

# Tutorial 01: CanIf\_AppDemo

The English version is [here](#).

## Einleitung

Dieses Tutorial beschreibt den grundsätzlichen Umgang mit dem ExpansionPack. Beginnend mit dem Aufsetzen eines neuen Projektes in der STM32CubeIDE über das Laden des Packs und dessen Konfiguration, sowie dem Empfang und dem Senden von einfachen CAN Messages. Die Tutorialbeschreibung ist sehr Bilder lastig, man muss also relativ viel scrollen. Ich denke aber, dass die Bilder jeden helfen sich zu Recht zu finden.

## Das STM32CubeIDE Expansion Package herunter laden

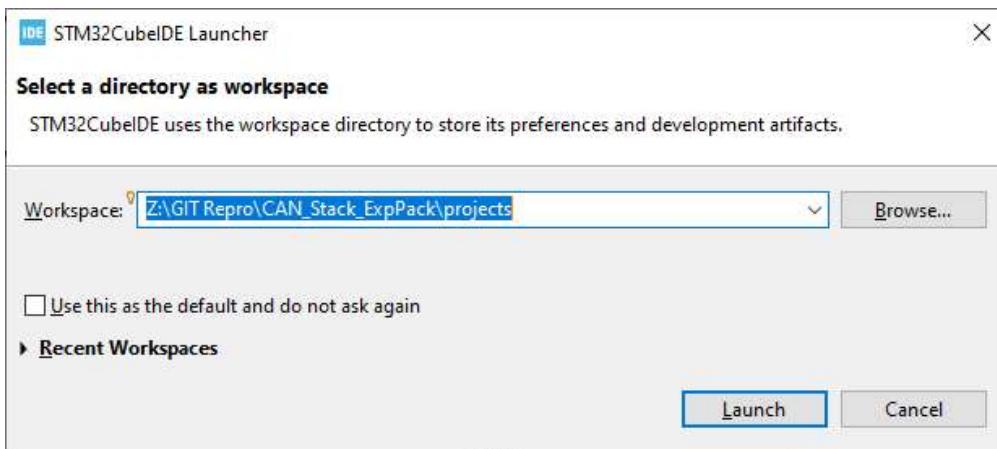
Es macht Sinn das ExpansionPack vor Beginn zu downloaden. Wenn man sich im Repository bewegt, hat man das ExpansionPack schon vorliegen. Nun auch auf [github](#)

## Starten eines neuen Projektes

Nach dem Start der STM32CubeIDE wird man nach dem *Workspace* gefragt in welchem man arbeiten möchte. Ich nutze für das gesamte Projekt um den *CAN Stack* durchgängig den Workspace wie im folgenden Bild.

### Table of Contents

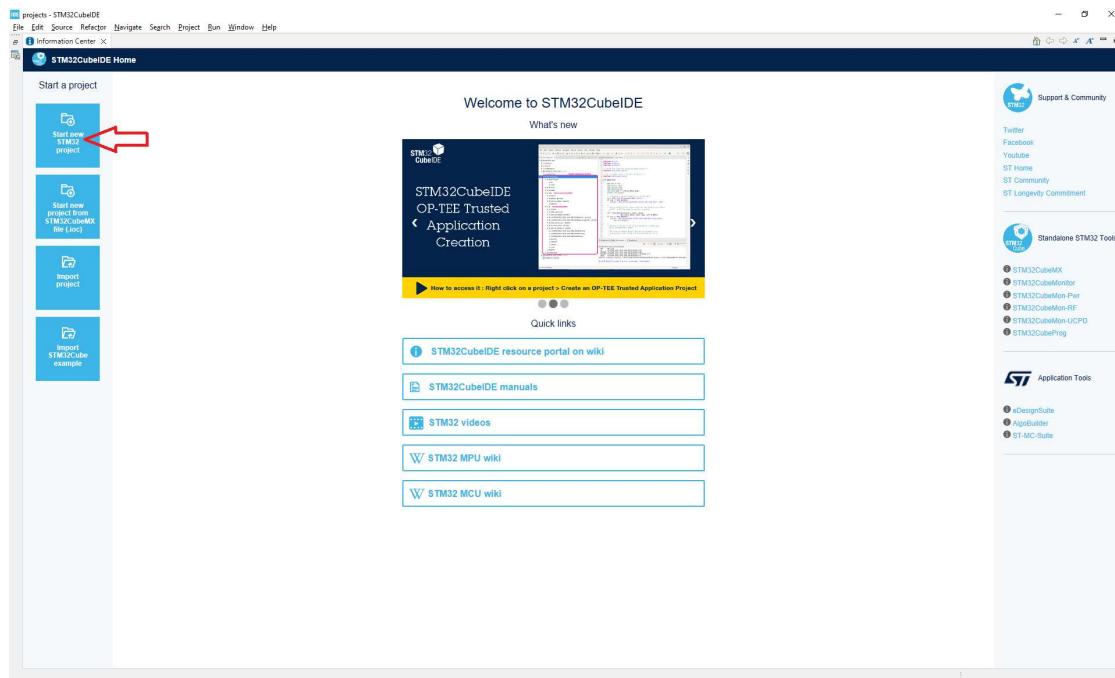
- Einleitung
- Das STM32CubeIDE Expansion Package herunter laden
- Starten eines neuen Projektes
  - Kurzer Überblick über den Aufbau der main.c Datei
  - Wie kommt das Expansion Package in die STM32CubeIDE
  - Manage Software Packs
  - Auswahl der Module
  - Konfiguration der Module
  - Hardware Settings
    - Bitrate Generator
    - bxCAN Master Setup / bxCAN Slave Setup
  - CAN Interface
    - Parameter Settings
    - RX/TX PDU Configuration
  - Den Code für das CanIf Modul generieren
  - Das ExpansionPack übernimmt Aufgaben aus der HAL
    - Diese Punkte muss ich noch ins Tutorial aufnehmen
  - Was ist bei einem "normalen" Öffnen der IDE anders als beim ersten Start



Select Workspace directory

Hierbei sei erwähnt, dass das "projects"-Verzeichnis eben auch jenem entspricht, welches als Spiegel zum Repository gepflegt wird.

Beim ersten Start der STM32CubeIDE wird man von dem "Information Center" begrüßt. Hier wählen wir natürlich für's Erste aus, dass wir ein neues Projekt starten möchten.



### Start new STM32 project

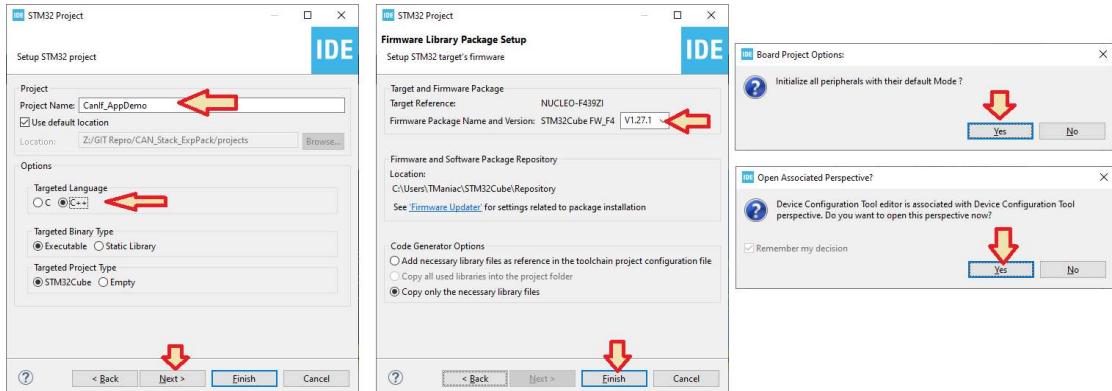
Später wird dieses "Information Center"-Fenster nicht unmittelbar beim Start angezeigt. Ich werde am **Ende des Tutorials** kurz auf den Neustart der STM32CubeIDE eingehen.

Ein STM32CubeIDE-Projekt (und auch ein STM32CubeMX-Projekt) baut natürlich auf einen bekannten STM32-Mikrocontroller auf. Ich habe ein Nucleo-F439ZI Board, auf welchen ich arbeite. Das NUCLEO-F439ZI hat, wie fast alle anderen NUCLEO-Boards auch, standardmäßig kein CAN. Man kommt aber über die "ARDUINO(R) UnoV3" Verbinder an die notwendigen Pins heran. So habe ich mir ein "Extension Board" mit zwei CAN Transceiver gebastelt. Wir werden später die verwendeten Port-Pins auswählen.

Board	Commercial Part No.	Status	Unit Price (US\$)	Mounted Device
NUCLEO-F413ZH	Nucleo-14			
NUCLEO-F429ZI	Nucleo-144	Active	23.0	<a href="#">STM32F429ZITB</a>
<b>NUCLEO-F439ZI</b>	<b>Nucleo-144</b>	Active	23.0	<a href="#">STM32F439ZITB</a>
NUCLEO-F446ZE	Nucleo-144	Active	19.0	<a href="#">STM32F446ZETB</a>

## Select Board or MCU

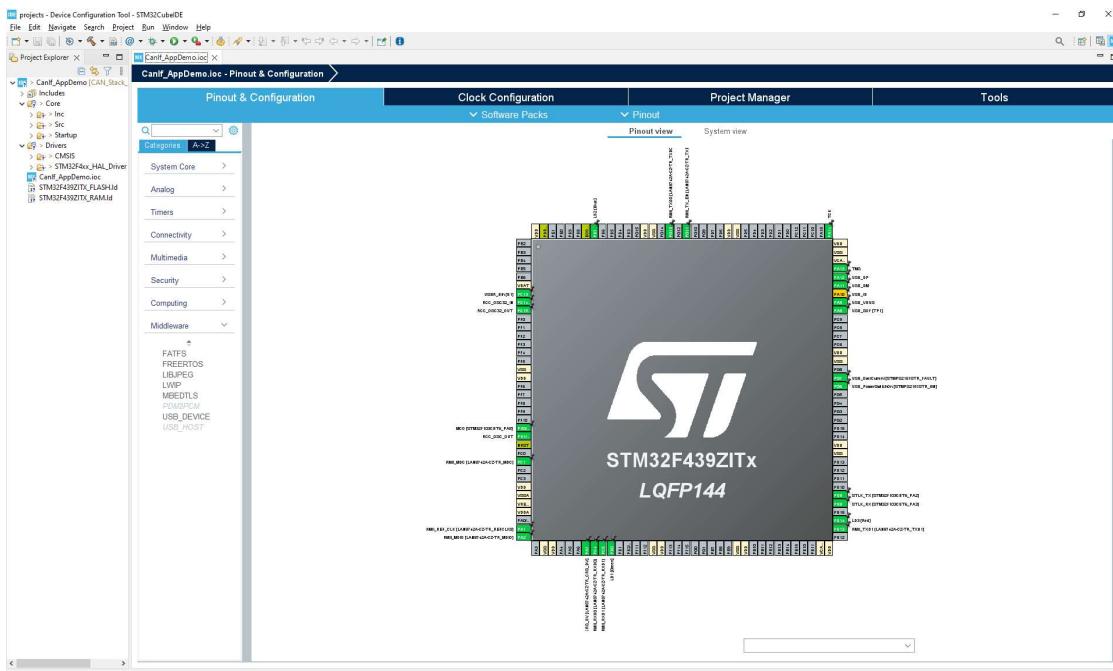
Die folgenden Dialoge sollten selbst erklärend sein. Wichtig ist, dass das ExpansionPack in C++ geschrieben ist. Die Voreinstellung ist hier leider immer C. Die Auswahl des verwendeten STM32Cube Firmware Package spielt aktuell noch keine Rolle, also sollte das aktuellste verwendet werden.



Steps to create project

Das Initialisieren der gesamten MCU-Peripherie ist nicht unbedingt erforderlich. Da es aber auch die Takterzeugung beinhaltet, ist es in den meisten Fällen sinnvoll.

Mit "Open Associated Perspective" wird in der STM32CubeIDE (entsprechend der Eclipse Umgebung) immer die Ansicht geöffnet, welche eben für den Arbeitsschritt am besten geeignet oder evtl sogar erforderlich ist. Hier wird im folgenden die Perspective geöffnet, welche die Konfiguration mit grafischer Unterstützung ermöglicht. Die Perspective sieht wie folgt aus:



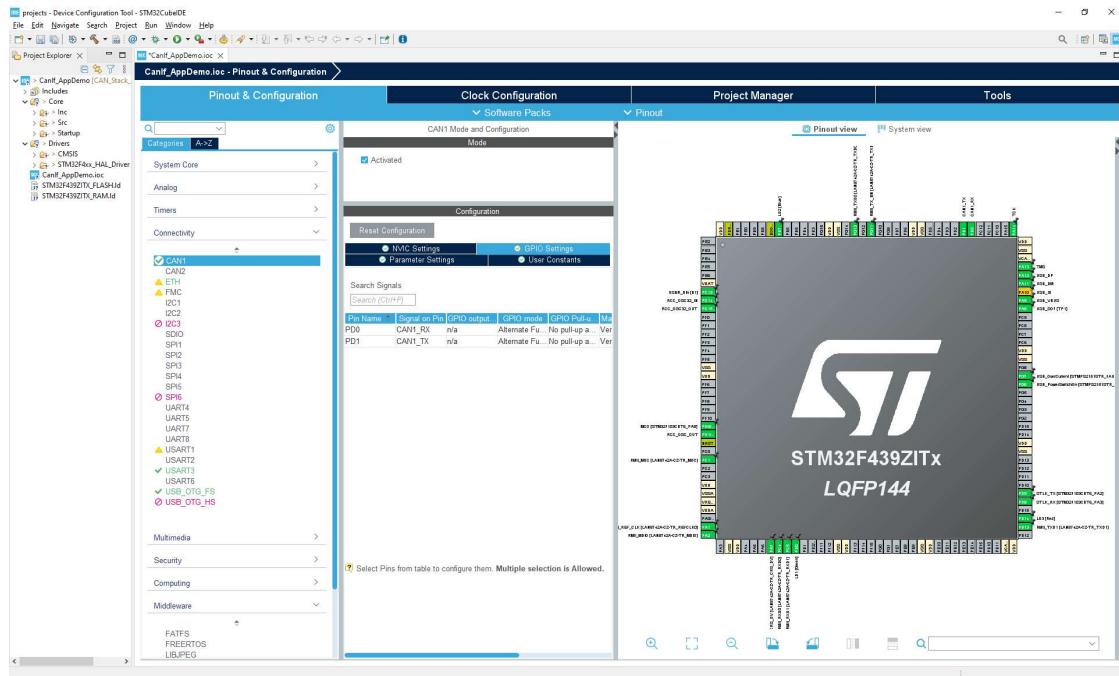
New project perspective

Ich möchte hier nicht explizit auf den Aufbau der STM32CubeIDE eingehen. Wichtig an dieser Stelle ist, dass wir auf der linken Seite den "normalen" Eclipse Project Browser haben. Der Großteil der Arbeitsfläche wird von der Darstellung der "\*.ioc"-Datei verwendet. Die Ansicht des Controllers ist auf den jeweiligen Footprint angepasst. Wie man auf dem Bild erkennen kann, hat der STM32F439ZIT auf meinem NUCLEO-F439ZI Board einen LQFP144 Footprint.

Die STM32CubeIDE benennt die neu erzeugte "\*.ioc"-Datei genauso wie das Projekt. Demnach haben wir jetzt eine "CanIf\_AppDemo.ioc".

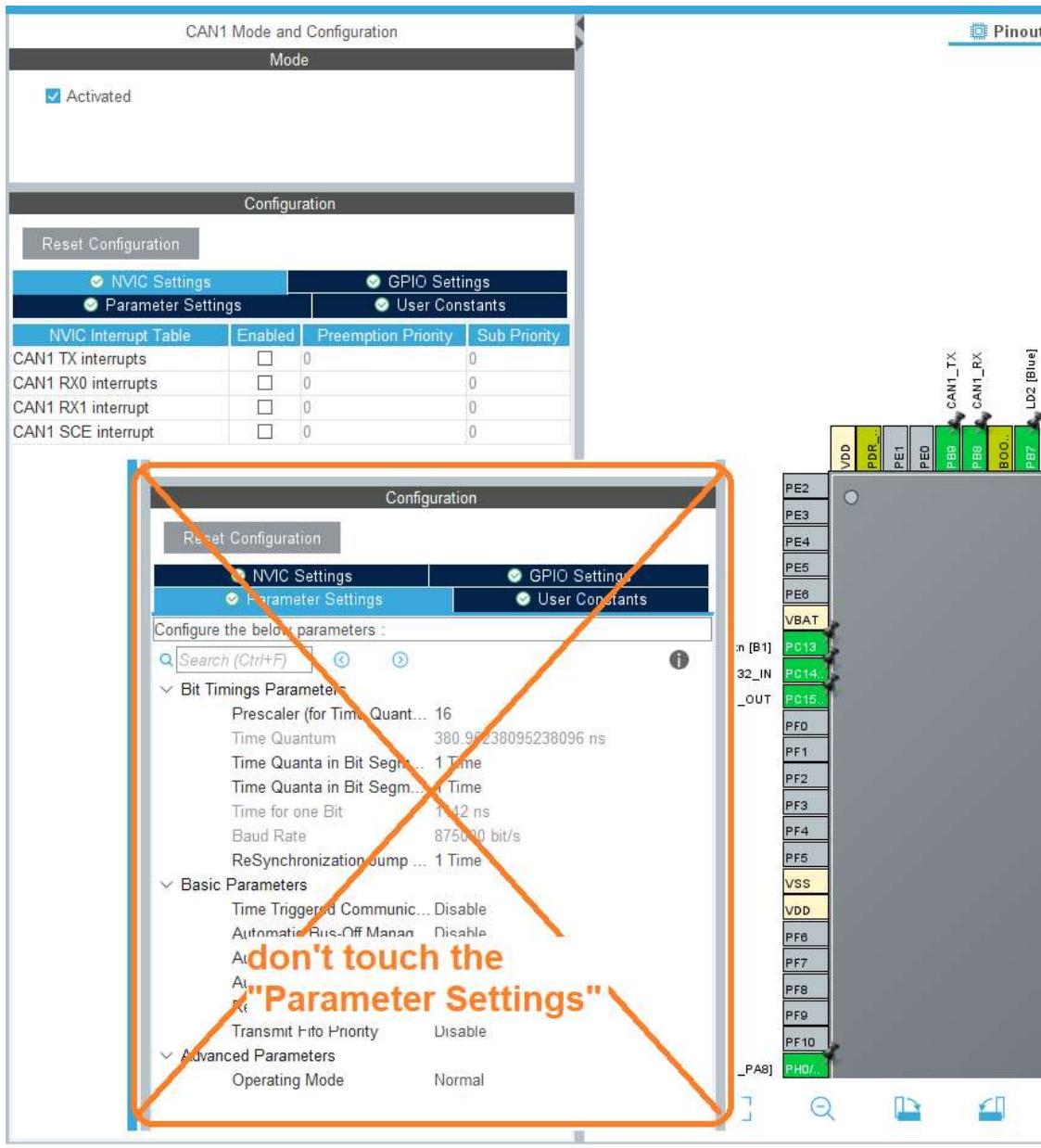
Da wir den CAN-Controller verwenden wollen, müssen wir diesen auch konfigurieren. CAN stellt eine "Connectivity" Peripherie dar. Also finden wir unter diesem Punkt auch die Module "CAN1" und "CAN2". Bei meinem NUCLEO-F439ZI gibt es standardmäßig keine Vorbereitung für CAN, also sind diese Module deaktiviert. Wählt man "CAN1" (oder auch "CAN2") aus, so erscheint eine zusätzliche Spalte mit der Bezeichnung "CAN1 Mode and Configuration". Die Darstellung mit diesen drei Spalten ist jener Aufbau welcher für die Konfiguration der STM32Cube-Elemente (welche intern immer als Pack gehandhabt werden) verwendet wird.

Im Bereich "Mode" aktiviert man die jeweiligen Module. Wir aktivieren nun das CAN1 Module durch setzen des Hakens "Activated".



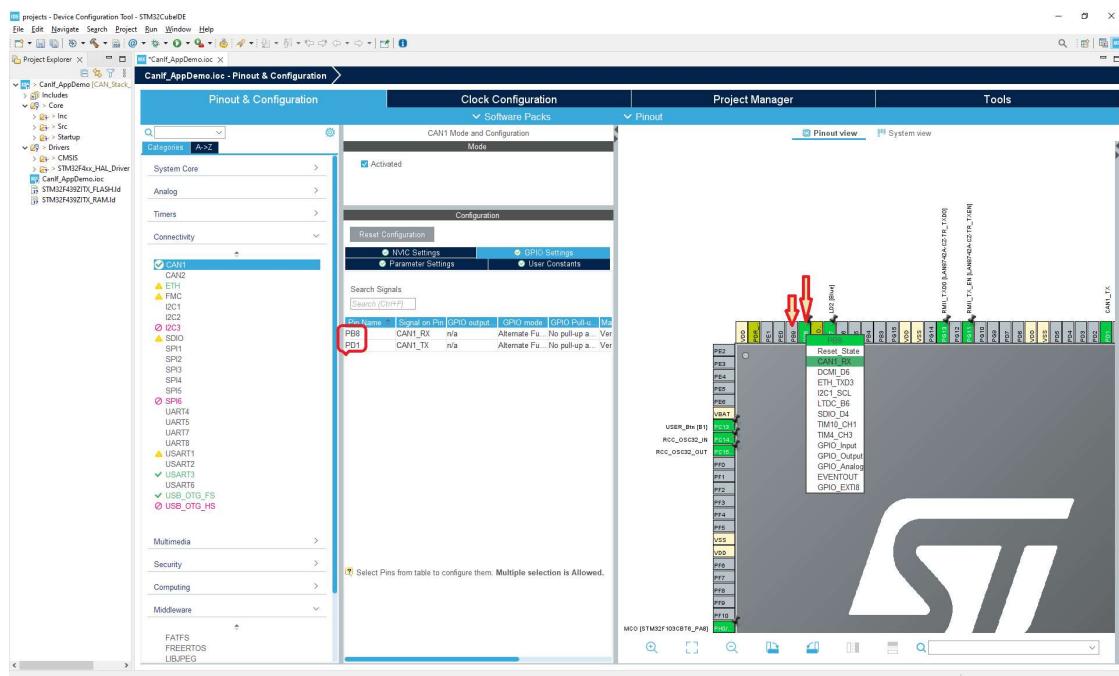
### CAN1 activated

Jetzt werden im Bereich "Configuration" vier Tabs angezeigt. Der Tab "User Constants" spielt für uns keine Rolle. Dieser Tab wird in jeden Modul angezeigt. "Parameter Settings" beinhaltet die Startup Parameter des CAN-Controllers und NVIC Settings die Einstellung der verwendeten Interrupt-Callbacks. Das *ExpansionPack CAN Stack* übernimmt auch die Konfiguration der Startup Parameter, also lassen wir die "Parameter Settings" hier wie sie sind. Die Konfiguration der Interrupt-Callbacks werden wir später aufgreifen.



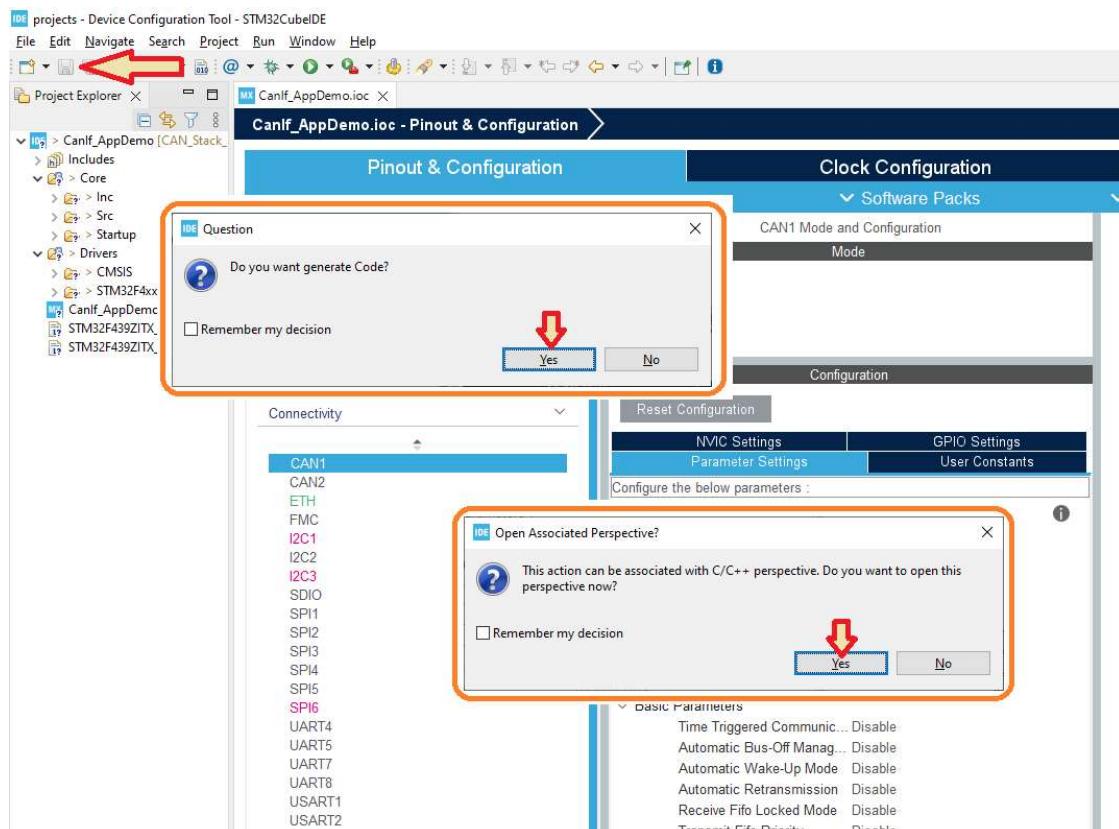
### Parameter and NVIC Settings

Im Tab "GPIO Settings" werden die Port-Pins eingestellt, bzw angezeigt. Die Funktion der Pins lassen sich durch anklicken des Pins in der µC-Darstellung direkt auswählen. Bei meinem CAN-Transceiver-Board sind CAN1\_RX an PB8 und CAN1\_TX an PB9 angebunden. So nehme ich diese Einstellung hier auch vor.



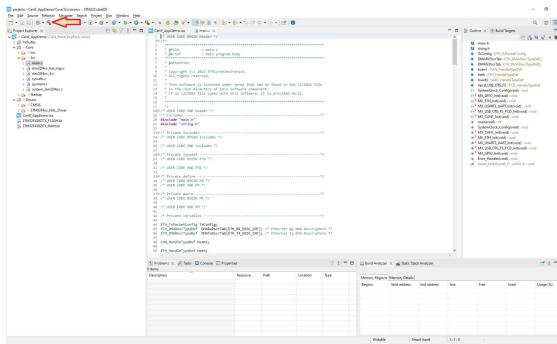
### GPIO Settings

Nun wird es Zeit die Konfiguration das erste Mal zu speichern. Beim Klick auf die "Diskette" speichert STM32CubeIDE (entsprechend der Eclipse-Umgebung) die jeweils aktive Datei. Da wir gerade die "CanIf\_AppDemo.ioc" bearbeitet haben, wird diese gespeichert. Beim speichern einer "\*.ioc-Datei" fragt die STM32CubeIDE standardmäßig ob der Source Code generiert werden soll. (Das sollte man später im Hinterkopf behalten, wenn man testweise Code ändert.) Danach wird man gefragt ob man zur C/C++ Perspective wechselt möchte.



### Save CanIf\_AppDemo.ioc and generate code

Der erzeugte Code sollte build-fähig sein. Das können wir durch den Klick auf den "Hammer" probieren



### Build generated code

Im Project Browser sind nun auch die neuen Dateien sichtbar. Die C/C++ Perspective bietet uns eine "Outline"-Übersicht über die aktive Datei. Im unteren Bereich der IDE findet man weitere hilfreiche Fenster. Details sollte man sich selber in diversen Eclipse Tutorials erarbeiten.

## Kurzer Überblick über den Aufbau der main.c Datei

Ich möchte hier an Hand der main.c Datei ganz kurz auf ein paar Besonderheiten im Umgang mit der Code Generierung in der STM32CubeIDE eingehen. Vielleicht verschiebe ich das Kapitel mal und mache es etwas ausführlicher. Dann kann man auch Besonderheiten bei der Bearbeitung der \*.ftl Templates eingehen.

```

main.c X MK_CanIf_AppDemo.ioc
1 /* USER CODE BEGIN Header */
2 /**
3  * @file           : main.c
4  * @brief          : Main program body
5  *
6  * @attention
7  *
8  * Copyright (c) 2022 STMicroelectronics.
9  * All rights reserved.
10 *
11 * This software is licensed under terms that can be found in the LICENSE file
12 * in the root directory of this software component.
13 * If no LICENSE file comes with this software, it is provided AS-IS.
14 *
15 */
16
17 /**
18 * USER CODE END Header */
19 /* Includes -----*/
20 #include "main.h"
21
22 /* Private includes -----*/
23 /* USER CODE BEGIN Includes */
24
25 /* USER CODE END Includes */
26
27 /* Private typedef -----*/
28 /* USER CODE BEGIN PTD */
29
30 /* USER CODE END PTD */
31

```

### how to work on main.c

STM nutzt eine Doxygen taugliche Formatierung, so dass man auch die Lizenzinformationen in der Source Code Doku einbinden kann. Der Dateikopf ist in einem

```

USER CODE BEGIN Header
[...]
USER CODE END Header

```

Block eingeordnet. Alles was in solch einem Block steht, wird bei einer erneuten Generierung nicht geändert. Diese Blöcke werden aber durch den Generator vorgegeben. Es macht also keinen Sinn selber solche Blöcke zu ergänzen. Man sieht hier zum Beispiel die "User Code" Blöcke für Includes und typedefs.

Es sei aber erwähnt, das der Code "verschwindet" wenn die Datei bei der Code Generation entfernt wird, zum Beispiel dann wenn man ein Modul deaktiviert.

```

40 /* Define handles */
41 /* @brief CAN handle structure definition */
42 CAN_HandleTypeDef hcan1;
43
44 UART_HandleTypeDef huart3;
45
46 /* USER CODE BEGIN PV */
47
48 /* USER CODE END PV */
49
50 /**
51 * @brief CAN handle Structure definition
52 */
53 #typedef struct _CAN_HandleTypeDef
54 {
55     /* CAN_TypeDef           *Instance;          /*!< Register base address */
56     /* CAN_InitTypeDef       Init;              /*!< CAN required parameters */
57     /* _IO HAL_CAN_StateTypeDef State;           /*!< CAN communication state */
58     /* _IO uint32_t          ErrorCode;         /*!< CAN Error code.
59     /* This parameter can be a value of @ref
60     /* @ref hal_can_error_t
61 }
62
63
64 /**
65 * @brief The application entry point.
66 */
67
68 /**
69 * @brief Application entry point.
70 */
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108

```

## Handles to work with the hardware

STM nutzt "Handle"-Strukturen um mit der jeweiligen Peripherie (bis auf GPIO) zu arbeiten. Standardmäßig wird das Handle-Objekt im Kontext der `main.c` erzeugt. Dieses enthält neben dem Pointer zu den Registern auch eine Struktur zur Initialisierung.

```

50 /* Private function prototypes -----*/
51 void SystemClock_Config(void);
52 static void MX_GPIO_Init(void);
53 static void MX_USART3_UART_Init(void);
54 static void MX_CAN1_Init(void);
55 /* USER CODE BEGIN PFP */
56
57 /* USER CODE END PFP */
58
59 /* Private user code -----*/
60 /* USER CODE BEGIN 0 */
61
62 /* USER CODE END 0 */
63

```

## Function declarations

Diese `MX_(HW-Module)_Init()` Funktionen werden für jedes Modul generiert. Logisch werden diese Funktionen weiter unten dann mit den Parametern aus dem `*.ioc` File gefüttert.

Darüber hinaus haben wir hier auch Platz um eigene (einfache) Funktionen zu deklarieren ("User Code \* PFP") und auch zu definieren ("User Code \* 0"). Wie gesagt, was man in solch einen Block hinein schreibt, bleibt auch bei einer Re-Generation erhalten.

```

63
64 /**
65 * @brief The application entry point.
66 */
67
68 int main(void)
69 {
70     /* USER CODE BEGIN 1 */
71
72     /* USER CODE END 1 */
73
74     /* MCU Configuration-----*/
75
76     /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
77     HAL_Init();
78
79     /* USER CODE BEGIN Init */
80
81     /* USER CODE END Init */
82
83     /* Configure the system clock */
84     SystemClock_Config();
85
86     /* USER CODE BEGIN SysInit */
87
88     /* USER CODE END SysInit */
89
90     /* Initialize all configured peripherals */
91     MX_GPIO_Init();
92     MX_USART3_UART_Init();
93     MX_CAN1_Init();
94
95     /* USER CODE BEGIN 2 */
96
97     /* Infinite loop */
98
99     /* USER CODE BEGIN WHILE */
100    {
101        /* USER CODE END WHILE */
102
103        /* USER CODE BEGIN 3 */
104
105    }
106
107 }
108

```

## main() Function

Die eigentliche `main()` Funktion ruft der Reihe nach die `Init()`-Funktionen auf und geht anschließend in die für Mikrocontroller typische `while(1)` Schleife.

Die Reihenfolge der `MX_(HW-Module)_Init()` Funktionen wird leider vom Code Generator vorgegeben. So lange man ausschließlich die HAL nutzt, spielt das keine Rolle. Da wir in unserem Expansion Pack aber HAL-

Aufgaben übernehmen wollen, braucht es da einen gezielten Ansatz.

Nach der main() kommen dann die einzelnen MX\_(HW-Module)\_Init() Funktionen. Als Beispiel hier nur kurz MX\_CAN1\_Init() Funktionen

```
154@ /**
155 * @brief CAN1 Initialization Function
156 * @param None
157 * @retval None
158 */
159@ static void MX_CAN1_Init(void)
160 {
161     /* USER CODE BEGIN CAN1_Init 0 */
162
163     /* USER CODE END CAN1_Init 0 */
164
165     /* USER CODE BEGIN CAN1_Init 1 */
166
167     /* USER CODE END CAN1_Init 1 */
168     hcan1.Instance = CAN1;
169     hcan1.Init.Prescaler = 16;
170     hcan1.Init.Mode = CAN_MODE_NORMAL;
171     hcan1.Init.SyncJumpWidth = CAN_SJW_1TQ;
172     hcan1.Init.TimeSeg1 = CAN_BS1_1TQ;
173     hcan1.Init.TimeSeg2 = CAN_BS2_1TQ;
174     hcan1.Init.TimeTriggeredMode = DISABLE;
175     hcan1.Init.AutoBusOff = DISABLE;
176     hcan1.Init.AutoWakeUp = DISABLE;
177     hcan1.Init.AutoRetransmission = DISABLE;
178     hcan1.Init.ReceiveFifoLocked = DISABLE;
179     hcan1.Init.TransmitFifoPriority = DISABLE;
180
181     if (HAL_CAN_Init(&hcan1) != HAL_OK)
182     {
183         Error_Handler();
184     }
185     /* USER CODE BEGIN CAN1_Init 2 */
186
187     /* USER CODE END CAN1_Init 2 */
188 }
189 }
```

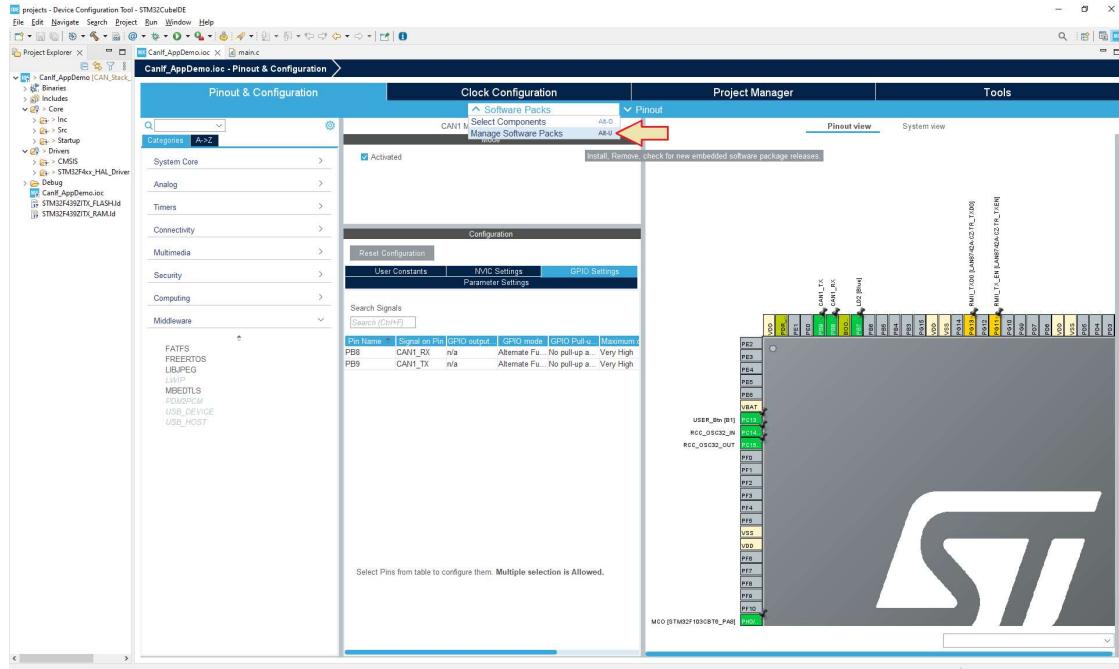
### Init the CAN1 Controller

Grundsätzlich wird nur die Init-Struktur gefüllt und dann die eigentliche HAL\_CAN\_Init() aufgerufen. Nur wo werden die Pins zu geordnet. Dazu komme ich dann wenn es so weit ist.

## Wie kommt das Expansion Package in die STM32CubeIDE

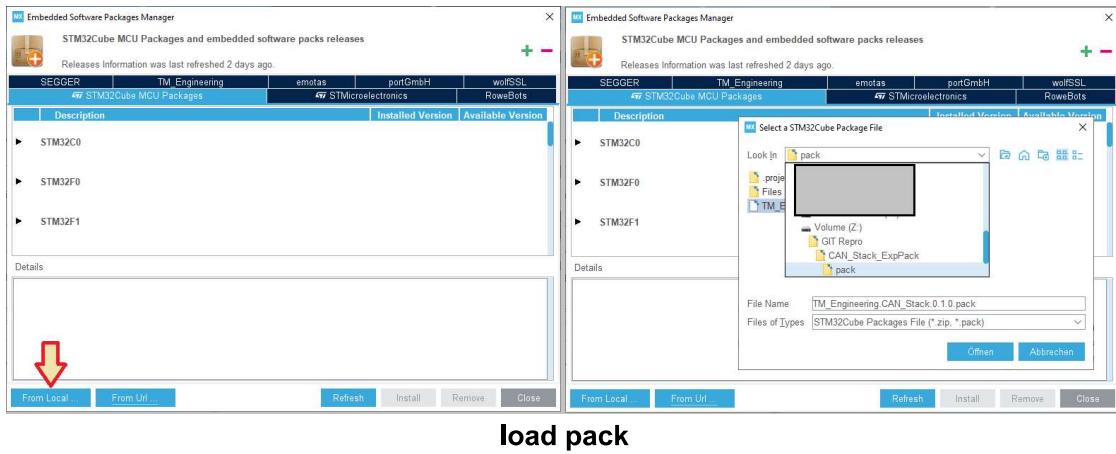
### Manage Software Packs

Nun wollen wir aber endlich das STM32Cube ExpansionPack zu unserem Projekt hinzufügen. Als erstes müssen wir der IDE sagen wo sie das ExpansionPack findet.



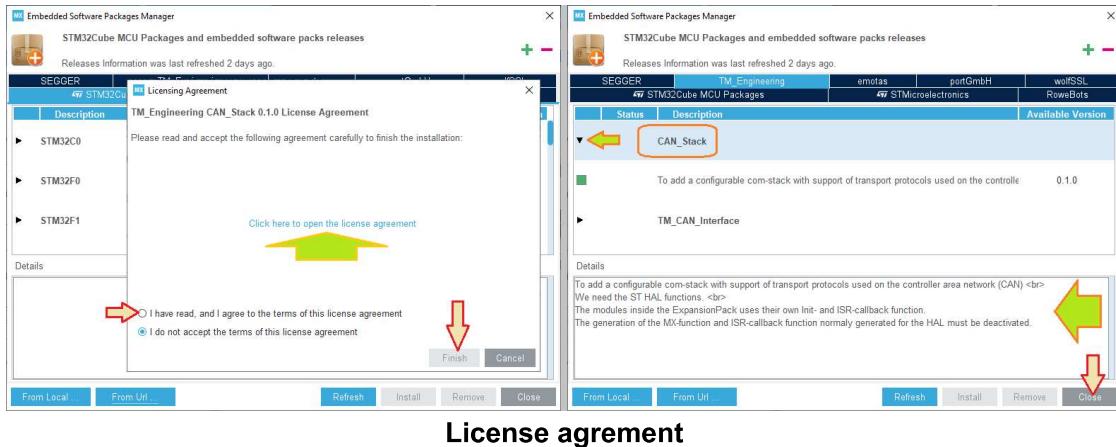
Open manage software packs dialog

Wir gehen also wieder in die Ansicht der CanIf\_AppDemo.ioc Datei und wählen "Manage Software Packs".



load pack

Man kann das ExpansionPack entweder lokal von der Festplatte oder per URL aus der Ferne integrieren. Wir hatten **am Anfang** das ExpansionPack herunter geladen (oder wir haben es parallel in einem Development-Verzeichnis). Dieses ExpansionPack suchen wir hier und wählen es aus.



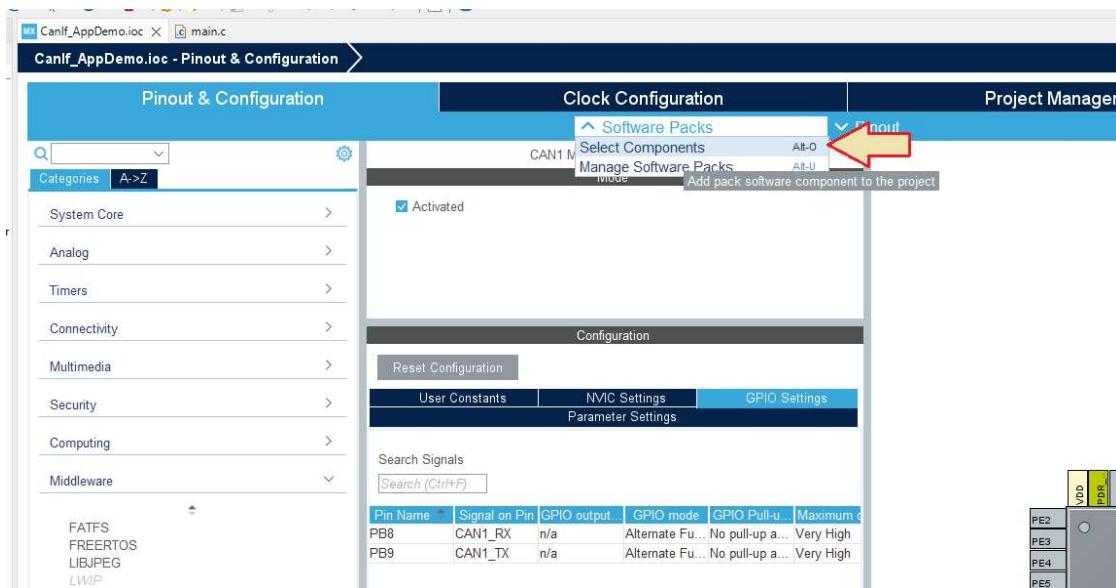
License agreement

Wenn wir dann die Lizenzbedingungen gelesen und akzeptiert haben, ist das ExpansionPack integriert.

Man findet hier in dem Fenster auch ein paar nützliche Informationen zum Umgang mit dem ExpansionPack. Da steht zum Beispiel etwas zu den MX-Funktionen.

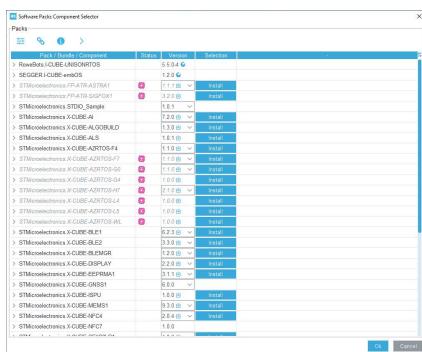
## Auswahl der Module

Jetzt müssen wir noch auswählen, welche Komponenten aus dem ExpansionPack genutzt werden sollen.



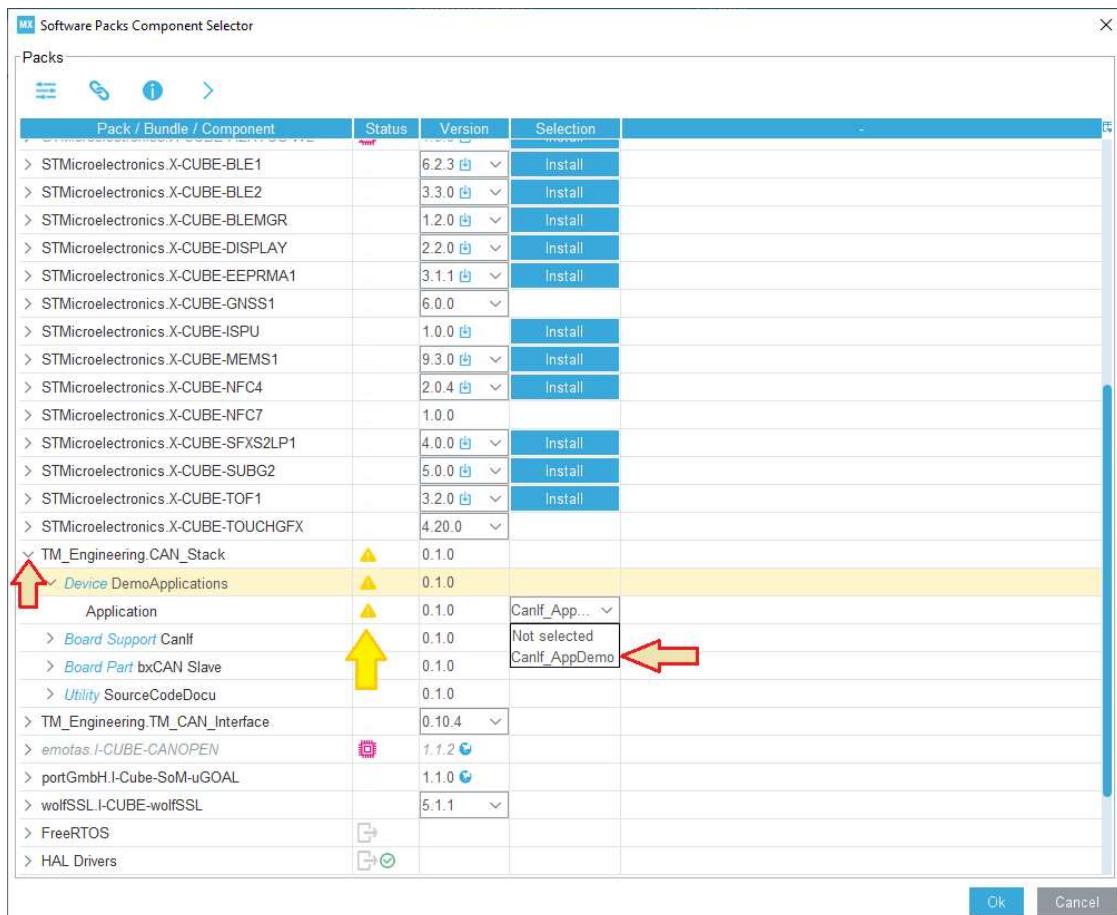
load pack

Es öffnet sich der "Software Pack Component Selector". Erst einmal ein kleines Bild. Ins Detail gehen wir gleich.



Software Pack Component Selector

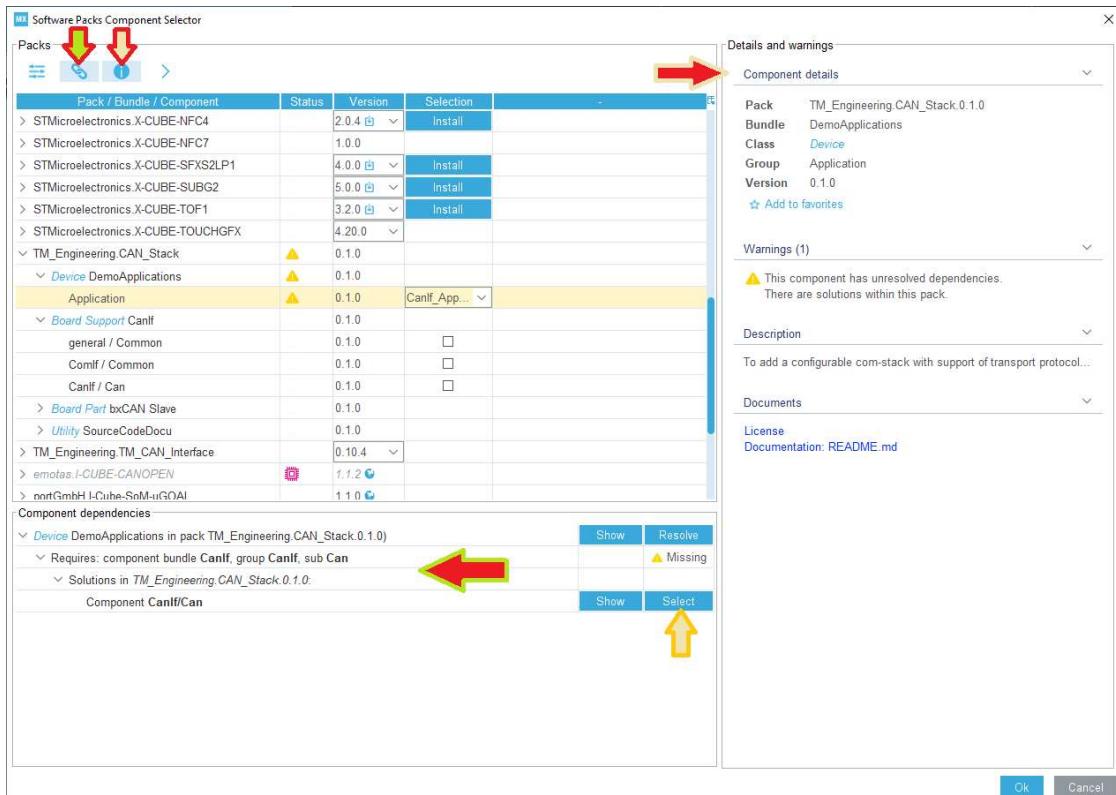
Hier sind unter anderem alle von STMicroelectronics verfügbaren ExpansionPacks auswählbar. Klickt man auf Install wird das jeweilige ExpansionPack bei STM herunter geladen und installiert. Wir scrollen aber ein wenig nach unten und suchen **TM\_Engineering.CAN\_Stack**.



Select the Demo Application

Durch Klick auf ">" kann man die einzelnen Module, oder genauer gesagt die einzelnen Bundles öffnen. Naiv wählen wir die Demo Application **CanIf\_AppDemo** (deswegen haben wir unser Projekt am Anfang auch so benannt) aus.

Nun werden da aber ein paar gelbe Warnschilder mit Ausrufezeichen angezeigt.



### Bundle warnings and Info

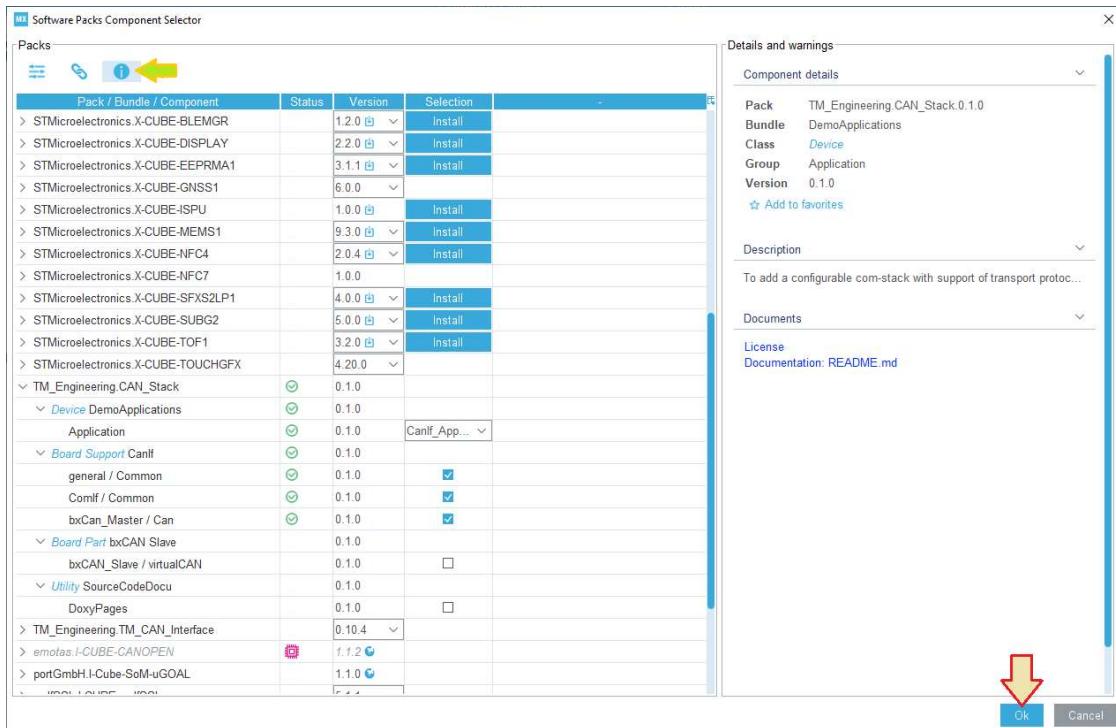
Klickt man auf den Button (siehe beige/roter Pfeil) kann man ein paar Informationen zum jeweils aktiven Element bekommen. So steht zum Beispiel bei unserer Application: "This component has unresolved dependencies". Es braucht also noch irgend ein anderes Modul. Weiter steht da "There are solutions within this pack". Die Lösung des Problems liegt also nahe.

Mit dem Button (grün/roter Pfeil) dann kann man die Abhängigkeiten unmittelbar anzeigen. In unserem Fall braucht die DemoApplication eine Komponente CanIf/Can

Wir ergänzen also die noch notwendigen Module die sich alle im Bundle Board Support befinden.

- Dies kann man unter anderem in dem man direkt auf "Select" klickt.
- Mit "Show" kann man die Lösung vor selektieren.
- Und mit "Resolve" löst man alle verketteten Probleme mit einmal. Also in unserem Fall werden die drei erforderlichen Module ausgewählt.

Module	Inhalt
general / Common	Hier sind die Header Files mit den Compiler Abhängigkeiten oder auch eine Klasse zur Handhabung von Versionsinformationen zur Laufzeit enthalten
ComIf / Common	Darin ist eine virtuelle Klasse enthalten, welche zur Abstraktion von Kommunikation Interface Klassen ähnlich dem AUTOSAR Ansatz vorgesehen ist
bxCAN_Master / Can	Dies ist unsere abgeleitete Interface Klasse welche den Master des bxCAN Controllers darstellt



### Select needed modules

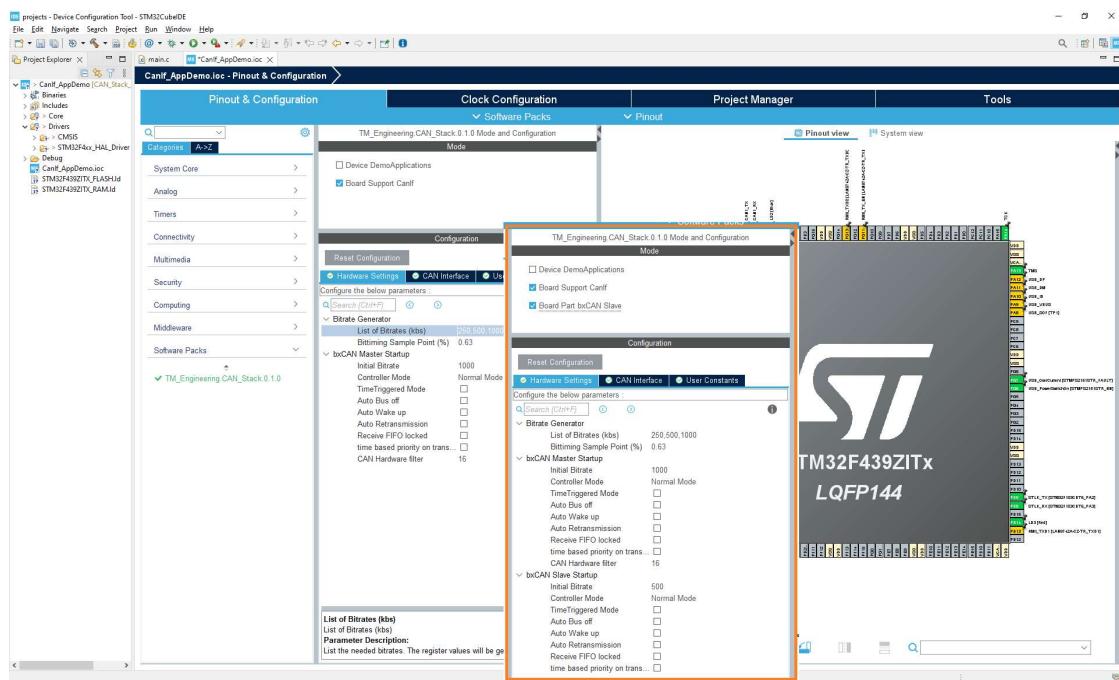
Haben wir diese drei Module aktiv, werden wir mit grünen Häkchen belohnt und können den Selector wieder schließen.

## Konfiguration der Module

Das Aktivieren der Module erfolgt auf dem gleichen Weg, wie wir zu Beginn das CAN-Modul aktiviert haben. In der linken Spalte des CanIf\_AppDemo.ioc Viewers findet man jetzt einen weiteren Punkt "Software Packs". Unter diesen Punkt werden alle aktiven ExpansionPacks aufgeführt. Aktuell haben wir nur ein Pack, also ist hier auch nur unser TM\_Engineering.CAN\_Stack zu finden. Aktiviert man das Modul "Board Support CanIf" erhält man im Configuration Abschnitt die Tabs "Hardware Settings", "CAN Interface" und "User Constants". Der Tab "User Constants" ist der gleiche den wir schon aus den Einstellungen des CAN-Modules kennen.

## Hardware Settings

Die Hardware Settings sind jene, welche durch das CanIf-Module kontrolliert werden. In der AUTOSAR Spec werden diese Parameter durch das CAN-Modul bearbeitet. Wir müssen dies über die STM-HAL erledigen.



### Config the Hardware Settings of the CanIf module

Auch in dem "Configuration" Abschnitt gibt es wieder einen (i) Button um mehr über die einzelnen Parameter zu erfahren.

## Bitrate Generator

Mit diese Parameter werden durch den Source Code Generator die notwendigen Hardware Parameter berechnet um die jeweiligen Baudraten zu erzeugen. Neben diesen Parametern wird die Clock des CAN Modules aus den Systemparametern genutzt.

### Remarks

Aktuell existiert kein Support für System Clock Switch während der Laufzeit!!!

Parameter	Beschreibung
List of Bitrates	Hier wird die Liste der gewünschten Bitraten angelegt. Die einzelnen Bitraten werden durch Kommas getrennt. Die hier gewählten Baudraten sind während der Laufzeit auswählbar.
Bittiming Sample Point	Sample Point als Prozent oder Kommazahl. Der Sample Point muss entweder größer 50% oder eben $0.5 < x < 1$ sein

## bxCAN Master Setup / bxCAN Slave Setup

Hier werden die Startup Parameter des bxCAN Master Controllers festgelegt.

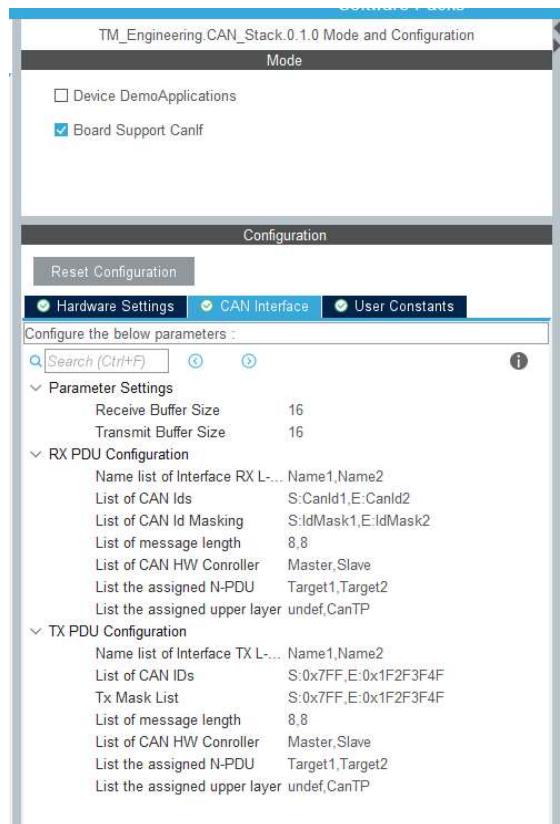
Wenn man den bxCAN Slave auch aktiviert hat wird ein identischer Abschnitt für diesen angezeigt.

Parameter	Wert	Beschreibung
Initial Bitrate	[ Zahl, Integer ]	Hier kann eine Bitrate ausgewählt werden welche zuvor im Bereich "Baudrate Parameter" definiert wurde
Controller Mode	Normal	Standard. Der Controller nimmt ganz normal am Bus teil
	Loopback	CAN-Tx und CAN-Rx des Controllers werden innerhalb des µC verbunden. Damit lässt sich ohne weiteren Teilnehmer ein aktives Busverhalten

		simulieren.
	Silent	Der Controller nimmt nicht an der Arbitrierung Teil. Damit kann man auf einem Bus mithören ohne diesen zu beeinflussen
	Silent & Loopback	Kombination aus den beiden
Time Triggered Mode	true / false	Zeitstempel
Auto Bus off	true	Der CAN Controller geht nach einem Bus-Off von allein in den aktiven Zustand
	false	Der CAN Controller muss per Software vom Bus-Off in den aktiven Zustand gebracht werden.
	-	Der Bus-Off Zustand kann erst verlassen werden wenn 128 mal 11 rezessive Bits empfangen wurden (Siehe CAN-Error Management)
Auto Wake up	true	Der CAN Controller wacht bei eingehenden Nachrichten von alleine auf
	false	Der CAN Controller muss per Software geweckt werden
Auto Retransmission	true / false	Wählt aus ob die Nachrichten bei fehlgeschlagener Arbitrierung erneut gesendet werden sollen
Receive FIFO Locked	true	Neue Nachrichten überschreiben den vollen Rx-FIFO
	false	Es werden keine weiteren Nachrichten mehr angenommen, wenn der Rx-FIFO voll ist
time based priority on transmission	true	Das Senden erfolgt in der zeitlichen Reihenfolge wie die Messages dem Tx-FIFO übergeben werden
	false	Das Senden unterliegt den normalen Arbitrierungsregeln, es wird also entsprechend der CAN-Id der Vorrang gegeben
CAN Hardware filter	[ Zahl, Integer ]	Gibt die maximale Anzahl der genutzten Hardware Filter für diesen Controller an. Der bxCAN Slave hat die restlichen Filter des bxCAN Masters zur Verfügung

## CAN Interface

Ich muss hier dann noch das System mit den PDUs erklären.



### Parameter on CAN Interface

## Parameter Settings

Größe des Rx bzw des Tx Buffers.

Im Gegensatz zu AUTOSAR habe ich die Buffer im **CanIf** Module umgesetzt. Das bxCAN Modul hat nur drei Rx Mailboxen. Das Hardware Module entspricht der STM HAL, welche ja unverändert übernommen wird.

## RX/TX PDU Configuration

Alle Parameter sind als Komma-Listen zu benutzen. Das heißt auch, dass in jeder Liste die gleiche Anzahl an Elementen vorhanden sein muss. Stimmt die Anzahl der Elemente eines Parameters nicht, so wird der Source Code Generator eine Fehlermeldung generieren.

In diesem Fall sind die generierten Konfigurationsdateien nutzlos.

### Todo:

Vielleicht kennt jemand ja einen Weg diese Konfiguration in Tabellenform ähnlich der FreeRTOS Konfiguration in der STM32CubeIDE umzusetzen.

Ziel ist es, dass die Konfiguration hier nur für Nachrichten ohne Übertragungsprotokoll notwendig ist. Aktuell kann das CanFT2p0-Protokoll die Konfiguration schon selber durchführen.

## Name list of Interface L-PDUs

Liste der L-PDU Namen.

Mit Hilfe der PDU-Namen wird der jeweilige "Kommunikationspfad" ausgewählt. So wird zum Beispiel der Funktion **CanIf::Transmit()** die **TxPduId** und ein Pointer zu den zu sendenden Datenbytes übergeben. Die Namen werden hier in der Oberfläche immer ohne irgendwelche Prefixe oder Ergänzungen genannt. Im Code werden die Namen dann mit dem Prefix des jeweiligen Modules sowie mit Rx-/Tx-Kennung versehen. Die erzeugten Namen aller Module sind dann in der **EcuNames\_Cfg.h** zu finden.

Wir können im Tutorial die Namen erst einmal so belassen. Ich werde bei den folgenden Punkten nur dann explizit erwähnen, das wir etwas ändern, wo es notwendig ist.

### List of CAN Ids / List of CAN Id Masking

Diese zwei Parameter werden zur Konfiguration der Hardware Filter genutzt.

Das erste Zeichen gibt an ob das Extended Id Bit (IDE) gesetzt werden soll. Aktuell wird nur geprüft ob das erste Zeichen ein 'E' oder ein 'e' ist. Ist dies nicht der Fall, dann wird von Standard CAN ausgegangen. Das Trennzeichen an zweiter Stelle ist egal. Alle Zeichen ab der dritten Stelle bis zum nächsten Komma werden unverändert in den Source Code übernommen. Es stehen also alle gängigen Formationen (hex, dec, oct, bin) zur Verfügung.

CanId1, CanId2 oder auch IdMask1 und IdMask2 sind keine gültigen Zahlen. Demnach müssen wir hier im Tutorial gültige CAN-Ids eingeben

### List of message length

Anzahl der Datenbytes

### List of CAN HW Controller

Der verwendete CAN Controller per Name. Aktuell werden die Namen des bxCan Modules unterstützt. Also können Master oder Slave benannt werden. Alternativ kann mit dem ersten Buchstaben, also 'M' oder 'S' abgekürzt werden.

Die FDCAN Controller der neueren STM32 werden durchnummeriert. Ich werde aber auch Master als CAN1 und Slave als CAN2 akzeptieren.

Wir nutzen im Tutorial nur den Master des bxCAN. Also müssen wir aus dem Slave auch einen Master (oder Kurzform "M") machen.

### List of assigned N-PDU

Hier wird der Name des übergelagerten PDU benannt. Diese Namen sollten durch die Konfiguration des jeweiligen Protokollmodules entstehen

Im Tutorial nutzen wir keine höheren Layer. Da diese Namen trotzdem vom Compiler gesucht werden, müssen diese Dummynamen bleiben.

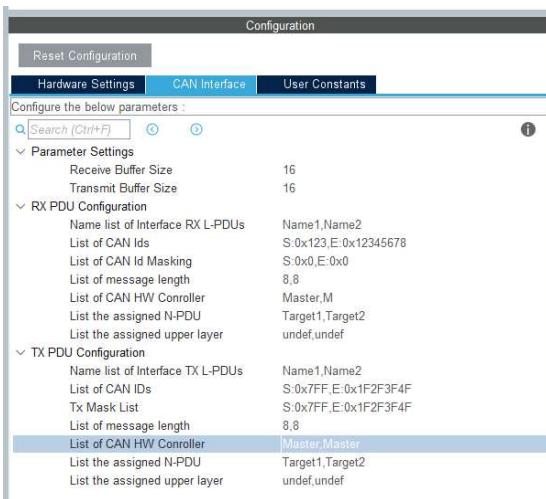
### List the assigned upper layer

Verwendetes Transport Layer (siehe auch [CanIf\\_UpperLayerType](#)). Das **CanIf** muss wissen an welches Protokollmodul eine empfangene Nachricht weiter gereicht werden muss, bzw wohin ein erfolgreiches Versenden zurück gemeldet werden muss.

AUTOSAR nutzt hierzu die Kombination aus "übergeordneten Layer" und "PDU-Id". Mit C++ lässt sich das auch einfach in Funktionspointeraufrufen realisieren. Es wird also die Zukunft zeigen, wie weit wir das hier nutzen.

Damit das **CanIf** nicht versucht das IsoTP-Protokoll anzusprechen, müssen wir das CanTP zweimal zu undef ändern.

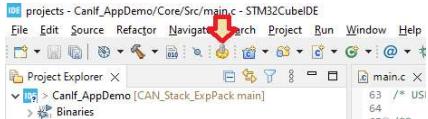
Nach der Änderung der Parameter sollte die Konfiguration so aussehen



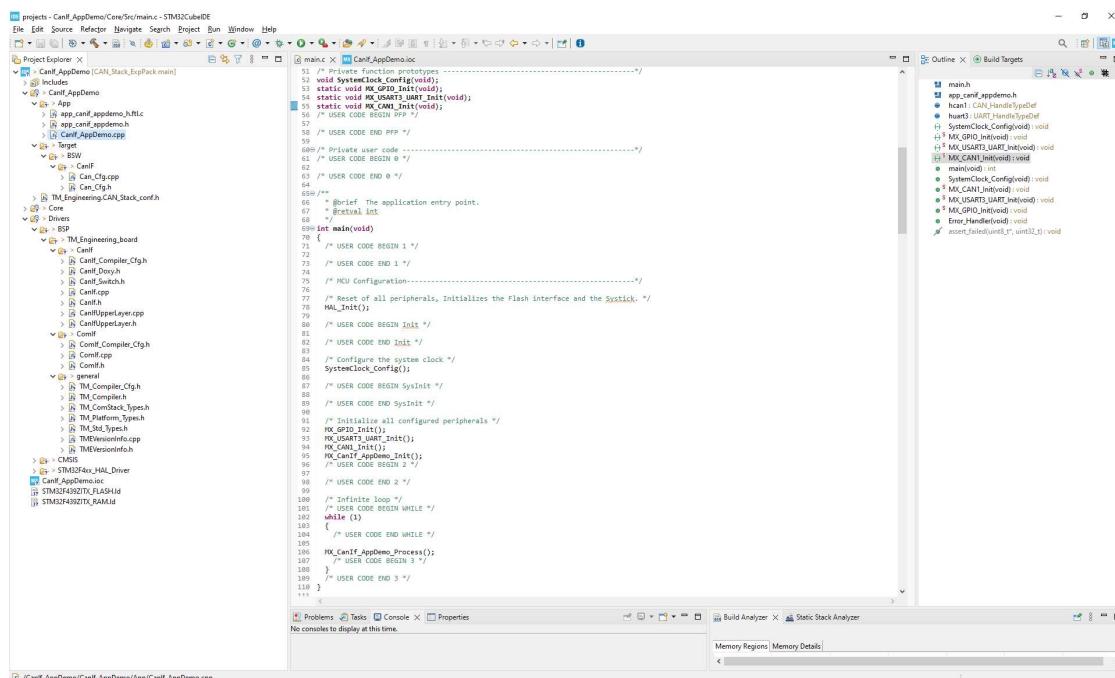
Change parameter CAN Interface

## Den Code für das CanIf Modul generieren

Jetzt noch das Modul für die DemoApplication aktivieren und wir können den Code Generator starten.



Wenn man jetzt im Projekt Browser mal in die vielen neuen Verzeichnisse hinein schaut, wird man neue Dateien entdecken.



All generated files

Alles was sich im Verzeichnis mit dem Application Namen befindet, sind generierte Dateien. Im Unterverzeichnis "Target" werden die eigentlichen Konfigurationen erzeugt. Das Verzeichnis "App" beinhaltet die eigentliche Application.

Die generierten Dateien verhalten sich sich genauso wie wir es in der `main.c` kennen gelernt haben. Es werden also nur Änderungen innerhalb der "User Code Blöcke" über eine Code Generation hinweg beibehalten. Schaut man zum Beispiel einmal in die `EcuNames_Cfg.h` so findet man ziemlich am Anfang (die Kommentare sind für die Doku geändert)

```

/* * USER CODE BEGIN EcuNames_Cfg_h 0 * /

/* ** @brief this enum shows the naming conventions of used PDU names * /
typedef enum
{
    N_PDU_Dummy_for_Test,
    L_PDU_Dummy_for_Test,
    CanUndefUl_Rx_Target1,
    CanUndefUl_Rx_Target2,
    CanUndefUl_Tx_Target1,
    CanUndefUl_Tx_Target2
}CanUL_PDU_for_Test;

/* * USER CODE END EcuNames_Cfg_h 0 * /

```

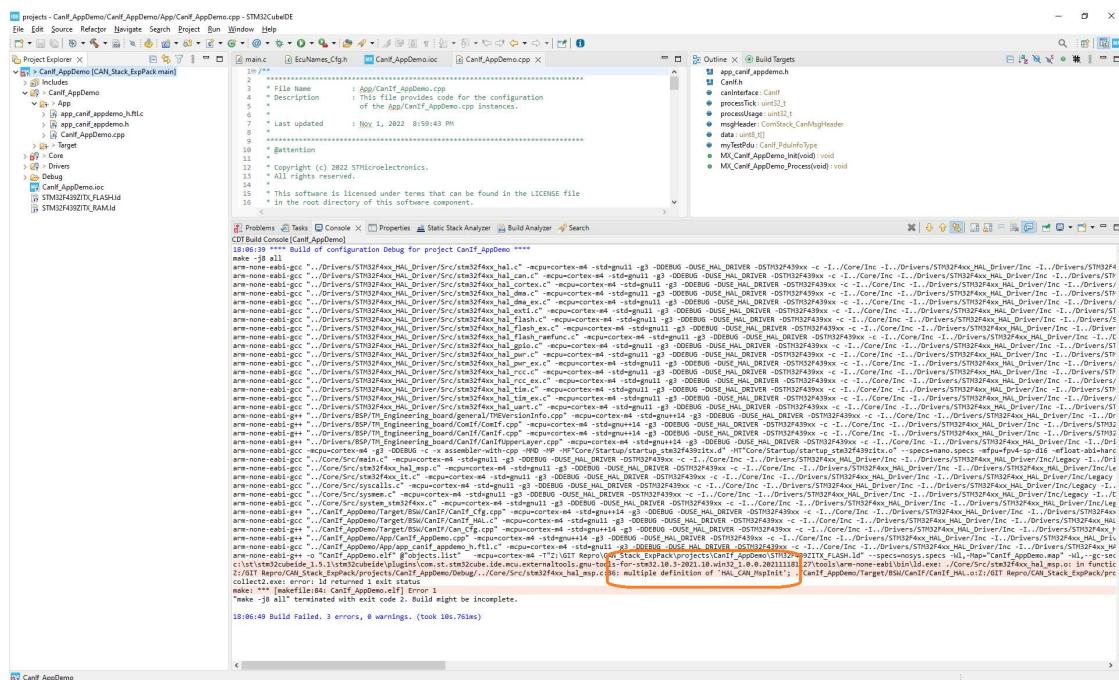
Man kann dieses enum also frei bearbeiten. Hier sind auch unsere Rx und Tx PDU Namen Target1 und Target2 als Dummy angelegt.

Im Verzeichnis "Drivers/BSP" gibt es jetzt ein Verzeichnis "TM\_Engineering\_board". Darin sind die kopierten Dateien aus dem Board Support Package. Diese Dateien werden bei jeder Code Generation neu hier her kopiert. Änderungen da drin können also nur zu Versuchszwecken durch geführt werden und sind nicht angedacht.

In der oben geöffneten main() kann man auch erkennen das eine MX\_CanIf\_AppDemo\_Init() und eine MX\_CanIf\_AppDemo\_Process() Funktion aufgerufen wird. Beide Funktionen sind in der CanIf\_AppDemo.cpp implementiert. Wir können also abseits der eigentlichen main() mit C++ objektorientiert weiter arbeiten.

## Das ExpansionPack übernimmt Aufgaben aus der HAL

Also wollen wir mal ausprobieren ob sich die "CanIf\_AppDemo" bauen lässt.



Wir erinnern uns daran, dass in der Information des ExpansionPack etwas stand, dass wir das Generieren der HAL-Funktionen deaktivieren müssen. Und genau das ist der Grund für diesen Linker Fehler.

## Diese Punkte muss ich noch ins Tutorial aufnehmen

- ich muss noch an das blaue LED denken
- Build führt zu Fehler multiple definition of 'HAL\_CAN\_MspInit'
- Deaktivierung des Code Generators für das STM-CAN-Modul
- Was ist mit den Interrupt Callbacks

## Was ist bei einem "normalen" Öffnen der IDE anders als beim ersten Start

Die "\*.ioc" wird nicht automatisch geöffnet