# Nummerierung der Outputs und der Inputs

es werden

* 64 pca9685 (Beginn ab 1b1000 0000, Ausnahme 1110 000X)
* 8 pca9555 (Beginn ab 1b0100 0000)
* 4 DS2482 (Beginn ab 1b0011 0000)

vorgesehen

diese werden zu beginn abgesucht und konfiguriert

Outputs, egal ob PWM oder on/off gehen stets über pca9685.

Es existieren also maximal 64\*16 = 1024 Outputs und 8\*16=128 Inputs.

Inputs und Outputs werden über einen uint16 definiert, wobei für den Output die untersten 10bit und für den Input die

untersten 7 bit rausgeschnitten werden. Dann wird die zahl um 4 bit nach rechts geschoben, um die Adresse des Chips zu bekommen.

Outputs und Inputs mit gesetztem ersten MSB (>0x8000) werden als "stm32-Intern" betrachtet und sind dann eben nach außen geführte Kontakte

BlockOnDoubleRegister wird gelöscht. Die Erkennung von Doppelregistrierungen findet zur Codeerzeugungszeit statt.

Die Unterscheidung zwischen PWM und nicht-PWM-Output fällt weg - Alles wird gleich behandelt. Ein technisch nicht-pwm-fähiger Ausgang schaltet voll durch, wenn das PWM-Niveau > 0 ist.

Der Treiber für PCS9685 und PCA9555 muss dahingehend geändert werden, dass der Chip bei allen Funktionen als Parameter übergeben wird und nicht ab Objekt gespeichert ist. Wegen der SharedIRQ-Lines muss auch das IRQ-Handling rausgenommen werden.

# Steuerung über MQTT und auch Statusabfrage

MasterNode hält immer den aktuellen Status jeder Applikation. Der Status der Applikationen auf den SlaveNodes ist mit der MasterNode zu synchronisieren. Jede Node sendet deshalb den Status ihrer Applikationen regelmäßig und bei Veränderungen sofort auf den CAN-Bus

Regelmäßig: Wann immer die Node eine „runde“ Laufzeit seit ihrem Start erreicht hat und sie in der letzten Zeit keine Nachrichten auf den Bus gelegt hat, sendet sie von der nächsten Applikation den Status. Immer schreibt CallMeRegularly den aktuellen Status ihrer Applikation in einen ihr übergebenen Buffer und die Länge in einen ihr übergebenen size\_t pointer. Auch der Status der Node selbst wird regelmäßig von der Node publiziert (u32 errno und u8[0] mit Busfehler-Zählern, Busse sind i2c1, i2c2, ow1/lin1, ow2/lin2

Der Status muss in ein u64 passen, weil er über den CAN-Bus transportiert wird. Ein u64 muss ausreichen, um den Status zumindest für die Visualisierung zu definieren. Der Status kann nie vom Master in die Node geschrieben werden. Hierfür sind stets Commands zu verwenden.

Jede Applikation bekommt eine GetStatus und eine SetStatus-Methode.

GetStatus erwartet einen Buffer mit einer Mindestlänge von 64byte, u32-Aligned. Er kann diesen Puffer vollschreiben mit seinem Status und gibt die Anzahl der geschriebenen Bytes zurück

SetStatus erwartet einen Buffer mit einer Länge, die von der Applikation abhängig ist. SetStatus liest diesen Puffer und sorgt dafür, dass der Status richtig gesetzt wird. SetStatus führt bei der Abarbeitung Prüfungen, insbesondere die Länge, durch. Wenn etwas „faul“ ist, wird der Status nicht gesetzt

Zunächst soll der Status immer in 8byte reinpassen.

Bei jedem Start einer Applikation muss diese ihren Status von der MasterNode holen. Ein Neustart der Masternode wird per Event kommuniziert und lässt jede Node so verhalten, als ob sie selbst gerade gestartet wurde. Jede Node erfragt also für jede ihre Applikationen den Status und setzt dann den Status in der Applikation. Eine kleine Besonderheit gibt es: Damit sich die Nachrichten auf dem CAN-Bus nicht überlagern, werden die Abfragen leicht verzögert gemacht.

Jede Node hat also Funktion SetAllApplicationStati(bool triggeredThroughMasterNodeStartup). Diese wird mit „false“ beim eigenen Hochfahren und mit „true“ beim Event „MASTER\_NODE\_STARTED“ aufgerufen.

Intern iteriert die Funktion über alle Applications, sendet ein SET\_MY\_STATUS(applicationId) aus und wartet so lange immer wieder (!) 200ms, bis die richtige Nachricht kommt. Sofern es hier einen Timeout gibt, springt die Node zur nächsten Applikation.

Die Master-Node sendet nach dem Hochfahren das „MASTER\_NODE\_STARTED“-Event aus. Eintreffende SET\_MY \_STATUS-Commands werden mit SET\_STATUS-Commands an die Applikation beantwortet, wobei der Master die Daten hierfür aus seinem Speicher nimmt.

Ein Hauptknoten – der auch die MQTT-Anbindung hat - verwaltet auch die Statuspakete zu jeder Applikation. Nur er kennt auch die Namen jeder Applikation. Alle anderen Nodes kennen weder Namen noch Statusinformationen der anderen. Beim Start fragt der Masterknoten alle ihm bekannten Applikationen nach ihrem Status per CAN-Bus. Die die Applikation hostende Node überführt die CAN-BUS-Anfrage in einen Aufruf der GetStatus-Funktion. Die Antwort besteht ggf aus mehreren Nachrichten wenn der Status länger als 8byte ist). In der Nachricht ist deshalb codiert, wie viele Nachrichten nach der aktuellen noch gesendet werden. Die Masternode kann sich dementsprechend für jede Node Speicher auf dem Heap allokieren und die Statusinfo abspeichern. (Problem: Was passiert, wenn eine Node neu geflashed wird und neue Applikationen hinzu kommen bzw wenn eine Applikation auf einmal mehr Speicher benötigt -> Neben dem Pointer auf den Status-Payload speichert die Mainnode auch ein Info-Objekt [Time\_t lastUpdate, size\_t length]. Die lenght wird immer abgeglichen mit dem, was die Node sendet. Beim Hochfahren fragt die Masternode immer alle Applikationen ab. Beim Hochfahren erfragen normale Nodes immer ihren Status bei der Master-Node. Sobald der Status durch eine andere Art als SetStatus akt

Wenn eine CAN-Nachricht „STATUS\_CHANGED“ am Bus anliegt, dass liest die MasterNode diese und aktualisiert ihr StatusObjekt auf binärer Ebene (über einen u8-Pointer)

Wenn ein CAN-Nachricht „CHANGE\_STATUS“ am Bus anliegt, wird diese der Applikation nicht als „normales“ Command übergeben, sondern: Bei jedem CallMeRegularly bekommt die Applikation einen pointer auf einen Buffer und einen Pointer auf einen size\_t. Hiermit werden neue Statusinfos in die Applikation hineingeschrieben. Außerdem kann die Applikation den Buffer und den size\_t verwenden, um ggf. den neuen Status zurückzugeben.

Bei jedem Statuswechsel legt die Node den neuen Status auf den CAN-Bus. Die Masternode kann diesen Status abgreifen und abspeichern. Jede Veränderung des Status leitet der Masternode auch an das MQTT-Topic „Sensact::StatusChanged“ weiter. Die Masternode kann die folgenden MQTT-Nachrichten empfangen:

* GetAllStatus: Auf diese Nachricht sendet die Masternode den Status aller Applikationen auf das Topic Sensact::StatusInfo. Die Kodierung des Payload ist dabei uint16 count, item[count] items mit item={u16 ApplicationId, u8 statusLength, uint8[statusLength] payload}.
* GetStatusOf(n). Auf diese Nachricht sendet die Masternode den Status der gewünschten Applikation oder aber eine Fehlernachricht, falls die ApplicationId unbekannt ist.

Innerhalb der Applikation könnte der Status gar nicht gespeichert sein. Statt dessen castet d… NEIN, dann müsste jedem command das u8-array übergeben werden und es müsste ein cast stattfinden – das machen wir nicht – sooo eng ist der Speicher nicht!

Was passiert, wenn die Masternode kurz weg ist und sich dann der Status einer node ändert, wie gehen wir dann vor, damit die masternode das beim Hochfahren wieder mit bekommt.

* Regelmäßiges Senden des Status von allen Applicationen?
* Exlizites Abfragen

Die masternode muss sich merken, ob sie für eine App bereits einen Status empfangen hat.

Ich meine, es ist doch sinnvoll, in jeder app einen GetStatus und einen SetStatus als Command explizit zu implementieren. Wenn die Masternode hochfährt, dann fragt sie alle Applikationen nach ihrem aktuellen Status im 200msek-Rhythmus. Im Betrieb fragt sie reihum alle 2sek. Eine Veränderung des Status wird immer direkt durchgepushed. Fraglich ist, ob es hier zu raceconditions kommen könnte: Antwort. Nein, da eine später losgesendete Nachricht auch IMMER einen späteren Status hat.