Konzept ESP32-sensact

# Begriffe

* Sensact = Sensor-Actor; der Projektname
* App = Ein Funktionsblock-artiges Softwareobjekt, das eine bestimmte Funktion wahrnimmt. Bekommt eine eindeutige u16-AppId innerhalb der Domain (s.u.). Einige AppId sind reserviert für NodeApps (s.u.)
* Node = Eine Hardwarebaustein, auf dem die sensact-Software läuft. Jede Node hat eine eindeutige u16-NodeId innerhalb einer Domain.
* NodeApp = Auf jeder Node läuft eine spezielle App, die NodeApp. Deren AppId entspricht der NodeId. Über diese App kann die Node als solche angesprochen werden, um beispielsweise Outputs zu setzen oder Inputs abzufragen
* Segment = ein Zusammenschluss mehrerer Nodes, die an einem gemeinsamen CAN-Bus hängen und die darüber einen gemeinsamen MessageBus unterhalten.
* Domain = ein komplettes Sensact-System, bestehen aus einem oder mehreren Segmenten. Innerhalb einer Domain ist die ID einer App und einer Node eindeutig
* Indizes sind in der Regel u16-Werte. Ein Index 0 hat KEINE spezielle Bedeutung! Ein hoher Indexwert als „magic number“ kann eine spezielle Bedeutung haben. Alle „magic numbers“ müssen in common.hh definiert werden

# Konzepte der Funktionsweise

Auf jeder Node laufen diverse Apps. Sie sind lose gekoppelt und kommunizieren durch Nachrichten miteinander.

### Variante 1: Nur eine Nachrichtenklasse „Event

Eine Nachricht geht an eine bestimmte ID und hat einen bestimmten Nachrichtentyp. JEDE Nachricht wird auf den lokalen und MessageBus gelegt und ist damit von allen Nodes im Segment empfangbar

### Variante 2: Zwei Nachrichtenklassen „Event“ und „Command“

mit den Nachrichten „Command“ und „Event“ miteinander. Ein Command ist dabei FÜR eine bestimmte App bestimmt „Du, Kellerleuchte 1, schalte dich ein!“. Demgegenüber informiert ein Event über ein Ereignis einer App „Ich, Lichtschalter 42, informiere darüber, dass ich gedrückt wurde – und zwar alle, die es interessiert“. Jeder App steht es frei, die Rollen EventEmitter, EventReceiver, CommandEmitter und CommandReceiver zu implementieren. „Alles kann, nichts muss“ lautet die Devise. Commands und Events werden über den sog. Message Bus verteilt. Dieser verteilt sowohl Nachrichten Node-intern, als auch über den CAN-Bus an alle anderen Nodes

## Konzept einer „Node“

Eine Node hat eine u16ID > 0, ist der Host für Apps und stellt physische Inputs und Outputs zur Verfügung.

# Konzept Inputs und Outputs

Jeder Input und jeder Output wird über eine 16bit ID angesprochen und liefert/erwartet einen 16bit-Wert. Inputs und Outputs leben im gleichen Namensraum. Es gibt also keinen Input mit der gleichen ID wie ein Output.

Die Inputs werden auf verschiedene Busse aufgeteilt. Von der 16bit ID werden die 2 MSB für die Auswahl des Busses verwendet. Es gibt die folgenden Bus-Typen

* DirectGPIO, also direkte GPIOs
* I2C
* CAN

Es gilt:

* 0b00 ist immer INTERNAL
* 0b11 ist immer CAN-Bus

Innerhalb des Busses können also 14bit frei verteilt / zugeordnet werden.

## Implementierungsstufe 1: Synchrones Abfragen und Setzen aller InOuts

Insbesondere der I2C-Bus läuft nicht in einem eigenen Task. In der Input-Phase werden alle konfigurierten I2C-Geräte abgefrage. In der Output-Phase wird auf alle I2C-Geräte geschrieben, bei denen es eine Änderung gab

Die Busse laufen in eigenen Tasks. Ihre Aufgabe ist es, effizient das in Ihnen intern gespeicherte Prozessabbild zu aktualisieren. Während des Loops darf aber keine Aktualisierung erfolgen.

## Implementierungsstufe 2: Asynchronität

Die Busse laufen in einer eigenen Task. Für Inputs gilt: Jeder Treiber verwaltet intern in einem Staging den neu hergeholten Wert. Ein CommitInputs-Aufruf sorgt dafür, dass die Werte von Scratchpad auf „WhitePaper“ geschrieben werden und dann für Apps zur Verfügung stehen. Dieses Umschreiben – und das ist entscheidend – findet nur beim Commit statt. Aufgabe des Nodemasters ist es deshalb, just vor dem App-Loop die ganzen Commits aufzurufen – über den Busmaster.

Umgekehrt können bei den Outputs die Apps erst in einen Staging-Bereich schreiben. Ein CommitOutputs sorgt dann dafür, dass die Werte möglichst effizinent an die Hardware rausgeschrieben wird. Wenn CommitOutputs zurück kehrt, müssen die Outputs nicht zwingend voll geschrieben sein, aber der Anstoß zum Schreiben muss unumkehrbar sein

## DirectGPIO

STM32: PA0 = 0, PA15=15, PB0=16, PB15=31 etc.

ESP32: GPIO-num

Beim Internal-Bus, finden sich auch die folgenden MAGIC-Numbers

Die ID 0x3FFF steht für einen Input mit dauerhaft 1 (letzte Bits sind 1 (wie I=input)-1)

Die ID 0x3FFE steht für einen Input mit dauerhaft 0 (letzte Bits sind 1 (wie I=input)-0)

Die ID 0x3FFC steht für einen unbeschalteten Output

## I2C-Bus.

* Problem: Über I2c können Subbusse bspw über DS2482-OneWire, implementiert werden
* Array mit 16 Einträgen fix (NULL zulässig). Erster Eintrag ist fest auf I2CMainBus eingestellt. Die anderen Einträge können (!) OneWireSubbus-Member enthalten
* Jeder Busmaster verwaltet eine Liste von InOut16-Membern (PCA9555, PCA9685, BME280/680). Am I2C können maximal 64 solche Member hängen. Dadurch kommen 32\*16=1024InOuts zusammen.
* Jeder Busmaster verwaltet eine Liste mit maximal 15 Subbussen(derzeit nur OneWire via Ds2482), Jeder Subbus bekommt auch 1024 InOuts, also 1024-2047, 2048-...
  + Offsets für OneWire-Subbusse mit dem DS2482
  + Subbus verwaltet eine Liste (max 96 Einträge) von sensactSE-Devices (ein Device hat 8 InOuts), also 768 InOuts
  + Subbus verwaltet eine Lists (max 128 Einträge) von DS2413-Devices (ein Device hat 2 InOuts) also 256 InOuts

## CAN-Bus

Auch über den CAN-Bus können Ios verteilt werden

* Wenn eine Node die IOHost-Rolle implementiert, dann kann über ihre NodeApp (AppId entspricht NodeId!) auf die Ein-und Ausgänge zugegriffen werden. Die NodeIds sollten sich im Bereich 0…64 bewegen. Jeder IOHost kann dann IO 0…255 haben.

## Konfigurierbare Ein- und Ausgänge

Bestimmte physische Pins können sowohl als Eingang als auch als Ausgang fungieren. Direkt nach dem Start führt eine Node eine Konfigurationsprogramm aus, in dem alle Pins (sowohl die CPU-Pins als auch I2C-Pins als auch Funk-Pins als auch Wifi-Pins als auch CAN-Pins, aber eben nur solche in Verantwortung der Node…) durchkonfiguriert werden. Vor der Konfiguration sollen alle Pins bevorzugt hochohmig bzw im inaktiven Zustand sein („0“ bzw. Transistor nicht durchgeschaltet / Relais nicht angezogen). Konfigurierbare Inputs/Outputs kommen sowohl im Input- als auch im Output-Indexraum vor und müssen einen identischen Index haben

Ein nicht vorhandener Input liefert 0. Auch die Anfrage eines nicht als Input konfigurierten Index liefert 0.

## Technische Realisierung von Outputs

* Transistor-Outputs an Port-Pins (beim ESP32 bitte immer PWM-fähig)
* Als Port-Expander am I2C
* Als Port-Expander am 1Wire
* Als per CAN angeschlossenes Device, das mit CAN-Messages in einem speziellen InOut-Namespace angesprochen werden kann. Weil derartige Outputs (ebenso die Inputs) letztlich jeder Node zur Verfügung stehen, müssen dafür bei den u16-Indices für Inputs und Outputs einige frei gehalten werden

## Technische Realisierung von Inputs

Technisch können Eingänge direkt über PortPINs der CPU realisiert werden. Alternativ

* I2C-Port-expander
* Fernbedienungen

## Aufteilung der Input- und Output-Namespaces

Letztendlich steht es in der Verantwortung der Node, die von ihr verwalteten Ios in den beiden u16-Namespaces zu organisieren. Empfohlen wird die folgende Aufteilung:

* Grundsätzlich 64 Blöcke a 1024 Indices (also: Ein kleinerer Block als 1024 wird nicht vergeben)
* 0 ist der NULL-Output
* Block 0( 1-1023) sind GPIO-Pins
* Block 1 bis 17 sind für 16 angeschlossene I2C-Devices
* Block 64 sind CAN-IOs

## Verhalten von Ausgängen

Es gibt binäre, analoge und PWM-Outputs. Ein binärer Output schaltet (voll) ein, wenn der Wert !=0 ist. Ein PWM-Output liefert dann nur ein 1/65535-PWM. Um auch mit PWM-Signalen binäre Ausgänge nachahmen zu können, sollte ein Ausgabebefehl bspw für ein Relais immer UINT16\_MAX als Wert setzen.

# Application

Eine App ist ein Softwareobjekt, das eine bestimmte fachliche Funktionalität mit Bezug zur Gebäudeautomation wahrnimmt. Beispielsweise sind „Taster Wohnzimmer 1“ oder „Rolladen Küche 2“ oder „Beleuchtung Kellerraum 1“ solche Apps. Eine App „lebt“ auf einer Node.

Jede App hat eine systemweit eindeutige ID. Über diese ID kann die App im System Events publizieren bzw. auf Commands reagieren.

## Blind-App

* Gegenwärtige Postion wird in einer s32-Variable „estimatedPosition“ gespeichert
* App kennt Command zum Verfahren auf eine Position und zum kompletten Öffnen und Schließen.
* SAFE\_CLOSE=10%, FULL\_CLOSED= 25%, FULL\_OPEN=75%, SAFE\_OPEN=90% des Variablen-Maximalwertes
* App kennt die Zeiten zum kompletten Öffnen und kompletten Schließen des Rollos. Daraus wird ein „millistepsUp“ und „millistepsDown“ ausgerechnet. In der angegebenen Zeit geht der Rollo von FULL\_CLOSED nach FULL\_OPEN.
* Ein externer Positionswunsch 0…100% wird auf intern 25…75% abgebildet \*0,5+0,25
* Ein externer Schließen-Wunsch wird intern auf „10%“ abgebildet
* Ein externer Öffnen-Wunsch wird intern auf 90% abgebildet.
* Ein externer Stopp-Wunsch wird auf einen internen Stopp-Wunsch abgebildet
* Ein interner Stopp-Wunsch stoppt die Motoren und macht 25%<=pos<=75%
* In der Loop wird geprüft, ob und in welche Richtung der Motor läuft. Es wird entsprechend die millistepsUp/Down auf die Position drauf gerechnet. Der Motor wird – je nach Richtung – gestoppt, wenn die estimatedPosition kleiner bzw größer ist, als der Zielwert
* requestedState={STOP,DOWN,UP}, requestedPosition=u32. Diese requested-Values werden gesetzt, wenn ein Command kommt. Ob up oder down, wird über die currentPositions festgelegt. Das Command wird nur angenommen, wenn die currentPos von der requestedPos signifikant abweicht (einen Motorlauf von >1sek erfordern würde)
* Wenn requestedState=STOP, wird die requestedPosition nicht weiter beachtet
* Wenn das Command gesendet wird, dann wird festgestellt, ob

# Code-Generierung

* Wird aktuell schon ganz gut gemacht
* Nutze dictionary intensiver!!!
* Generiere C++Code
* Die Code-Generierung soll minimal sein und möglichst keine festen Abschnitte umfassen. Deshalb werden mehrere kleine Dateien erzeugt, die im Template an verschiedenen Stellen eingebunden werden.
* Damit der Code compilierbar ist, auch wenn die Dateien noch nicht erzeugt wurden (wenn es nicht verfügbar ist, zumindest #warning ausgeben:

// Note the two possible file name string formats.

#if \_\_has\_include("myinclude.h") && \_\_has\_include(<stdint.h>)

# include "myinclude.h"

#endif

* Templates sollen auch ohne generierten Code syntaktisch korrekt sein
* Generierte Dateien haben die Endung .gen.hh bzw .gen.cc (bisher: inc), um klar zu machen, um was für Dateien es sich handelt

# CAN-Bus-Kommunikation

* 125kbits CAN-Bus
* ExtendedID mit 29bits, 5bit Namespace, dann 24bit namespace-spezifisch
  + NodeEvent = 0x00000000,
  + AppEvent= 0x01000000, AppEvents. 16bit EventEmitterID, 8 bit EventType
  + NodeCommand = 0x02000000,
  + NodeCommandAcknowledge= 0x03000000,
  + ApplicationCommand= 0x04000000, 16bit CommandTargetID, 8 bit CommandType
  + CommandAcknowledge= 0x05000000,
  + ApplicationStatus = 0x06000000,
  + Payload = 0x1F000000,
  + Namespace 0b00001
* Modi
  + Wie bisher: Wenn möglich, Dinge komplett lokal bearbeiten und nicht auf den Bus geben
  + Alle Events auf den Bus geben
  + SensactUP-Nodes sind „dumm“. Sie senden alle Aktivitäten des Nutzers an eine Zentrale. Diese verarbeitet, entscheidet und sendet dann wiederum Commands an Aktoren
  + Wie zuvor, allerdings ist die Zentrale ein per MQTT angebundener PC

## Namespace Events 0b00000

* EventID besteht aus EventEmitterID (16bit, wer sendet es weg) und EventType (8bit, was ist dort passiert).
* Ein Event kann außerdem noch bis zu 8 Byte Nutzdaten transportieren
* EmitterID=0 bedeutet: Event kommt von der Management-Ebene
* EventType=0 bedeutet: Unspezifiziertes Event vom Sender
* EventType 0xFF bedeutet: Dem Empfänger ist es egal, welches konkrete Event gesendet wird. Der Empfänger reagiert damit auf alle Events gleichermaßen
* Event Emitter sind
  + „ButtonSensor“ mit Press, Hold (wenn sicher ist, dass es kein ShortPress ist), ReleaseShort (auch wenn danach ein DoubleClick kommt), ReleaseLong, DoubleClick (DoubleClick hat zuvor einen ReleaseShort, aber dann kein Press und ReleaseShort mehr) (Prinzip: Events werden zum frühestmöglichen Zeitpunkt gesendet)
    - Was wird da gegenwärtig überhaupt gebraucht?, insbesondere für den SingleButton-Dimmer und den SingleButtonBlinds
  + „ButtonSensor“ remote mit Press und Hold
  + RotarySensor mit TurnLeft, TurnRight, Press
  + JogDial (sowohl rund als auch gerade) mit ValueSelected und dem aktuellen Wert
  + PWMLightingActor mit OnTriggerHeartbeat
* EventReceiver sind
  + BlindsActor kennen eine Liste ihrer „Up“-ButtonSensors und ihrer „Down“-ButtonSensors. SingleControl-Buttons gibt es da nicht
  + PWMLightingActor kennen die Liste der Ups, der Downs, der SingleControls und der RotarySensors. Ein derartiger Actor ist für die kompletten LEDs eines Raumbereichs zuständig (nicht: jeder Spot hat eine eigene App)
  + BinaryActor kennt Toggles, Ons, Offs, Heartbeats
  + SoundActor hat eine Map EventId🡪Song-Index

# Special-Hardware

* Lampe in Klaus Arbeitszimmer 🡪10V PWM
* Fingerprint 🡪Potentialfreier Kontakt
* Klingel 🡪Taster
* RGB-LEDs in den Bädern: 12V-LEDs, gemeinsame Kathode (mit zusätzlichem Vorwiderstand einfach an 24V)

# Software-Architektur

## Rollen

Alle Nodes, die über den CAN-Bus miteinander kommunizieren, erhalten eine Software, die dem im folgenden beschriebenen Architekturkonzept entspricht. Eine Node kann die folgenden Rollen (auch parallel) übernehmen

* Gateway: Führt Apps aus und bündelt sammelt alle Status- und Event-Infos im CAN-Netzwerk zusammen, um diese per MQTT weiter zu tragen. Leitet auch Infos aus MQTT in den CAN-Bus ein. Es muss im Netzwerk genau eine Node geben, die diese Rolle übernimmt.
* UserInterface:
* ApplicationHost: Führt Apps aus
* IOHost: Führt keine Apps aus, sondern verbindet nur „low-level“ physische Ein- und Ausgänge mit dem CAN-Bus

## Softwareobjekte und deren Verantwortlichkeiten

Frage: Nutzt HAL den Nodemaster oder nutzt Nodemaster den HAL? 🡪Letzteres!

* Main.cc
  + StartUp und grundlegende Konfiguration
  + Einmalige Ausführung des FirmwareUpdate-Prozesses beim Start
  + Start aller Tasks
* Updater.cc
  + Prüft regelmäßig oder explizit auf Befehl, ob eine aktualisierte Firmware vorliegt. Wenn ja, dann werden alle Tasks (Handle auf alle Tasks wird bei der Initialisierung übergeben) beendet und der Update-Zyklus ausgeführt
* HAL.cc
  + Unterste Hardware-Abstraktionsebene; greift auf nichts zu, sondern wird genutzt
  + Eine „Loop1ms“-Methode und eine „LoopTick“-Methode
  + Methoden zum Lesen oder Schreiben von GPIO-basierten u16-Inputs/Outputs
    - Auch ein möglicherweise angeschlossener Rotary-Encoder soll als u16-Input umgesetzt werden
    - Auch eine möglicherweise verbundene Fernbedienung soll als u16-Input umgesetzt werden
  + Methoden zum Zugriff auf I2C (synchron!, schöne Schnittmenge aus der STM32-HAL und „meiner“ I2C-Bibliothek für den ESP32
  + RGB-LED SetColor(ledId, CRGB color)
  + Sound: PlaySound(soundId), StopSound(), SetSoundVolume(u16)
  + Methoden zum Zugriff auf CAN (Gepuffert!)
  + Zugriff auf ein Display? 🡪NEIN, das muss separat gemacht werden
* Busmaster
  + Mittlere Abstraktionsebene
  + Gibt es in den Varianten LocalGPIO, I2C und CAN
  + Nutzt HAL für den Zugriff
  + Ist dafür verantwortlich, das Prozessabbild eines Busses zu aktualisieren
  + Jeder Busmaster startet seinen eigenen Task, in dem permanent das Prozessabbild aktualisiert wird.
  + Jeder Busmaster verwaltet seine Ios über Treiber-Objekte, die mit dem am Bus angeschlossenen Gerät kommunizieren können und auch dessen Daten cachen
* Nodemaster
  + Oberste Abstraktionsebene
  + Kennt die Rollen der Node und handelt entsprechend
  + Grundlage der Funktionsweise ist ein klassischer 3phasiger AS-Zyklus
  + Wenn Rolle APPLICATION\_HOST: Sie kennt alle Apps und führt die unendliche Schleife aus. Sie erzeugt ein HAL-Objekt und gibt allen Apps darüber Zugriff auf die Hardware. Für den Zugriff auf lokal verwaltete Inputs oder Outputs erzeugt Sie InputOutputRangeManager. Ein solcher stellt der Node für einen bestimmten Bereich im I-O-Indexraum Inputs und Outputs zur Verfügung. Ein IndexRange ist mind. 1024 Elemente groß. Node weiß für jeden Bereich, welcher Manager dafür vorgesehen ist. Jeder IoRangeManager hat die Lebenszyklusmethoden
* Init (beim Start des Systems)
* ReInit ? (zum Zurücksetzen, nach irgendwelchen Fehlern)
* Phase1\_UpdateInputs()
* Phase3\_UpdateOutputs()

Schreib- und Lesezugriffe werden direkt auf den IoRangeManager durchgeleitet. Durch die interne Konstruktion ist aber sicher gestellt, dass Ins und Outs frühestens beim Aufruf der entsprechenden Lebenszyklusmethode aktualisiert / für die Aktualisierung freigegeben werden

Alle Inputs und alle Outputs werden von einem IoMananger verwaltet. Ein IoManager

Ein LocalBus (incl. LocalSubbus) bindet IO-Baugruppen in den Input und Output-Indexraum ein. Bei de

## Tasks

* Sensor-Task holt sich die Messwerte der I2C und 1Wire-Sensoren
* PLC-Task agiert im 50ms-Zyklus

## Funktionalität des HAL

* „Rohzugriff“ auf I2C für die Enumeration und Konfiguration der Geräte
* Zugriff auf per u16 identifizierte Binäre/Analoge/Digitale Inputs und Outputs. Die Werte sind auch u16
* Zugriff auf den MessageBus als zentrales Kommunikationsvehikel
* KEINE Funktionalität ist:
  + Logging! Das Programmieren wir für den ESP-IDF alles um

Frage: Können diese „Spezialitäten“ über das Prinzip „u16-Input/Output“ oder „MessageBus“ an einen „SubsystemControl“-Namespace realisiert werden?

## Hardware-Konfiguration

Neben der Hardware (Hutschiene, Unterputz, Outdoor) und ihrer Version unterscheiden sich die Nodes durch die angeschlossene Hardware. Um welche Hardware handelt es sich? Kann diese durch Enumeration erkannt und in Betrieb genommen werden?

Typ der Geräte kann grundsätzlich mal über ihre I2C- bzw. OneWire Adresse erkannt werden. Aber: Einige dieser Geräte müssen konfiguriert werden und es muss klar sein, wie diese zu konfigurieren sind:

Alternative: Geräte kennen selbst Ihre Konfiguration und stellen diese dem Busmaster zur Verfügung? Nein, das kann nicht immer gefordert werden, weil „dumme“ Geräte ihre eigene Konfiguration nicht speichern können.

Es muss neben den sensact-Gerät-spezifischen generierten Dateien auch spezifisch programmierte Dateien geben, in denen dann die Konfiguration als Programmcode abgelegt wird . Ganz klassisch: Es handelt sich um eine Klasse, die von AbstractNodeHardwareConfiguration ableitet und eine noch zu definierende Menge an Funktionen für die verschiedenen Konfigurationsdomänen implementiert.

### Konfigurationsdomänen

* Konfiguration lokaler Ios
* Konfiguration 1Wire-Geräte
* Konfiguration der I2C-Geräte an ggf. mehreren Bussen
  + Enumeration aller Adressen
  + Map definiert, welche Befehle für welche Adresse zu senden sind
  + Fehler, falls ein in der Map definiertes Gerät fehlt bzw. ein Gerät gefunden wird, für das es keine Konfigurationseinstellungen gibt.
  + Zuordnung von InputIDs oder OutputIDs zu den Geräten
* Konfiguration Wifi: SSID und Passwort
* Konfiguration Ethernet: prüfe den Code, der da üblicherweise zur IP-Konfiguration benötigt wird
* Konfiguration CAN ist fix; keine individuelle Konfiguration erforderlic
* Konfiguration sonstiger lokal angeschlossener Peripherie
  + RS485 für Garagentor
  + eBus für Lüftung
  + Sound-Ausgabe
  + Lokale Sensoren, die mit I2C- oder 1Wire angeschlossen sind
  + RotaryEncoder
  + RGB-LED (die per Protokoll und nicht per 3xPWM angeschlossen wird)
  + Display
  + Funkempfänger Milight
* In der bisherigen Logik wurde das durch enum class realisiert   
  enum class Mp3Cfg{NONE, INTERNAL\_MP3\_I2S, UART4\_PA00\_PA01\_StandbyPE02, };  
  und es musste dann jede Möglichkeit im Quellcode berücksichtigt werden. Die enum musste als Template-Parameter gesetzt werden. In der main.cpp wurde dann eine Instanz der „richtigen“ Templates erzeugt 🡪die Main.cpp muss dann entweder „ifdefs“ enthalten oder für jede Node individuell existieren…