Konzept ESP32-sensact

# Begriffe

* Sensact = Sensor-Actor; der Projektname
* App = Ein Funktionsblock-artiges Softwareobjekt, das eine bestimmte Funktion wahrnimmt. Bekommt eine eindeutige ID innerhalb der Domain (s.u..)
* Node = Eine Hardwarebaustein, auf dem die sensact-Software mit Apps läuft.
* 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 eindeutig
* „0“ bzw „NULL“: Ein Index oder eine ID von 0 deutet innerhalb von Sensact stets darauf hin, dass etwas nicht vorhanden ist. Beispiele: Ein Ausgang mit dem Index 0 wird nicht benutzt (bzw: wenn etwas darauf geschrieben wird, wird das an keinem physischen PIN weiter gegeben) ein Eingang mit dem Index 0 wird nicht abgefragt (bzw. technisch: Der Eingang mit dem Index 0 liefert immer 0). „Echte“ Eingänge, Ausgänge und Apps beginnen deshalb immer bei Index 1.

# 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 ID > 0, ist der Host für Apps und stellt physische Inputs und Outputs zur Verfügung.

## Konzept Inputs und Outputs

Inputs und Outputs liegen in getrennten uint16-Indexräumen. Sowohl der Input 0 als auch der Output 0 haben besondere Bedeutung: Sie zeigen einen Not-Connected-Input bzw Output an. Die Werte von sowohl Inputs als auch Outputs werden durch einen uint16-Wert ausgedrückt (keine Unterscheidung zwischen binären, PWM und analogen Eingängen bzw. Ausgängen).

## 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.

# Konzept einer „App“

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 > 0. Über diese ID kann die App im System Events publizieren bzw auf Commands reagieren.

## Besondere AppIDs

AppIds sind uint16

0: Gibt es nicht. Eine Ziel-AppID von 0 bewirkt, dass keine Nachricht versendet wird, Eine Quell-AppID von 0 darf es nicht geben

0xFFFF ist eine einmalig im System vorhandene MasterApp, die Node-Befehle austeilen kann

# 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 (bisher: inc), um klar zu machen, um was für Dateien es sich handelt

# 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…

# 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?

# Bus-Konzept

Die Höchste Verwaltungsklasse ist „NodeManager“. 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

# 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

# SensactUP-Nodes

* Implementieren keine eigene Logik, sondern stellen ihre Bedienelemente einfach nur „zur Verfügung“
* Hintergrund: Der komplette Applikationsstatus soll an einer zentralen Stelle verfügbar sein
* Ggf ist auch der zentrale ESP32 „nur“ ein Protokollumsetzer?
* Nein, das grundsätzliche Konzept der dezentralen Hardware, die bestimmte Dinge lokal regelt und erst gar nicht auf den externen MessageBus schiebt, bleibt!

# 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

# Task-Konzept

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