Protocol v11 9-nov-2016

Stap-motor is neergezet als A. A heeft dus als waarde 1 t/m 4 maar zou verwarrend zijn in de communicatie-tabel. In ieder commando kan een X gebruikt worden die de informatie bevat van het commando. X wordt uitgelegd in kolom “Aanvullende informatie”

Achter ieder commando wordt 2 met 2 bytes (16 bit CRC) de CRC code van het commando meegegeven.  
Indien STM32 een niet kloppende CRC code ontdekt wordt direct geantwoord met 667:<crc fault> ipv met het antwoord uit tabel 1.

**Example CRC:**

**Commando vanuit client:** <y8:y1> **=> CRC functie => commando+CRC functie(commando)**

STM32 biedt de mogelijkheid om verloren commando’s opnieuw op te vragen. Indien een commando verloren is wordt geantwoord met 666:<cR> waarbij R het command id is van het command dat als eerste gestuurd dient te worden. Om deze functionaliteit te gebruiken moet check packet loss enabled worden met het eerste command dat gestuurd zal worden met: <eRA> waarbij R het eerste te sturen command id is en A een willekeurige stappen motor 1 tot 4.

**Tabel 1: Commando’s geinitieerd vanuit Windows**

|  |  |  |  |
| --- | --- | --- | --- |
| Doel van commando | Windows zenden | Verwacht Stm32 antwoord | Aanvullende informatie |
| Reset/ stop | <y8:yA> | <8:y># | Waarde van A doet er niet toe, wordt alleen meegestuurd omdat bij alle andere commando’s dezelfde decodering routine gebruikt wordt. |
| Enable check packet loss | <eRA> | <16:R> | Waarde van A doet er niet toe, wordt alleen meegestuurd omdat bij alle andere commando’s dezelfde decodering routine gebruikt wordt. R is het command id dat als eerste gestuurd zal worden. Standaard na hardware reset van STM32 staat controle op packet loss uit. |
| Sturen van verhouding tussen stap motor stappen en stap motor encoder stappen | <dXA> | <12:X># | X=steps\_per\_rpm divided by steps\_per\_rpm\_encoder  X is floating point en mag positief of negatief zijn (afhankelijk van checkbox voor draairichting encoder) |
| Sturen van maximaal hoeveelheid stappen die gemist mogen worden | <bXA> | <13:X># | X=max\_allowed\_miss is het maximaal aantal stappen verschil tussen gestuurd naar stap motor en gezien door encoder. X is een integer |
| Disable van de encoder als deze niet aanwezig is/ niet gebruikt wordt | <x0A> | <14:0># | Let op, de kleine x is het commando! 0 is disable, 1 is enable |
| Disable van de stappenmotor als deze niet aanwezig is/ niet gebruikt wordt. | <a0A> | <15:0># | Let op, de kleine a is het commando! 0 is disable, 1 is enable. |
| Sturen van positieve draairichting van stappen motor | <vXA> | <9:X># | X= draairichting, integer kan 0 of 1 zijn |
| Sturen van dwell time | <iXA> | <18:X># | X= dwelltime in seconds, float |
| Sturen van aantal stappen | <sXA> | <6:X># | X=aantal stappen, integer mag positief en negatief zijn. |
| Sturen van aantal stappen/omwenteling | <pXA> | <5:X># | X=aantal stappen per omwenteling. Integer moet positief zijn |
| Sturen van maximale omwenteling snelheid in RPM (Revolutions Per Minute) | <rXA> | <3:X># | X=maximale RPM, floating point, moet positief zijn. |
| Sturen van feedrate omwenteling snelheid in RPM (Revolutions Per Minute) | <gXA> | <17:X># | X= feedrate RPM, floating point, moet positief zijn. |
| Sturen van tijd in secondes waarin de stappen motor op maximale omwenteling snelheid moet zijn | <oXA> | <2:X># | X=tijd in secondes waarin stappen motor op snelheid moet zijn. Floating point. Moet positief zijn. |
| Enable/ disable stappen motor. Zet stroom op de stappen motor | <fXA> | <11:X># | X=enable/disable. Integer moet 0 of 1 zijn.0=disable, 1=enable |
| Informeer STM32 dat programma gereed is en dat gestart kan worden | <kFAULTA> | <0:rdy$R>0;# | Het woord “FAULT” komt terug van STM32 als niet alle benodigde parameters gestuurd zijn. De A na het woord FAULT doet er niet toe. STM32 zal “done:” sturen als de beweging gereed is. De R na het $ teken is het command\_id, integer, optioneel, alleen als gebufferde commando’s gestuurd zijn met een command\_id. |
| Stuur alleen een step upgrade. Wordt gebruikt als alle parameters al gestuurd zijn naar een stappen motor | <qXA> of  <qX;Y;Z;W;A> | <0:rdy$R>0;# | X,Y,Z,W=aantal stappen. Integer kan positief of negatief zijn. STM32 antwoord direct met 0:rdy. De R na het $ teken is het command\_id, integer, optioneel, alleen als gebufferde commando’s gestuurd zijn met een command\_id. |
| Stuur homing commando | <hXA> | <6:1># | X=homing richting. Integer kan 1 of 2 zijn. 1= homing op linker stopswitch. 2= homing op rechter stopswitch. STM32 start niet direct met homing. Hiervoor moet eerst <kFAULTA> gestuurd worden. |
| Vraag software versie op | <zA> | <14:version># | A doet er niet toe. |
| Onderhouden van de verbinding | <wA> | <wA>X;Y# of zie tabel 2 | <wA> moet alleen gestuurd worden terwijl op done: gewacht wordt. Bijv. In een timer routine.  Het antwoord van STM32 bevat in de X het aantal pending commands. Als er geen pending commands meer zijn en het finale commando verwerkt wordt , dan wordt X=-9999 gestuurd als teken dat pending commands 0 is maar nog wel het finale commando verwerkt wordt. De Y bevat het percentage van het huidige commando dat al verwerkt is. Hiermee kan bijv. Een status bar bijgewerkt worden. Bij homing is dit percentage steeds 0 (omdat onbekend is hoelang homing zal duren). |

**Tabel 2: Informatie geinitieerd vanuit STM32**

|  |  |  |
| --- | --- | --- |
| Doel van commando | Stm32 informatie | Aanvullende informatie |
| Informeren dat afhandelen van commando gereed of gestopt is | <wydone:X;Y@KZ;Q>R &# | Meestal zal vanuit Windows een <wA> gestuurd zijn om de verbinding te onderhouden. Als antwoord kan dus ook deze informatie terugkomen. Na afhandelen van het <wA> commando daarom controleren op “done:”  X = stopswitch die eventueel brand. Als geen stopswitch dan 0. Anders integer waarde 1 t/m 7.  Y= Aantal pending commands, integer  K=stappen motor die de maximale encoder slip veroorzaakte. Alleen ongelijk nul als er daadwerkelijk gestopt is vanwege teveel slip.  Z=Aantal gemiste stappen (encoder stappen motor), integer. Bij homing van de stappenmotor wordt de encoder niet bijgehouden. Als homing gereed is wordt in Z het aantal genomen homing stappen gegeven. (K is bij homing altijd gelijk 0)  Q= Ongedefinieerde status changes van encoder,integer. Q is bij homing altijd gelijk 0.  R= command\_id, integer, optioneel, alleen als gebufferde commando’s gestuurd zijn met een command\_id. |

<q32641>

**Sturen gebufferde commando’s**

Doel van gebufferde commando’s is de verwerkingssnelheid van stappenmotor bewegingen maximaal maken en geen tijd te verliezen tussen de bewegingen aan communicatie.

De commando’s uit tabel 1 kunnen al gestuurd worden terwijl STM32 bezig is een stappen-motor te bewegen. In dat geval wordt in plaats van <wA> het commando <c commando\_inhoud > gestuurd.

Bijv.

<cq1001>

Dit betekent dat stappenmotor 1 100 stappen moet uitvoeren. Als de buffer leeg is (geen pending commands) wordt dit direct uitgevoerd. Anders nadat voorgaande commando’s in de buffer verwerkt zijn. Gebufferde commando’s worden sequentieel uitgevoerd. Om te controleren of alle commando’s uitgevoerd worden kan een command\_id meegegeven worden in de vorm:

<c commando\_inhoud $ command\_id>

Bijv.

<cq1001$333>

Als het commando uiteindelijk wordt uitgevoerd zal STM32 bij het informeren dat de opdracht verwerkt is het command\_id meesturen. Zie tabel 2.

Als in plaats van het gebufferde commando het normale commando gestuurd wordt terwijl STM32 bezig is dan doet STM32 hier niks mee. Hij stuurt als antwoord wel het commando letterlijk terug net als het <wA> commando. De enige commando’s waar STM32 dus op reageert als hij bezig is een opdracht te verwerken zijn dus:

* Stop/reset
* Gebufferde commando’s

Als STM32 alle commando’s verwerkt heeft dan reageert hij natuurlijk weer wel op ongebufferde commando’s.

Het antwoord op een gebuffered commando is (let op start haakje voor 10: ontbreekt in antwoord)

10:c< q1001$333>Y;Z#

Waarbij Y is het aantal pending commands en Z is het percentage van het aantal te nemen stappen dat al genomen is van het huidige commando, net als bij w commando. (kan gebruikt worden om status bar mee te laten bewegen.) Indien homing is dit percentage 0.

**Gedrag bij stopswitch**

Als een stopswitch geraakt is dan wordt het gehele programma en gebufferde commando’s vergeten. Alle parameters uit tabel 1 zoals stappen per omwenteling, maximale snelheid etc. moeten opnieuw gestuurd worden.

**CRC routine in Delphi**

**VAR crctable16 : array [0..255] of uInt16;**

**const generator:uInt16 = $1021;**

**function CRC\_CCITT(aMsg:pointer; MsgLength:integer ): boolean;**

**VAR Index:byte;**

**pMsg : PByte;**

**crc:uint16;**

**i : Integer;**

**begin**

**crc := $FFFF;**

**pMsg := aMsg;**

**for I := 1 to MsgLength do**

**begin**

**//\* XOR-in next input byte into MSB of crc, that's our new intermediate divident \*/**

**Index := (crc SHR 8) XOR pMsg^;**

**INC(pMsg);**

**//\* Shift out the MSB used for division per lookuptable and XOR with the remainder \*/**

**crc := uInt16((crc SHL 8) XOR crctable16[Index]);**

**end;**

**pMsg^ :=Hi(crc);**

**inc(pMsg);**

**pMsg^ :=Lo(crc);**

**// If we are generating a CRC on a message that already includes a CRC, and if the message was received without error, the new CRC will be 0x00 and this fuction will return '0'. \*/**

**if crc=0 then**

**Result := true**

**else**

**Result := false;**

**end;**

**procedure generate\_table;**

**VAR**

**divident : integer;**

**curbyte : uInt16;**

**bit : byte;**

**begin**

**{$R-}**

**for divident := 0 to 255 do//\* iterate over all possible input byte values 0 - 255 \*/**

**begin**

**curByte := uInt16(divident) SHL 8; //\* move divident byte into MSB of 16Bit CRC \*/**

**for bit := 0 to 7 do**

**begin**

**if ((curByte and $8000) <> 0) then**

**begin**

**curByte := curByte SHL 1;**

**curByte := curByte XOR generator;**

**END**

**else**

**begin**

**curByte := curbyte SHL 1;**

**end;**

**end;**

**crctable16[divident] := curByte;**

**end;**

**{$R+}**

**end;**

**Initialization**

**generate\_table;**

**end.**