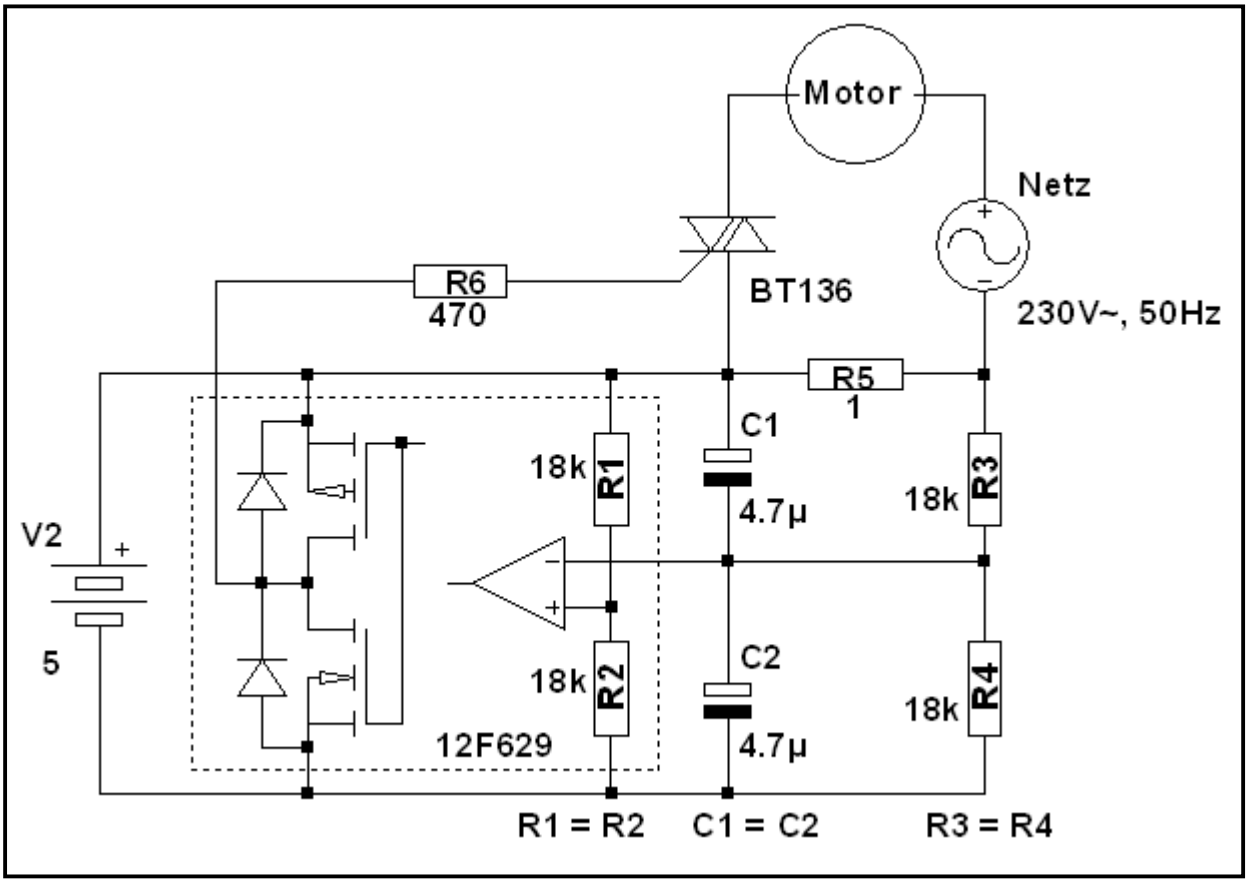


Abbildung 11: Triac-Treiber und Strommessung

Bei der Stromsymmetrierung werden die positiven und negativen Netzhalbwellen vorzeichenrichtig über mehrere Sekunden integriert und dieses Integral dann per asymmetrischem Phasenanschnitt auf Null abgeglichen.

Der Strommesswiderstand R5 liegt auf positiver Versorgungsspannung des Controllers. Diese Versorgungsspannung V2 ist stark mit 50Hz verbrummt, was der Einweggleichrichtung des Kondensatornetzteils, der kleinen Siebkapazität (100 uF) und dem relativ hohen Triac-Gatestrom geschuldet ist.

Die Messung geschieht mit einem internen Komparator daher in Brückenschaltung.

Die Referenzspannung des Komparators muss mit R1 und R2 aus der Vorsorgungsspannung des Controllers gewonnen werden. R1 und R2 werden per Software auf gleiche Werte eingestellt. Am nichtinvertierenden Eingang des Komparators liegt also die mit 50Hz verbrummte halbe Betriebsspannung des Controllers. Sowas kann z.B. ein Tiny85 nicht.

Über C1, C2 (kapazitiver Spannungsteiler, gepaarte Kondensatoren) und R3 und R4 wird dem invertierenden Eingang eine gleichartige Spannung angeboten.

In Reihe mit R3 liegt jedoch der Spannungsabfall des Strommesswiderstands, dessen halbe – durch C1 und C2 gemittelte - Spannung am Komparatoreingang erscheint.

Durch diese Komparatortrick wurden folgende Forderungen erfüllt:

- Herabsetzung der Messspannung auf Vdd / 2
- Messung unabhängig von der Höhe der Vdd oder deren 50Hz Brummen
- Vorzeichenrichtige Integration der Messspannung
- Keine Timing-Probleme der AD-Wandlung durch „Berechnung per Hardware“
- Der Software genügt eine simple Abfrage des Komparatorausgangs:

```
// ********** Stromkompenastion **********************************
if(COUT) {                                      // mittlerer stromfluss positiv
   if(symmetrie < MAX_ANSCHNITT) symmetrie++;   // anschnitt vergrößern
}
else if(symmetrie) symmetrie--;                 // sonst vermindern
```

Abbildung 12: Komparatorabfrage

Software

```
#include "allpic.h"

#pragma config = 0b000110000100          // hex-datei: fuses des 12F629

#define _PHASE              0            // GPIO-Bitposition
#define _SENSE              1            // GPIO-Bitposition
#define _GATE               2            // GPIO-Bitposition
#define _TASTER             3            // GPIO-Bitposition
#define _GEBER_A            4            // GPIO-Bitposition
#define _GEBER_B            5            // GPIO-Bitposition

#define _VR3                3            // VRCON-Bitposition
#define _VREN               7            // VRCON-Bitposition
#define _GPIE               3            // INTCON-Bitposition
#define _GIE                7            // INTCON-Bitposition

#define WAIT_PHASE_UP  while(!GPIO._PHASE)       // warte auf pos. halbwelle
#define WAIT_PHASE_DOWN      while(GPIO._PHASE)// warte auf neg. halbwelle
#define GATE_ON             GPIO._GATE = FALSE       // aktiv low
#define GATE_OFF            GPIO._GATE = TRUE        // aktiv low


//****************** ISR-Geraffel ******************************
/*volatile*/ unsigned char _gpio;            // isr-snap des ports

#define make_save_regs      unsigned char s1, s2   // save-regsiter deklaration
#define save_W              { s1 = W;}
#define restore_W           { s1 = swap(s1); W = swap(s1);}
#define save_regs           { save_W; s2 = swap(STATUS); }
#define restore_regs_return   { STATUS = swap(s2); restore_W; return; }

#pragma origin 4
interrupt server(void)
{
        make_save_regs;                  // sicherungsregister deklarieren
        save_regs;                       // W, STATUS retten
        unsigned char tmp = ~GPIO;       // port samplen

        _gpio.7 = TRUE;                  // Flag: es ist ein int gekommen
        if(tmp._TASTER) _gpio._TASTER = TRUE;       // not-aus geht immer
        _gpio._GEBER_A = tmp._GEBER_A;              // sieht dumm aus....
        _gpio._GEBER_B = tmp._GEBER_B;              // ... ist aber klug

        GPIF = FALSE;                    // muss gezielt gecleared werden
        restore_regs_return;             // viel isr-aufwand für wenig äktschn
}
```

Abbildung 13: Programmlisting (1)

```
//****************** Phasenanschnittssteuerung ****************
#include "timeloop.h"                   // anschnitte in 100us-steps

#define STARTUP_MS              5000    // startupzeit mit erhöhtem strom
unsigned char startup;                  // startup mit mehr strom?

#define MAX_ANSCHNITT           70      // vor nächster halbwelle stoppen!
unsigned char symmetrie;                // stromsymmetrie


void triac_fire(unsigned char anschnitt)
{
        if(anschnitt >= MAX_ANSCHNITT) anschnitt = MAX_ANSCHNITT - 1;
        if(anschnitt > 0) usec100(anschnitt); // delay zum anschnitt
        GATE_ON;                        // triac zünden
        usec100(MAX_ANSCHNITT - anschnitt);  // dauerhaft gatestrom
        GATE_OFF;                       // nun sollte er gezündet haben
}

//********** Hüllkurven für unterschiedliche Drehfrequenzen ****************
void hz0(void)
{
        GATE_OFF;                       // triac dauerhaft aus
        WAIT_PHASE_UP;                  // warte auf 1.halbwelle (pos)
        WAIT_PHASE_DOWN;                // warte auf 2.halbwelle (neg)
}

void hz16(void)
{
        WAIT_PHASE_UP;                  // warte auf 1. halbwelle (pos)
        triac_fire(symmetrie);          // stromsymmetrierung

        WAIT_PHASE_DOWN;                // warte auf 2.halbwelle (neg)
        WAIT_PHASE_UP;                  // warte auf 3.halbwelle (pos)

        WAIT_PHASE_DOWN;                // warte auf 4.halbwelle (neg)
        if(startup < STARTUP_MS / (6 * 10)) triac_fire(0);          // viel strom
        else triac_fire(40);            // wenig strom nach hochlaufen

        WAIT_PHASE_UP;                  // warte auf 5.halbwelle (pos)
        WAIT_PHASE_DOWN;                // warte auf 6.halbwelle (neg)
}
```

Abbildung 14: Programmlisting (2)

```
void hz25(void)
{
	WAIT_PHASE_UP;			// warte auf 1. halbwelle (pos)
	triac_fire(symmetrie);		// stromsymmetrierung

	WAIT_PHASE_DOWN;		// warte auf 2.halbwelle (neg)
	if(startup < STARTUP_MS / (4 * 10)) triac_fire(0);	// viel strom
	else triac_fire(40);		// wenig strom nach hochlaufen

	WAIT_PHASE_UP;			// warte auf 3.halbwelle (pos)
	WAIT_PHASE_DOWN;		// warte auf 4.halbwelle (neg)
}

void hz33(void)
{
	WAIT_PHASE_UP;			// warte auf 1.halbwelle (pos)
	triac_fire(symmetrie);		// stromsymmetrierung

	WAIT_PHASE_DOWN;		// warte auf 2.halbwelle (neg)
	triac_fire(0);

	WAIT_PHASE_UP;			// warte auf 3. halbwelle (pos)
	triac_fire(50);
	WAIT_PHASE_DOWN;		// warte auf 4.halbwelle (neg)
	WAIT_PHASE_UP;			// warte auf 5.halbwelle (pos)
	WAIT_PHASE_DOWN;		// warte auf 6.halbwelle (neg)
}

void hz40(void)
{
	WAIT_PHASE_UP;			// warte auf 1.halbwelle (pos)
	triac_fire(symmetrie);		// stromsymmetrierung

	WAIT_PHASE_DOWN;		// warte auf 2.halbwelle (neg)
	triac_fire(0);

	WAIT_PHASE_UP;			// warte auf 3. halbwelle (pos)
	triac_fire(0);

	WAIT_PHASE_DOWN;		// warte auf 4.halbwelle (neg)
	triac_fire(50);

	WAIT_PHASE_UP;			// warte auf 5.halbwelle (pos)
	WAIT_PHASE_DOWN;		// warte auf 6.halbwelle (neg)
	WAIT_PHASE_UP;			// warte auf 7.halbwelle (pos)
	WAIT_PHASE_DOWN;		// warte auf 8.halbwelle (neg)
	WAIT_PHASE_UP;			// warte auf 9.halbwelle (pos)
	WAIT_PHASE_DOWN;		// warte auf 10.halbwelle (neg)
}
```

Abbildung 15: Programmlisting (3)

```
void hz50(void)
{
        GATE_ON;                        // triac dauerhaft ein
        WAIT_PHASE_UP;                  // warte auf 1.halbwelle (pos)
        WAIT_PHASE_DOWN;                // warte auf 2.halbwelle (neg)
}

//******************** Hauptprogramm ************************************
void main(void)
{
        // *********** initialisierungen *********************
        RP0 = 1;                        // erstmal alle Spezialregister...
        #asm
          DW /*CALL*/ 0x2000 + /*ADRESSE*/ 0x03FF // oscal abholen
        #endasm
        OSCCAL = W;                     // und Oszillatorkalibrierung speichern

        OPTION = 0;                     // global weak-pullup ein (je 20kOhm)
        WPU = _BV(_GEBER_A) | _BV(_GEBER_B);
        TRISIO = _BV(_PHASE) | _BV(_SENSE) | _BV(_GEBER_A) | _BV(_GEBER_B);

        IOCB = _BV(_TASTER) | _BV(_GEBER_A);      // port-change-ints

        VRCON = _BV(_VR3) | _BV(_VREN);     // komp ein, referenz auf vdd / 2
        RP0 = 0;                        // nun die normalen register und ram
        CM2 = TRUE;                     // Komparator-Minuseingang auf GP1

        INTCON = _BV(_GPIE) | _BV(_GIE);      // interrupts freischalten

        GPIO = _BV(_GATE);              // ports ausschalten, gate inaktiv

        timeloop_init();                // timer initialisieren

        unsigned char speed = 0;        // schleifen-variablen init
        startup = symmetrie = _gpio = 0;
```

Abbildung 16: Programmlisting (4)

```
        FOREVER {                       // ewige schleife
         // ********** Drehgeber auswerten **********************
         if(_gpio.7) {                  // ist ein IOC-int gekommen?
          if(_gpio._GEBER_A != _gpio._GEBER_B) speed++;    // geber rechtsrum
          else if(speed) speed--;       // geber links herum
          if(_gpio._TASTER) speed = 0;  // taster schaltet alles aus
          startup = symmetrie = _gpio = 0;  // speed-wechsel: parameter-init
         }
         // ********** Drehfrequenzprogramm auswählen ********************
         switch(speed) {                // es wird je nur ein zyklus ausgeführt
          case 0:   hz0(); break;
          case 1:  hz16(); break;
          case 2:  hz25(); break;
          case 3:  hz33(); break;
          case 4:  hz40(); break;
          default: hz50(); speed = 5; break;  // maximaler speed
         }
         // ********** Stromkompensation *********************************
         if(COUT) {                     // mittlerer stromfluss positiv
          if(symmetrie < MAX_ANSCHNITT) symmetrie++;      // anschnitt vergrößern
         }
         else if(symmetrie) symmetrie--;      // sonst vermindern

         // ********** beim hochlaufen mehr power ************************
         if(!++startup) startup--;

        }
}

/* ENDE */
```

Abbildung 17: Programmlisting (5)

Neben den unveränderten Headern des Compilers werden in „allpic.h“ noch folgende Definitionen eingebunden

```
#define BOOL            bit
#define TRUE            1
#define FALSE           0
#define FOREVER         while(1)
#define _BV(a)          (1 << (a))
```

Abbildung 18: Header-Auszug "allpic.h"

Im Header „timeloop.h“ befindet sich ein einfacher Timer:

```
#define timeloop_init()                                                 \
{                                                                       \
        T1CON = 0x05;                   /* Timer1 parametrieren */\
}

void usec(unsigned char cnt)
{
        TMR1L = ~cnt;
        TMR1H = ~0;
        TMR1IF = FALSE;                 // das int-flag verwenden
        while(!TMR1IF);                 // obwohl ints ausmaskiert sind
}

void usec100(unsigned char cnt)
{
        do {usec(100);} while(--cnt != 0);
}
```

Abbildung 19: Header-Auszug "timeloop.h"

Der kostenlose norwegische CC5-Compiler erzeugt kompakten Code:

```
CC5X Version 3.5, Copyright (c) B Knudsen Data, Norway 1992-2014
 --> FREE edition, 8-16 bit int, 24 bit float, 32k code, reduced optim.
Warning : Main file extension should be '.c'
fu.cp:
 Chip = 12F629
 RAM: 00h : -------- -------- -------- --------
 RAM: 20h : =======. ..****** ******** ********
 RAM: 40h : ******** ******** ******** ********
RAM usage: 10 bytes (7 local), 54 bytes free
 Optimizing - removed 41 instructions (-14 %)
 File 'fu.pic'
 File 'fu.lst'
 File 'fu.hex'
Total of 245 code words (23 %)
```

Abbildung 20: Compiler-Ergebnis

Aufbau des Frequenzumrichters

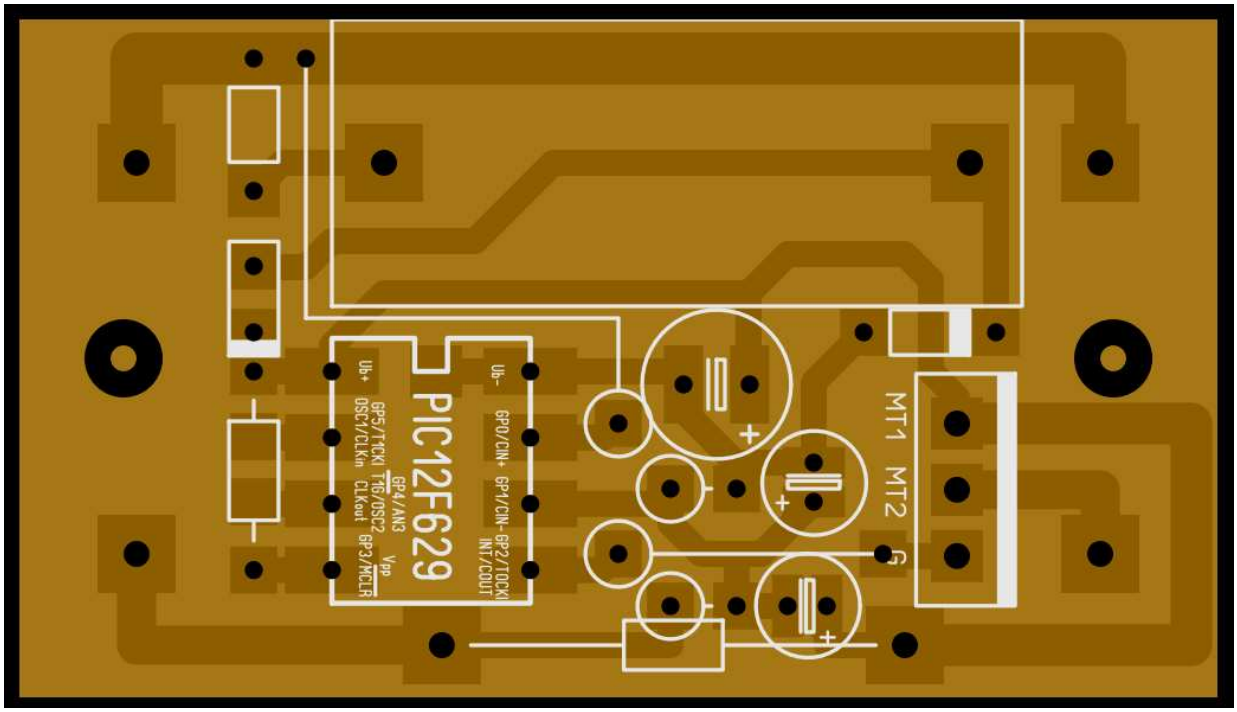


Abbildung 21: Platinen-Layout

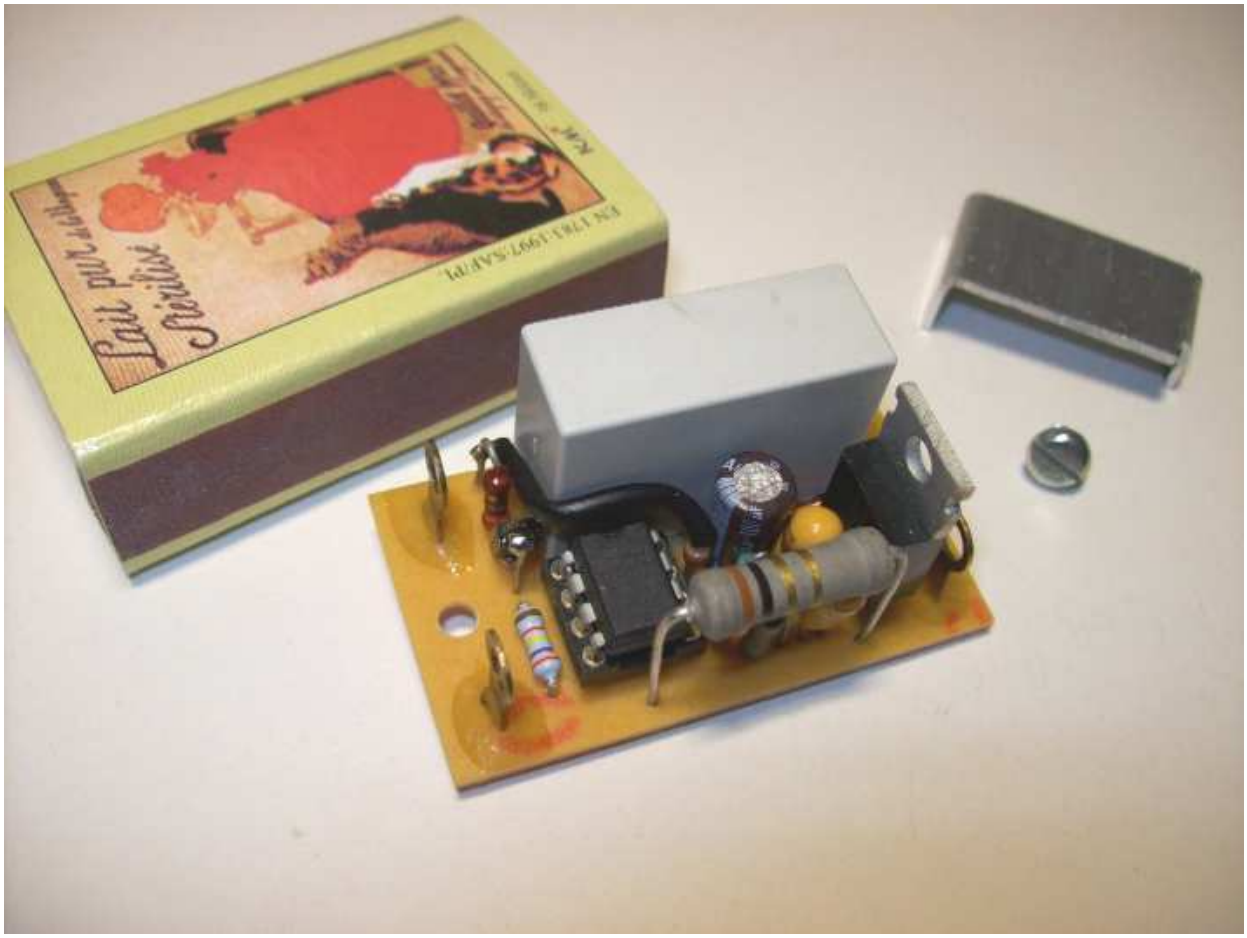


Abbildung 22: Platine mit abgeschraubter Triac-Kühllasche

Einbau in die Bohrmaschine

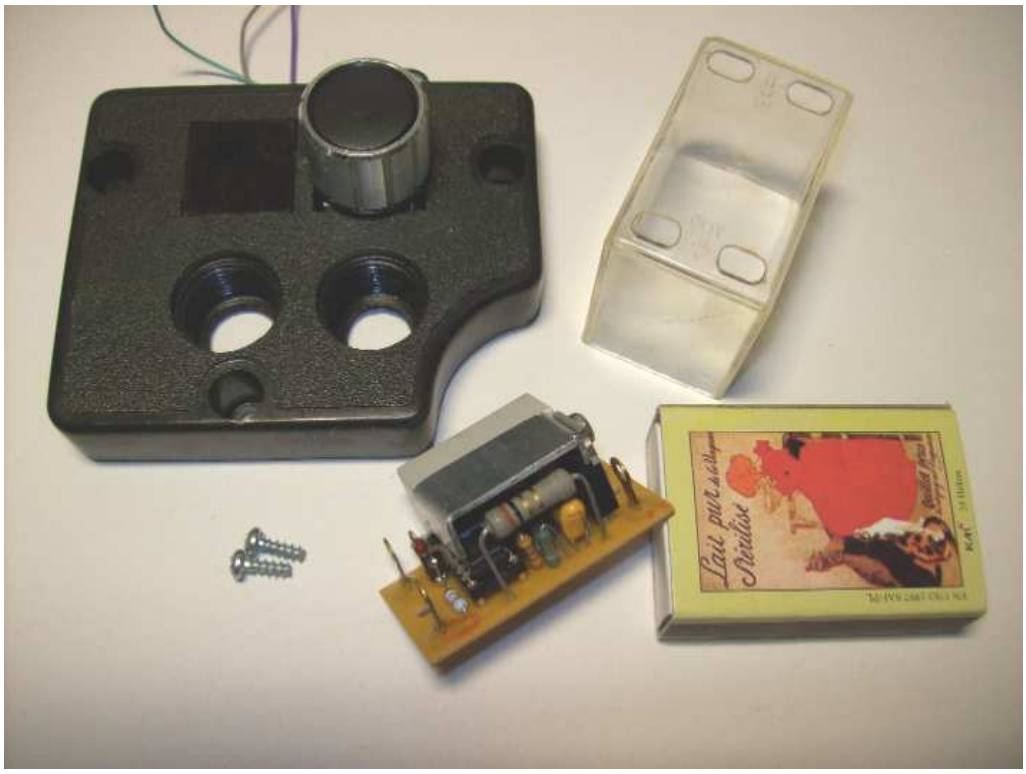


Abbildung 23: Umrichter, Drehgeber und Haube

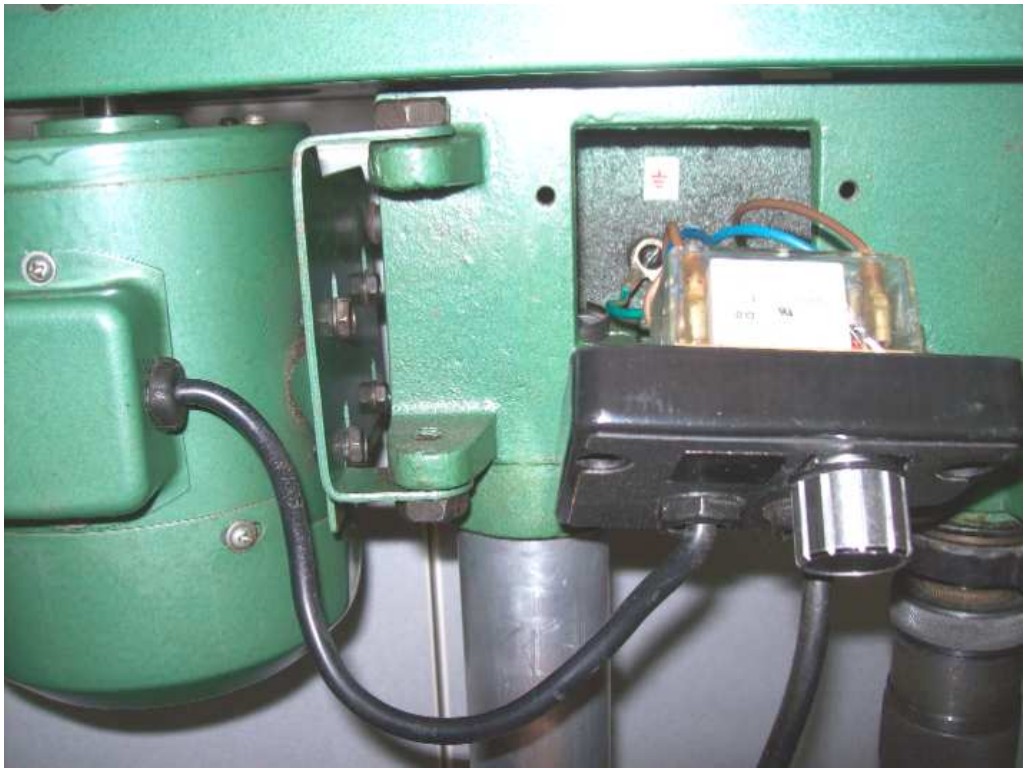


Abbildung 24: Endmontage

Abbildung 25: Bohrmaschine betriebsbereit

Betriebserfahrungen

In der Bohrmaschine verhielt sich der Frequenzumrichter bisher vorzüglich. Das vorsichtige Anbohren sowie niedertourige Alu- und Plastikbearbeitungen wurden deutlich sicherer. Die Bohrerdrehzahl lässt sich nun per Umrichter und Riemengetriebe zwischen 180 und 2600 U/min einstellen. Da das Hochlaufen und Abbremsen wesentlich schneller erfolgt und man nicht mehr mühevoll nach dem richtigen Taster suchen muss, wird der Kurzzeitbetrieb gefördert. Der Motor wurde dabei nicht mal handwarm.

Das Modul wurde zusätzlich auch an dem Kondensatormotor einer Schleifmaschine betrieben. Auch hierbei funktionierte es einwandfrei. Allerdings dreht der Motor der Schleifmaschine mit seiner Polpaarzahl = 1 doppelt so schnell wie der Bohrmaschinenmotor. Handhabungsvorteile bringt eine verlangsamte Schleifmaschine jedoch nicht.