

# Breaking the 32kB Limit

Verwendung des ARM GCC Compilers  
unter der Keil MDK-Lite IDE

Compilieren/Linken von  
Baseflight bzw. Harakiri

© by Joerg Quinten  
aBUGSworstnightmare



**Revisions:**

v0.1: initial release

Um die Firmware des Naze32 – mit den Projekten Baseflight (Original des Naze32 Hardwareentwicklers Timecop) bzw. Harakiri (Erweiterung von Baseflight von crashpilot1000) selber editieren, compilieren bzw. debuggen zu können benötigt man eine IDE welche keine Codegrößenbeschränkung hat. Im Falle der Keil MDK-Lite, der kostenlosen Version der Keil IDE, liegt die maximale Codegröße bei 32kB was für die Naze32 Firmware deutlich zu wenig ist. Was liegt also näher als die Keil IDE so einzurichten dass der GCC ARM Compiler verwendet wird um diese Beschränkung aufzuheben.

Dieses Tutorial soll zeigen welche Schritte dafür notwendig sind. Am Ende verfügt man dann über eine kostenlose, voll funktionale Entwicklungsumgebung für ARM Mikrocontroller und kann sich entweder mit Baseflight/Harakiri oder eigenen Projekten austoben.

## 1. Vorbereitung – IDE und ARM Compiler downloaden und installieren

Zuerst muss die benötigte Software geladen werden. Es handelt sich dabei um:

- die Keil MDK-Lite (32KB) Edition

<http://www.keil.com/arm/mdk.asp>

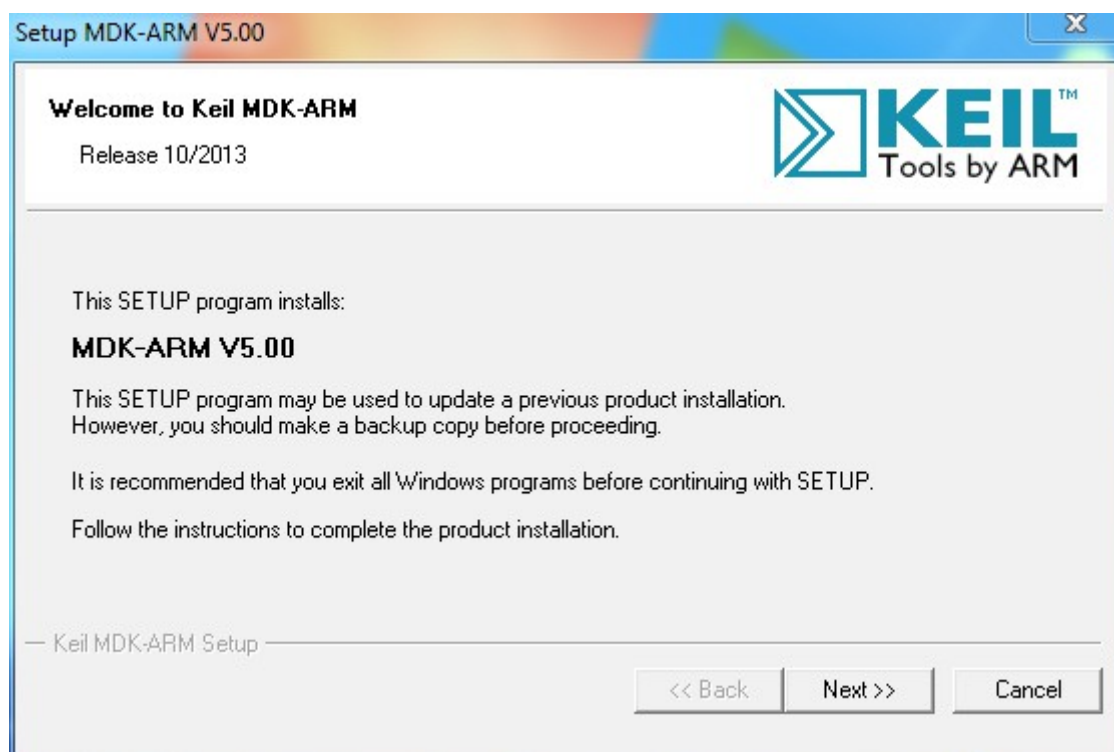
(zum Zeitpunkt der Erstellung dieses Tutorials in der Version V5.00)

- die GNU Tools for ARM Embedded Processors

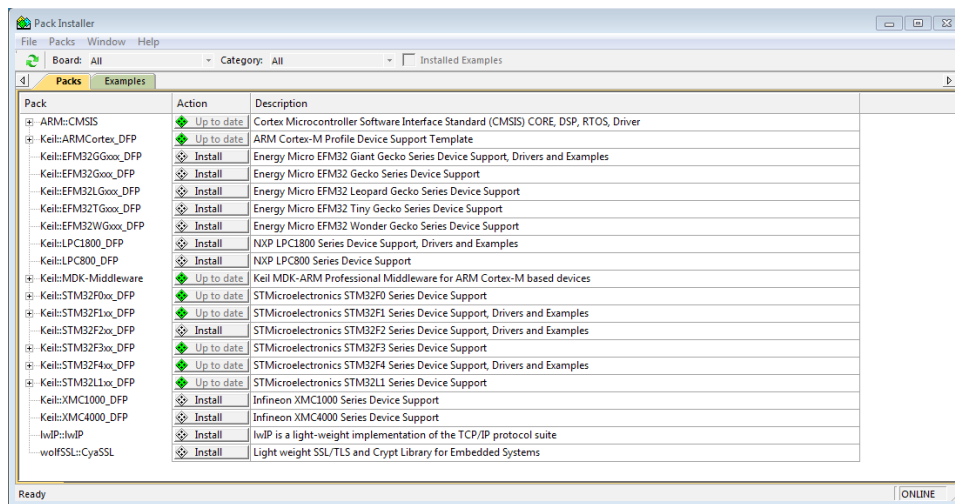
<https://launchpad.net/gcc-arm-embedded/>

(zum Zeitpunkt der Erstellung dieses Tutorials in der Version 4.7-2013-q3-update)

Beide Pakete werden nun installiert. Gestartet wird mit Keil MDK-Lite.



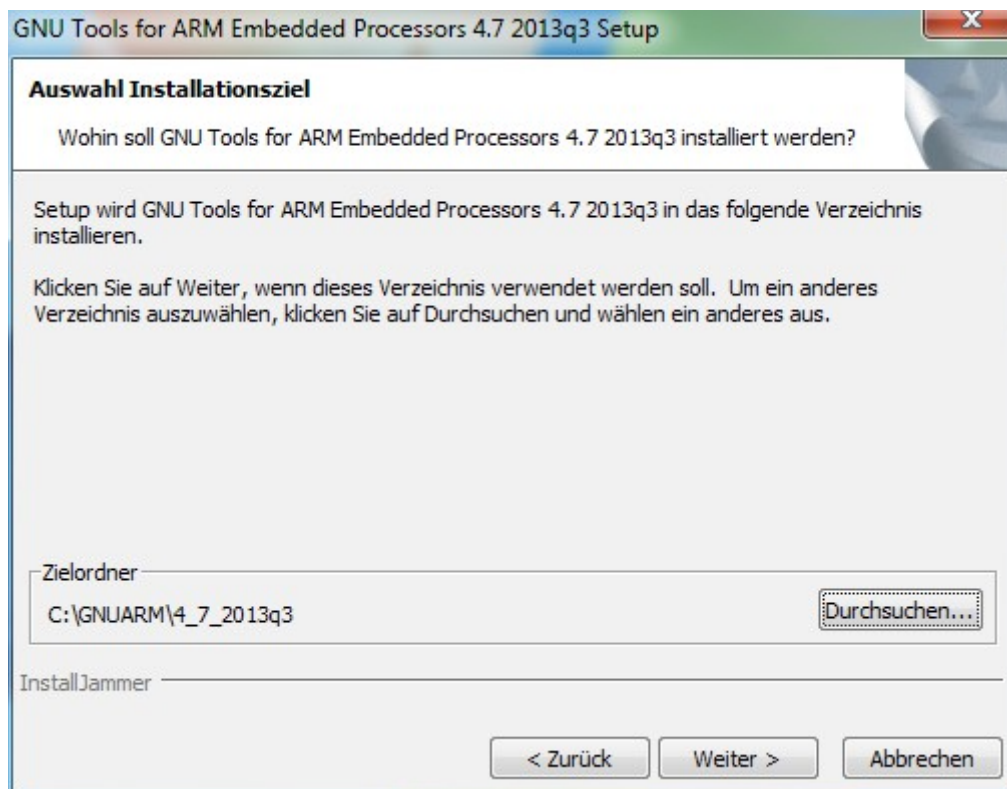
Die Installation ist problemlos und bedarf keiner weiteren Erläuterung. Am Ende der Installation startet die Toolchain mit dem Package Installer. Hier werden die Pakete für ARM:CMSIS sowie für die STM32-Serien installiert.



Jetzt sind die GNU Tools für ARM an der Reihe. Hier sollte darauf geachtet werden dass das Installationsverzeichnis im ROOT-Directory liegt. Dadurch wird die Pfadangabe vereinfacht um später keine Probleme mit rekursiven Pfadangaben zu haben.

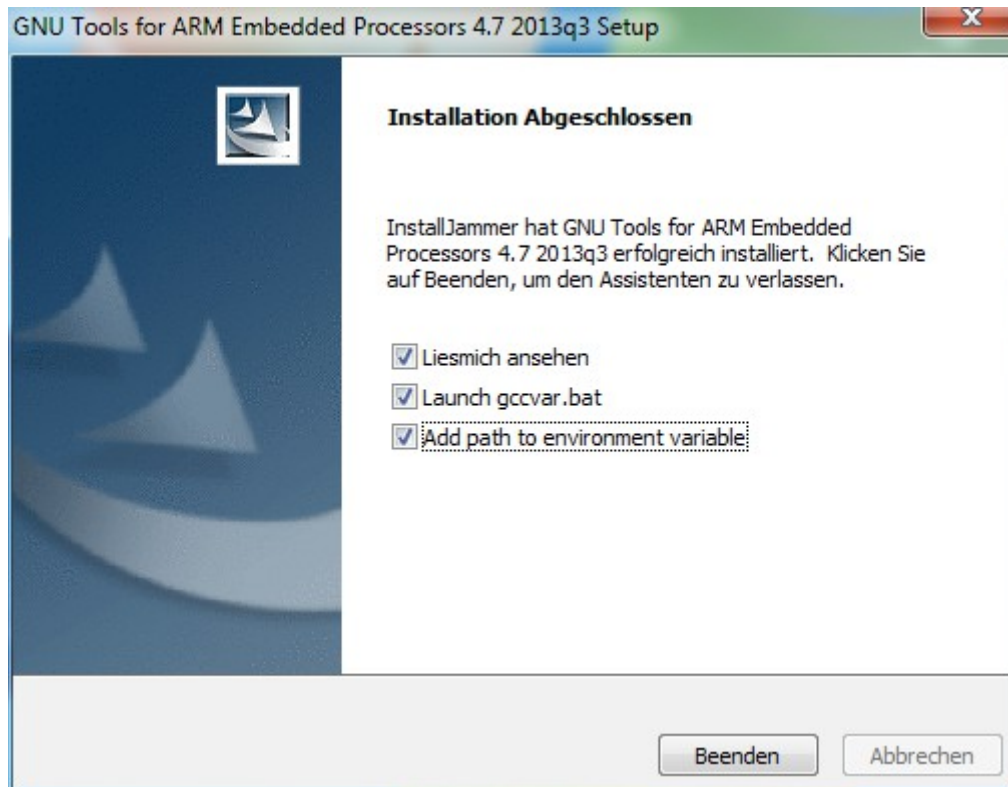
Für das Tutorial wird folgender Pfad verwendet:

C:\GNUARM\4\_7\_2013q3



Es empfiehlt sich die Versionsnummer beizubehalten. Dadurch können später weitere Compilerversionen installiert werden ohne die Settings bei bestehenden Projekten ändern zu müssen.

Am Ende der Installation wird noch die Checkbox für 'Add path to environment variable' gesetzt. Wer will kann sich auch die Readme.txt ansehen sowie den Compiler starten.



## 2. OPTIONAL - Support Libraries für den STM32

Wer auch eigene Projekte auf Basis der STM32-MCUs realisieren möchte sollte auch gleich die original STM Support Libraries laden und installieren. Für die Arbeit mit dem Baseflight- bzw. Harakiri-Projekt werden diese nicht benötigt!

Die STM32 Standard Peripheral Drivers werden von der STM32 Webseite geladen werden. Dieser Download ist z.B. unter den Design Resources zum STM32F103CB zu finden:

[http://www.st.com/web/catalog/mmc/FM141/SC1169/SS1031/LN1565/PF189782?s\\_searchtype=partnumber](http://www.st.com/web/catalog/mmc/FM141/SC1169/SS1031/LN1565/PF189782?s_searchtype=partnumber)

Man verwendet die

*STM32F10x standard peripheral library*

(STSW-STM32054, zum Zeitpunkt der Erstellung dieses Tutorials in der Version V3.5.0)

Z.B. wird nun im ROOT-Verzeichnis ein Arbeitsverzeichnis angelegt:

`C:\STM32_Workspace\`

Nun werden die unter Schritt 2 geladenen Standard Peripheral Library Files entpackt und folgende Dateien in dieses Arbeitsverzeichnis kopiert:

#### Das Verzeichnis

*STM32F10x\_StdPeriph\_Lib\_V3.5.0\Libraries\CMSIS\CM3\CoreSupport*  
nach  
*STM32\_Workspace\CMSIS*

#### Das Verzeichnis

*STM32F10x\_StdPeriph\_Lib\_V3.5.0\Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x*  
nach  
*STM32\_Workspace\STM32F10x*

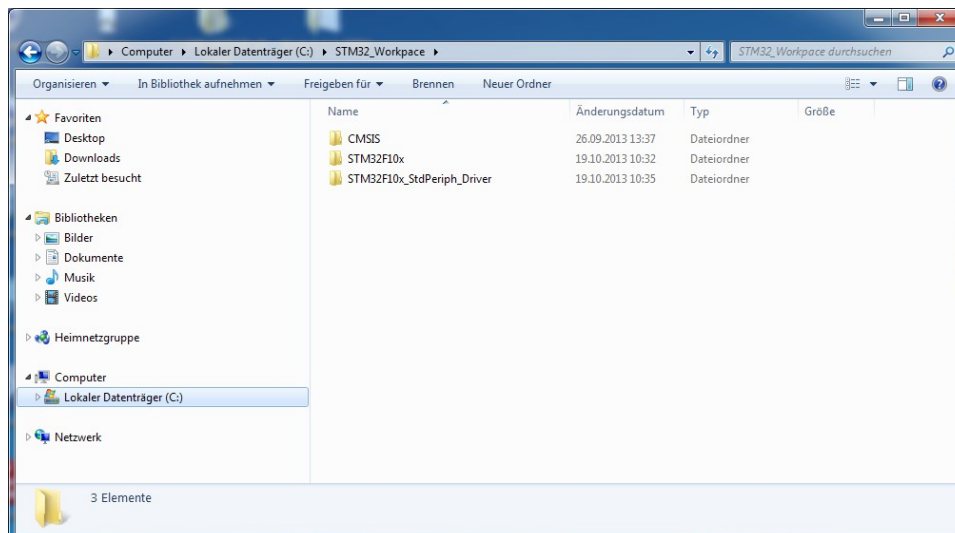
#### Das Verzeichnis

*STM32F10x\_StdPeriph\_Lib\_V3.5.0\Libraries\STM32F10x\_StdPeriph\_Driver*  
nach  
*STM32\_Workspace\STM32F10x\_StdPeriph\_Driver*

#### Das Verzeichnis

*STM32F10x\_StdPeriph\_Lib\_V3.5.0\Project\STM32F10x\_StdPeriph\_Template*  
nach  
*STM32\_Workspace\STM32F10x\_StdPeriph\_Driver*

Das Arbeitsverzeichnis sollte nun wie folgt aussehen:



### 3. Baseflight Sources auschecken

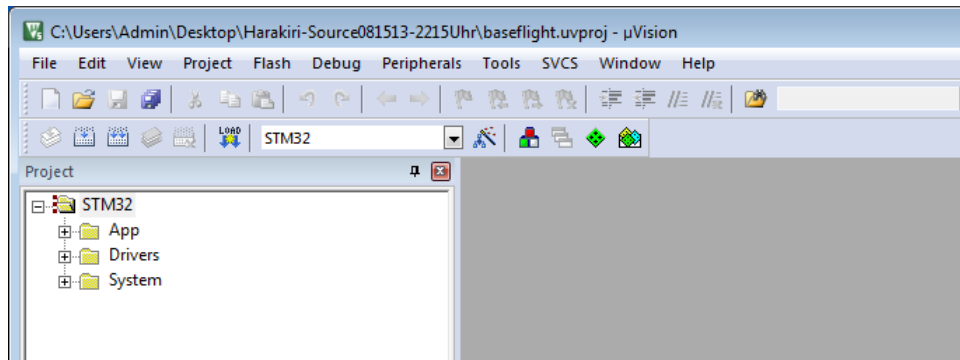
Da ich lieber auf dem Mac OS X und nicht unter Windows arbeite checke ich die Sourcen mit X-Code aus (Shame on me!). Hier gibt es jedoch verschiedenste Programme zur Verwendung unter Windows → eine entsprechende Ergänzung des Tutorials folgt. Die Sourcen in der Version r4xx liegen bei. So kann jeder die nachfolgenden Schritte durchspielen.

### 4. Öffnen des Projektfiles

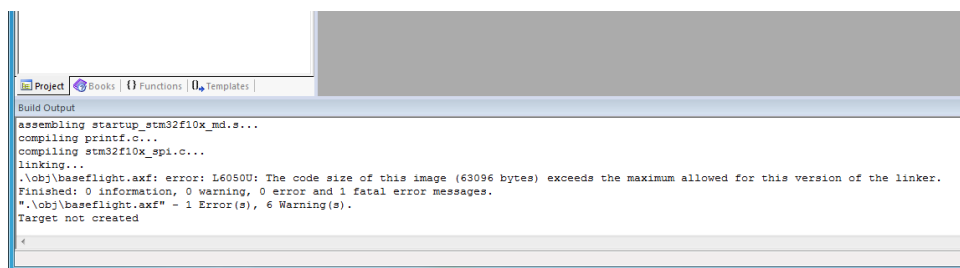
Timecop verwendet für die Baseflight Entwicklung ebenfalls die KEIL MDK IDE, weshalb er auch gleich ein passendes Projektfile mitliefert.

Mit einem Doppelklick auf die Datei *BASEFLIGHT.UVPROJ* wird dieses geöffnet. In der

µVision IDE sieht das dann wie folgt aus:

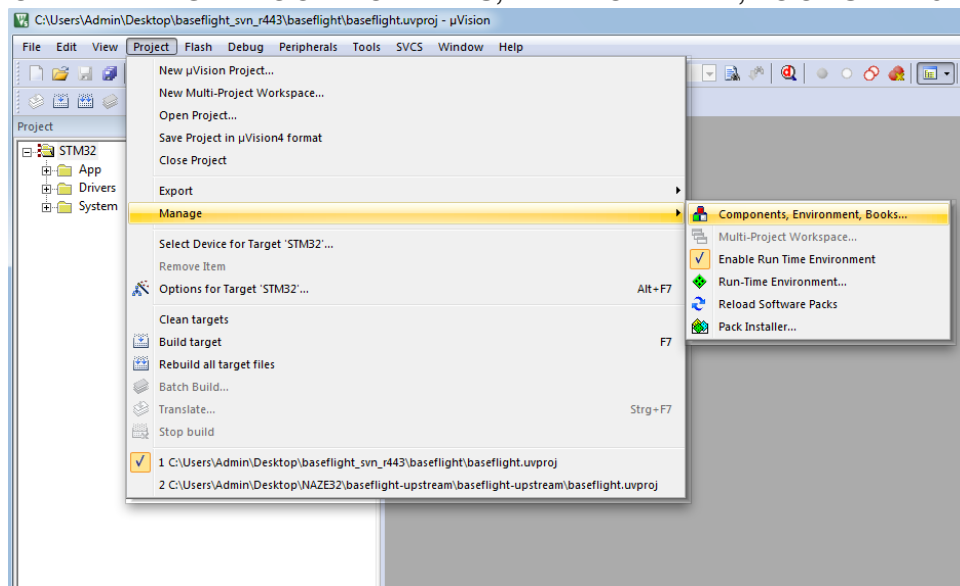


Wenn das Projekt schon mal offen ist kann ein BUILD (F7) nicht schaden! Da aktuell jedoch noch der auf 32kB limitierte Compiler verwendet wird endet dieser Versuch mit einer Fehlermeldung.

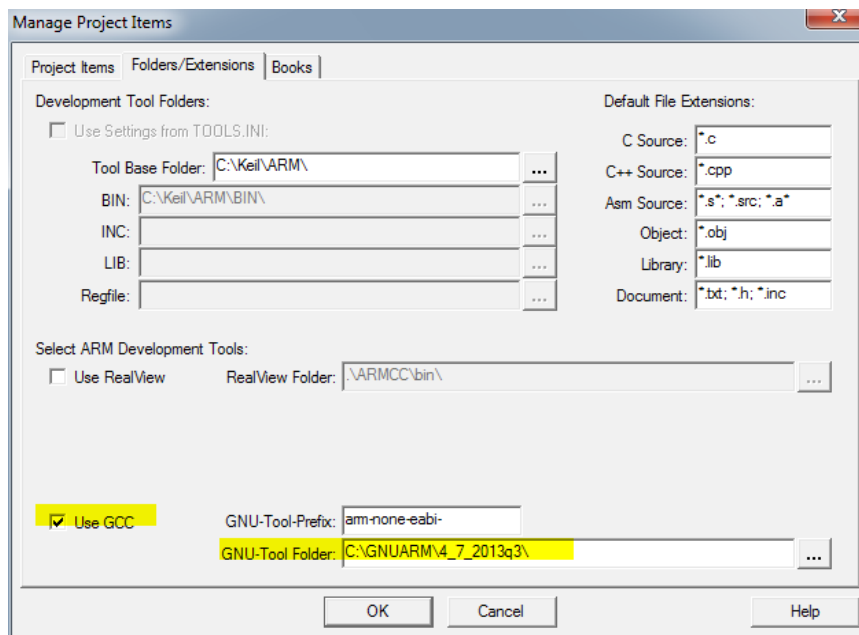


## 5. Aufheben der 32kB Limitierung

Auf **PROJECT-->MANAGE-->COMPONENTS,ENVIRONMENT,BOOKS...** klicken



Jetzt auf den **FOLDERS/EXTENSIONS** Reiter klicken und die **USE GCC** checkbox aktivieren.



Es erscheint eine Warnung dass alle Project-Settings verloren gehen. Diese wird mit YES bestätigt.

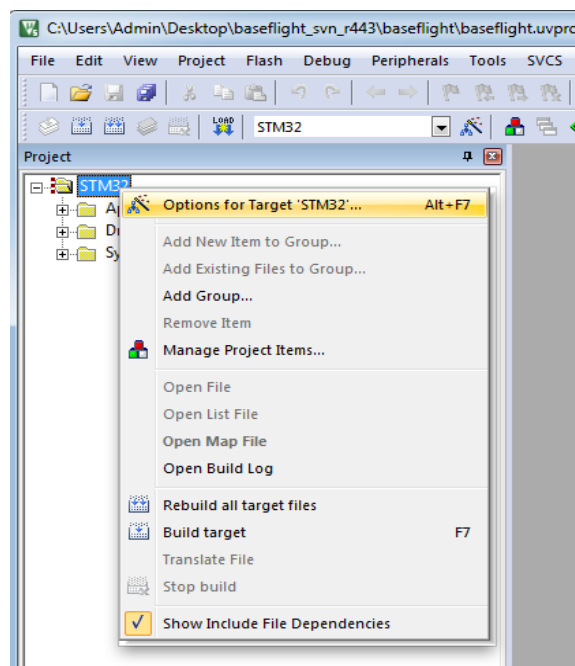
Jetzt wird der Pfad zur ARM GNU EMBEDDED Installation (aus 1.) eingetragen.

Für dieses Tutorial lautet dieser C:\GNUARM4\_7\_2013q3

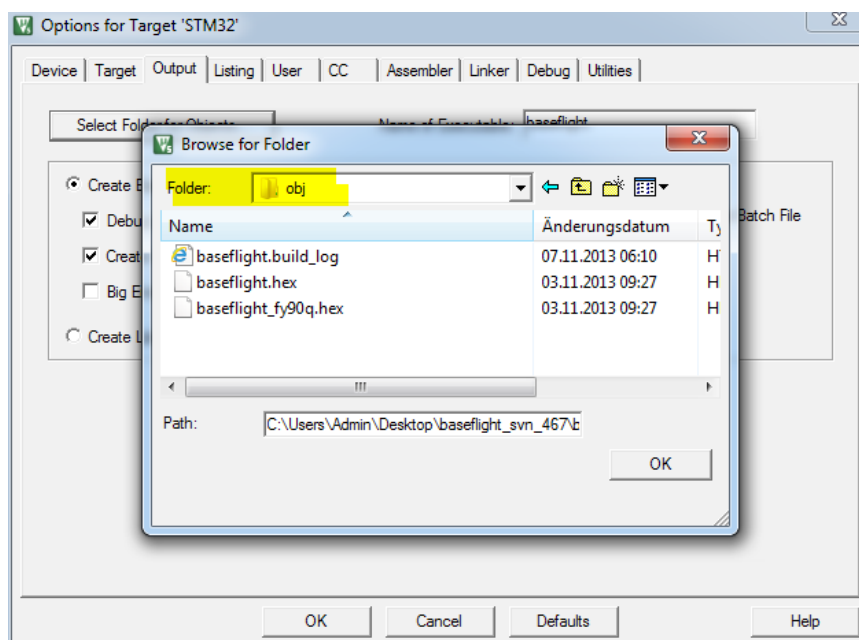
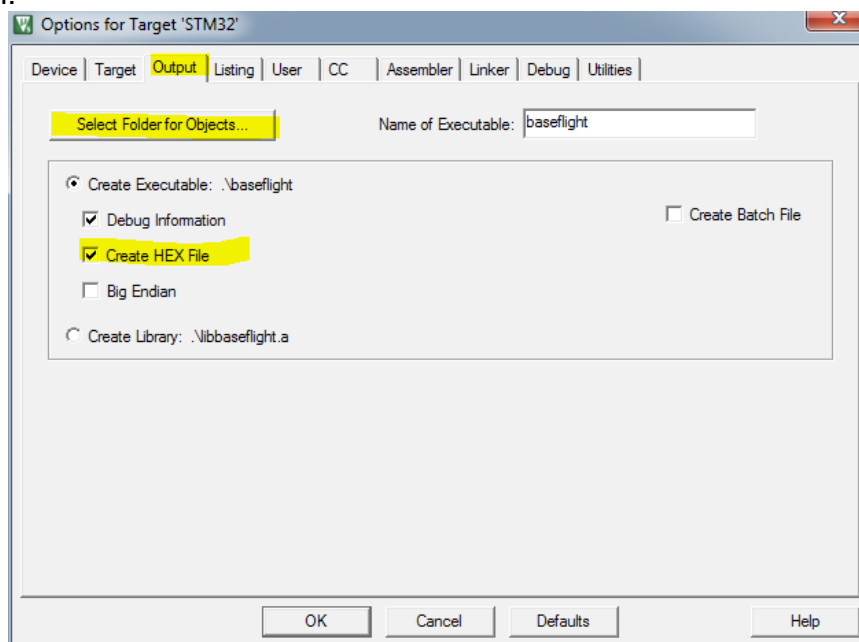
Mit OK bestätigen!

Im nächsten Schritt müssen nun die Project-Settings angepasst werden damit mit GCC compiliert werden kann.

Rechts auf *STM32* klicken und *OPTIONS FOR TARGET 'STM32'* auswählen (alternativ über ALT+F7)

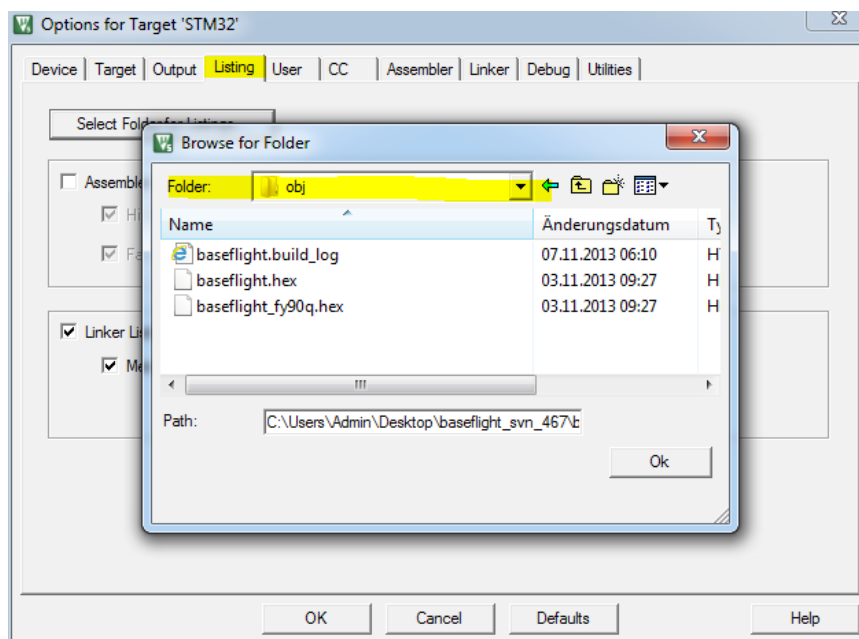
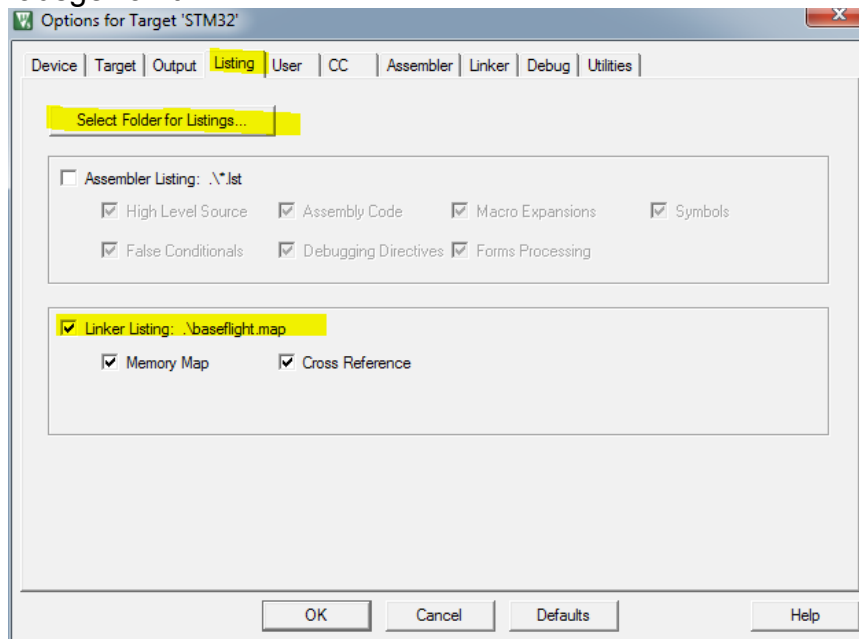


Auf den Reiter *OUTPUT* klicken, die Checkbox für die Erstellung des .HEX-Files anwählen, und über *SELECT FOLDER FOR OBJECTS* den Dateipfad auf 'obj' umstellen und bestätigen.





Im LISTING-Tab wird die Erstellung des .MAP-Files aktiviert und als Dateipfad ebenfalls der 'obj'-Folder ausgewählt.



Weiter geht es mit dem CC-Tab – den Compilereinstellungen. Hier werden folgende Änderungen gemacht:

Als PREPROCESSOR SYMBOLS wird

`STM32F10X_MD,USE_STDPERIPH_DRIVER`

eingetragen.

Der Optimization-Level wird von DEFAULT auf *LEVEL 2 (SIZE)* geändert.

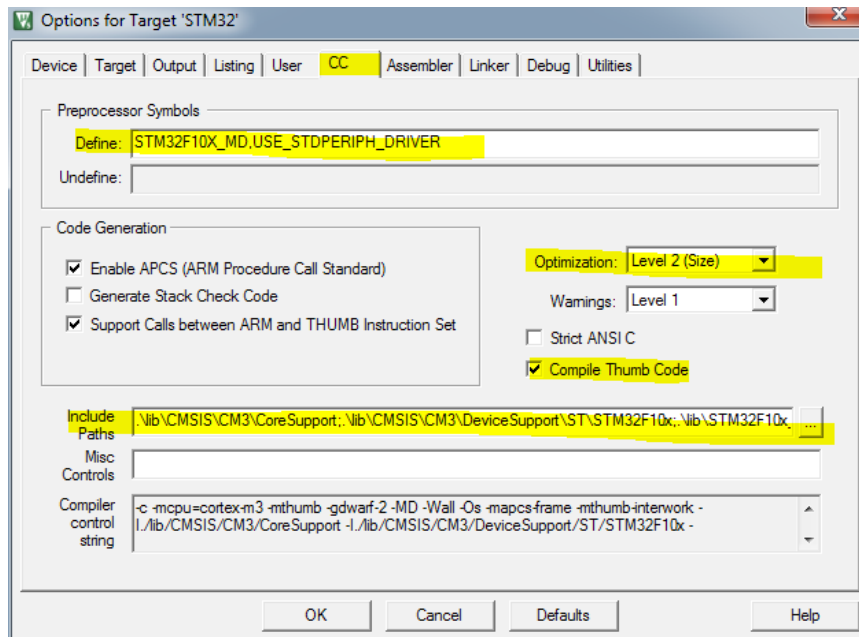
*COMPILE THUMB CODE* Checkbox aktivieren.

Bei den INCLUDE PATHS wird

`.\lib\CMSIS\CM3\CoreSupport;.\lib\CMSIS\CM3\DeviceSupport\ST\STM32F10x;.\lib\STM32F10x_StdPeriph_Driver\inc`

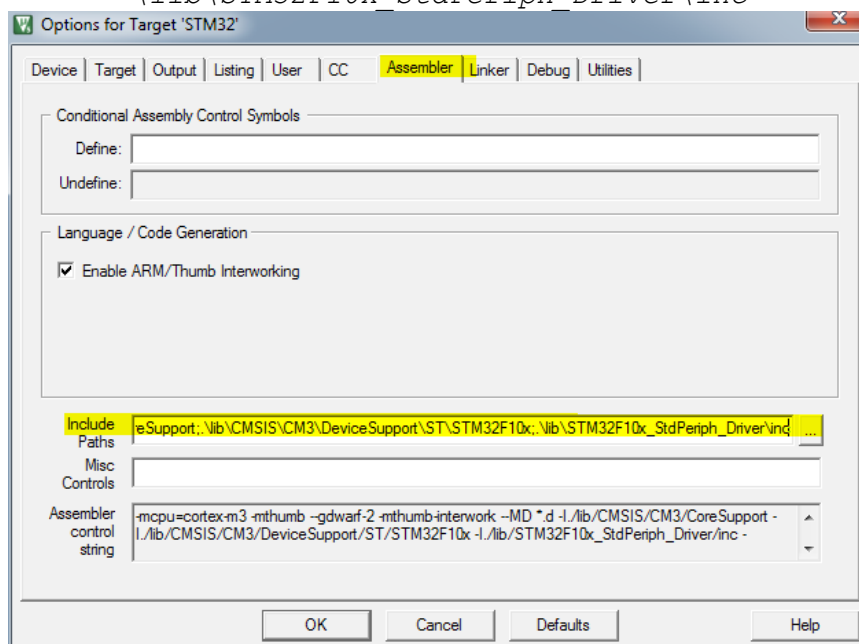
eingetragen.

Tip: Die Symbols und Paths am einfachsten per Copy&Paste aus diesem Dokument nehmen und eintragen!



Im ASSEMBLER-TAB werden ebenfalls die INCLUDE PATHS wie folgt eingetragen:

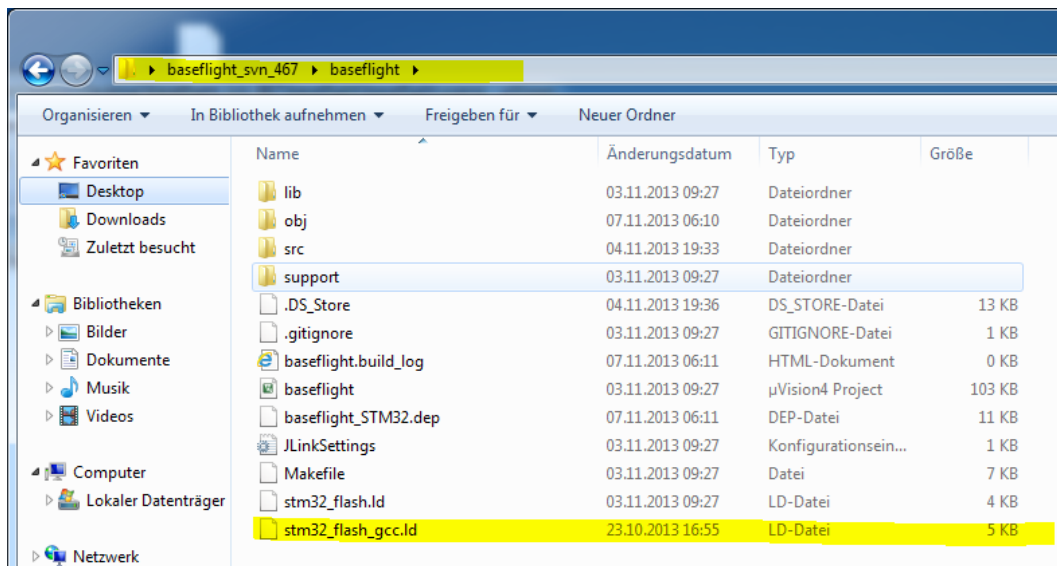
```
.\\lib\CMSIS\CMS3\CoreSupport;.\\lib\CMSIS\CMS3\DeviceSupport\ST\STM32F10x;.\\lib\STM32F10x_StdPeriph_Driver\inc
```



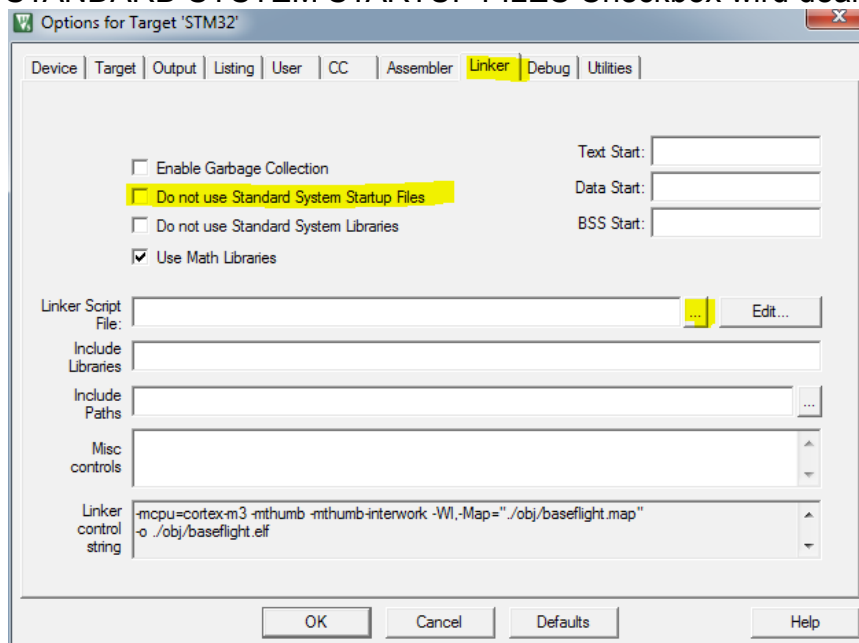
Jetzt folgt noch der LINKER-TAB.

Um hier die Einstellungen vornehmen zu können wird die beigelegte Datei *stm32\_flash\_gcc.ld* benötigt. Es handelt sich dabei um das Linker-Script-File für den ARM GCC Compiler.

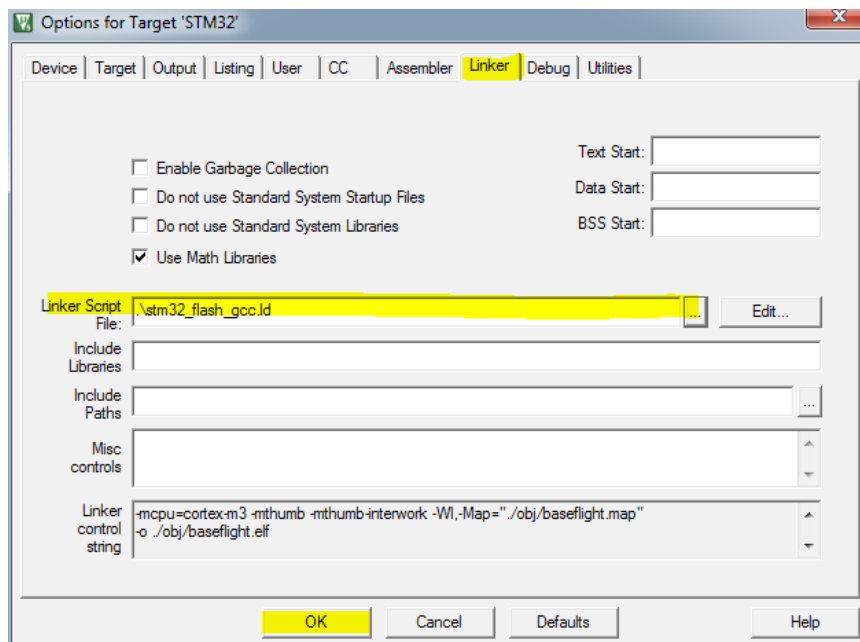
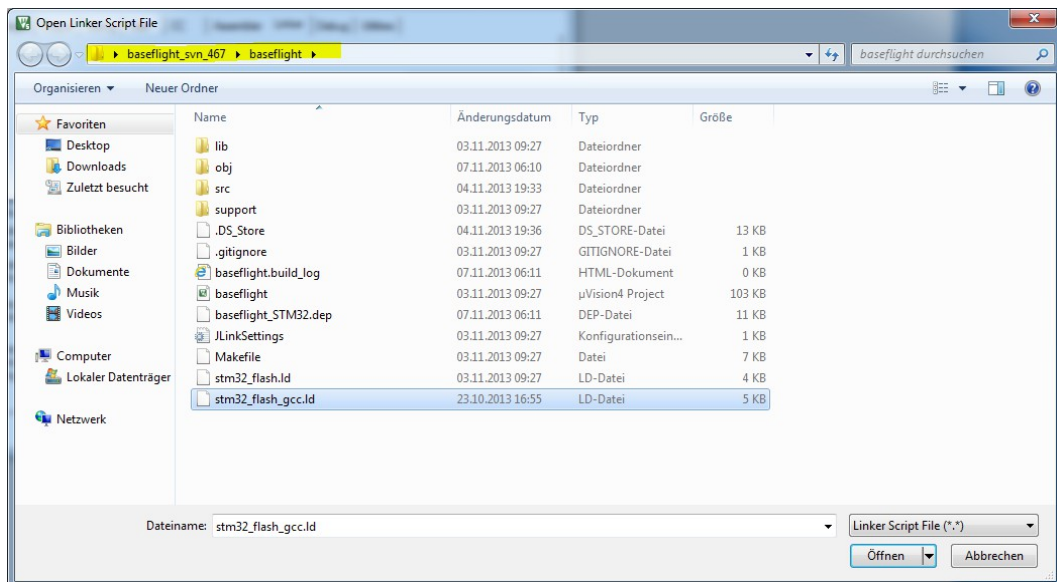
Die Datei wird einfach in das Baseflight-Verzeichnis kopiert.



Nun werden die folgenden Einstellungen gemacht:  
DO NOT USE STANDARD SYSTEM STARTUP FILES Checkbox wird deaktiviert.



Über den Browse-Button wird das vorher hinzugefügte Linker-Script-File ausgewählt.



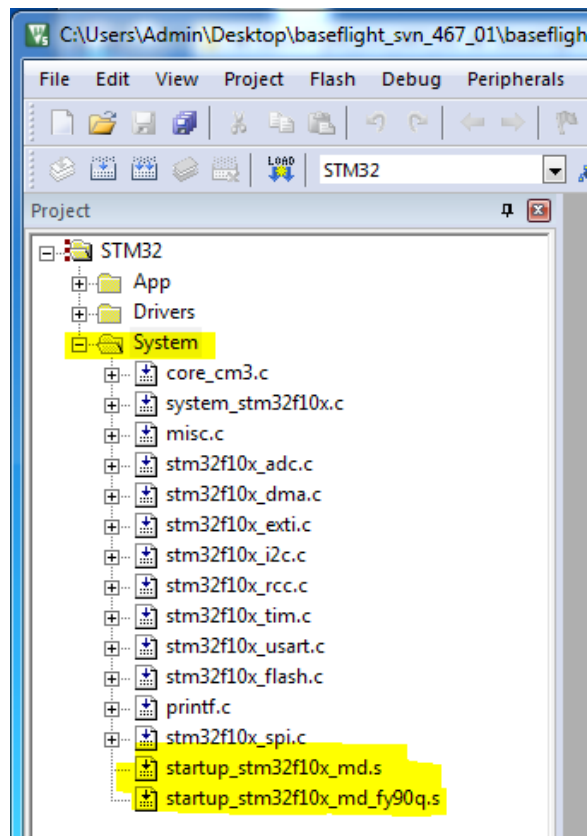
Durch einen Click auf OK werden alle gemachten Einstellungen übernommen.

## 6. Modifizieren des Projekts

Nachdem nun alle Einstellungen zur Verwendung des ARM GCC Compilers gemacht wurden muss das Projekt noch leicht angepasst werden um erfolgreich compiliert/gelinkt werden zu können.

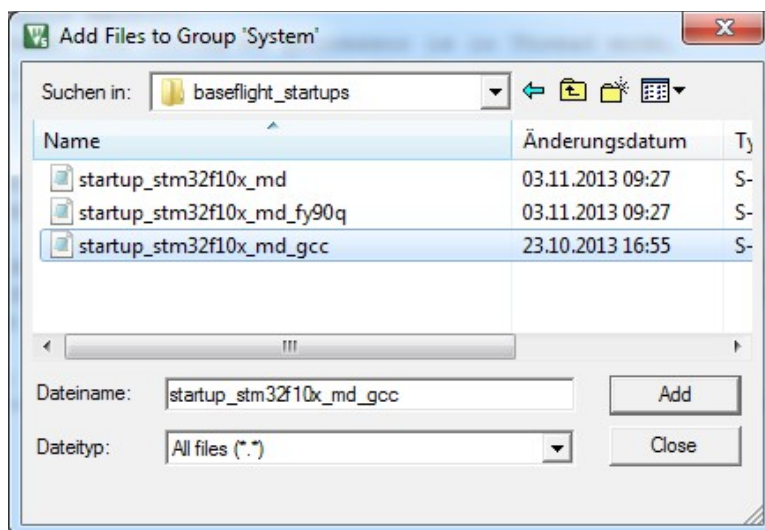
### 6.1 Startup-File

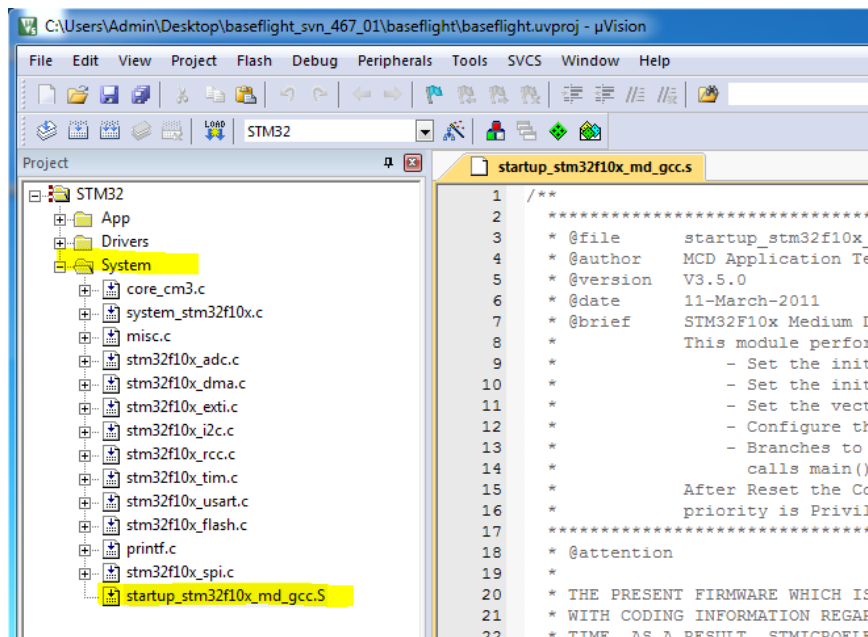
Da ein anderer Compiler verwendet werden soll muss ein passendes Startup-File verwendet werden. Dazu wird der SYSTEM-Folder aufgeklappt und die beiden vorhandenen Startup-Files `startup_stm32f10x_md.s` sowie `startup_stm32f10x_md_fy90q.s` werden vom Projekt entfernt. Die erfolgt durch Rechts-Click auf die entsprechende Datei → im Kontext-Menü dann auf 'REMOVE FILE <name>' klicken.



Das neue Startup-File ist die Datei `startup_stm32f10x_md_gcc.s` welche ebenfalls diesem 'How-To' beigefügt ist. Sie wird in den SRC->BASEFLIGHT\_STARTUPS Ordner des Baseflight Projektordners kopiert. Dort befindet sich bereits eine Datei mit diesem Namen welche durch die neue Version ersetzt werden muss!

Nun per Rechtsklick auf den SYSTEM-Folder das Kontext-Menü öffnen und auf ADD EXISTING FILES TO GROUP 'SYSTEM' ... klicken. In dem sich öffnenden Browserfenster navigiert man in den BASEFLIGHT\_STARTUPS Ordner und wählt dort das neue Startup-File aus (den Datei-Typ auf 'All files (\*.\*)' umstellen, sonst werden keine Dateien angezeigt).





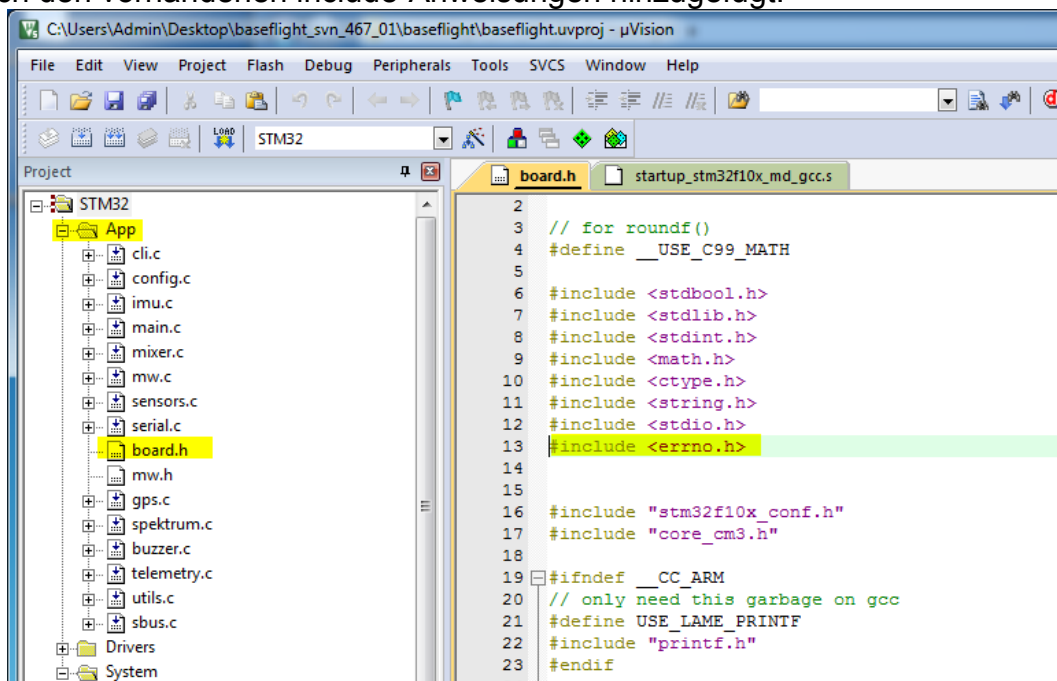
## 6.2 errno.h Header-File

Jetzt wird der APP-Folder aufgeklappt und das Header-File BOARD.H geöffnet. Hier muss ein weiteres Header-File hinzugefügt werden.

Der Befehl

```
#include <errno.h>
```

wird nach den vorhandenen include-Anweisungen hinzugefügt.



## 6.3 \_\_errno und \_exit Symbols

Damit das Projekt erfolgreich kompiliert und gelinkt werden kann werden zwei zusätzliche Symbole benötigt. Es handelt sich dabei um `__errno` und `_exit`.

Im APP-Folder wird die Datei MW.C geöffnet und der nachstehende Code wird hinzugefügt (z.B. als erste Funktion im Source, vor 'blinkLED(..)').

```

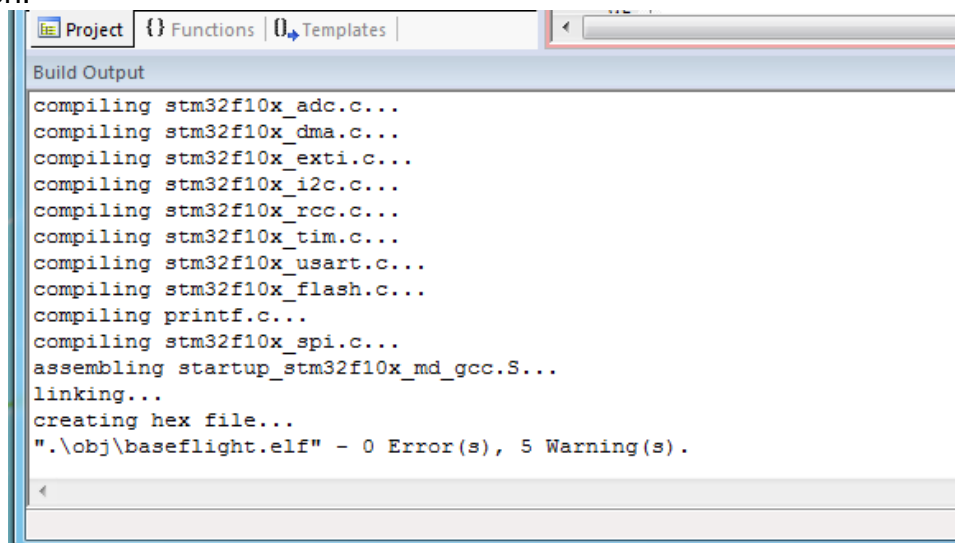
// aBUGSworstnightmare, changed on 10/20/2013
// __errno and _exit symbols were needed to compile/link the project
under Keil+GCC

// add '#include <errno.h>' to 'board.h'
// There is no default implementation of __errno. Keeping it out of the
library
// means that it's possible to customize it's behavior whitout rebuilding
the library.
// This is just one possible default implementation
int *__errno (void) {return &errno;}

// exit is defined in 'stdlib.h'
// exit (usually) terminates the calling process, returns to the startup
code
// and performs the appropriate cleanup process.
// This implementation is a simple trap and must be terminated by reset!
void _exit(int exit_code)
{
    while(1)
    {
        // Loop until reset
    }
}
// END OF NEW SYMBOLS

```

Jetzt auf REBUILD klicken und den Compiler/Linker seine Arbeit verrichten lassen.  
Das Projekt sollte ohne Fehler compiliert und gelinkt werden. Lediglich 5 Warnings werden  
ausgegeben.



TODO:

- Debugging