

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

| Markus Thill & Hendrik Boldt

1. Inhaltsverzeichnis

1. Inhaltsverzeichnis.....	3
Danksagung	6
2. Einführung.....	7
Motivation.....	7
Aufgabenstellung und Zielsetzung	7
Arbeitsschritte.....	9
3. Konzept zur Realisierung.....	10
MOSES.....	10
SPSIni:.....	10
SPSAus:.....	10
SPSEin:.....	11
InitFIFO:.....	11
CountFIFO:.....	11
receiveCheck:	11
Speicherprogrammierbare Steuerung.....	11
4. Einführung MOSES-Programm	12
Übersicht der relevanten Funktionen	12
SPSIni.....	13
initFIFO	13
SPSAus	13
Parameter.....	13
Rückgabewert (X-Register).....	13
SPSEin.....	14
receiveCheck	14
5. Spezifikation der Schnittstelle	15
Grundlagen.....	15
Vorgaben.....	15
Pegelanpassung.....	15
Periodendauer.....	16
Initiierung eines Kommunikationsvorganges	17
Sendemodus.....	17
Empfangsmodus.....	17
Signalisierung der Empfangsbereitschaft	17

Festlegung des Zeichenrahmens	18
Ablauf einer Übertragung.....	19
Besonderheit des Wait- und Final-Bits	20
Berechnung der Parität	20
Mögliche Übertragungsfehler und deren Behandlung.....	22
Daten-Byte falsch gelesen (Empfangsfehler)	22
Falsches Paritäts-Bit gelesen (Empfangsfehler)	22
Paritätsprüfung fehlgeschlagen (Empfangsfehler)	22
Start-Bit nicht erkannt (Empfangsfehler)	23
Falsches Stop-Bit gelesen (Empfangsfehler)	23
Wechsel in den Empfangsmodus wird nicht erkannt (Sendefehler)	23
Verlassen des Empfangsmodus wird nicht erkannt (Sendefehler).....	24
Flanken während der Übertragung werden nicht erkannt (Empfangsfehler)	24
Empfänger verlässt Empfangsmodus zu früh	24
Sender verlässt Sendemodus zu früh	25
Negative Quittierung	25
Falsches ACK-Bit gelesen (Sendefehler)	25
Empfänger wechselt nicht in Empfangsmodus	25
Leitung TxD des Senders führt falschen Spannungspegel während Wait- und Final-Bit	26
Vollständig belegtes Empfangs-FIFO	26
Mögliche Optimierung der Schnittstellendefinition.....	26
6. Funktionsweise der FIFO	27
Allgemein	27
Vergleich zwischen SPS und MOSES.....	27
MOSES	27
SPS	27
7. MOSES-Programm	28
Verwendung der Bibliothek	28
Grundsätzliches	28
Subroutinen.....	28
Benötigte PIAT-Ressourcen	28
Weitere Ressourcen	29
Verwendung von Interrupts im eigenen Anwenderprogramm	29
Verwendung des IRQ-A	29
Verwendung des IRQ-T.....	29

Zusätzliche Informationen.....	30
Parallele Verwendung der Zusatzleitung am MOSES?	30
Blockierung von Interrupts.....	30
Keine Subroutine für den Empfangsmodus.....	30
Kommunikation zwischen Subroutinen / IRQ-Behandlungsroutine.....	30
Realisierung der Swap-Time	30
8. SPS Programm	31
Programmstruktur.....	31
OB1.....	31
ComMOSES.....	32
RcvMOSES	33
SndMOSES	34
Beschreibung der wichtigsten Funktionen, Funktions- und Organisationsbausteine	34
Funktion einer Schrittfolge	34
FB – ReceiveMOS.....	35
FB – SendMOS	35
FC – ReadFrameBit	35
FC-RecvMos.....	35
FC – SendData	35
OB-40 Prozessalarm	35
9. Oberfläche WinCC	36
Freigaben und Statusmeldungen	37
Empfangs- und Sende-FIFO der SPS	37
Fehlermeldungen und Resetfunktion.....	38
10. C++-Programm	39
Grundsätzliches	39
Funktionen des Testprogramms.....	40
11. Tests	41
Übertragungsgeschwindigkeiten.....	41
MOSES zu MOSES	41
SPS zu SPS	41
Überprüfung der Schnittstellenfunktionalitäten.....	42
Durchführung der Tests.....	42
Empfang: Allgemeiner Test	43
Die letzten Schritte einer Übertragung	45

Empfang: falsche Parität	46
Empfang: Fehlerhaftes Stop-Bit	48
Empfang: Falsches Startbit	49
Empfang: Vollständig belegtes Empfangs-FIFO	50
Senden: Allgemeiner Test.....	51
Senden: Verpasstes / Ignoriertes Start-Bit.....	53
Senden: Frühzeitiges Verlassen des Empfangsmodus.....	54
Senden: Negative Quittierung des Empfängers	56
Senden / Empfangen: Test des Swap-Bits	57
12. Anhang	59
Beispielanwendungen für MOSES und SPS	59
Beispiel für ein Sendeprogramm.....	59
Beispiel für ein Empfangsprogramm	62
Sendevorgänge mit Fehlerbehandlung	64
Übertragung der aktuellen Schalterstellung am MOSES.....	66
Übertragung von Datenfeldern	68
Verwendung von Sende-FIFOs	71
Komplexere FIFO-Struktur.....	75
Auswertung von bool'schen Funktionen mithilfe der SPS.....	96
MOSES - Programm	103
Quelltext.....	103
Visual-C++ Quelltext.....	149
13. Abbildungsverzeichnis.....	194

Danksagung

Wir möchten uns bei all den Personen bedanken, die uns bei unserem Projekt unterstützt haben. Besonderer Dank gilt dem wissenschaftlichen Mitarbeiter Frank Schäfer für die unermüdliche Unterstützung. Bei der Entwicklung des SPS-Programmes konnten wir auf Programmteile zurückgreifen, die von Herrn Schäfer entwickelt worden sind. Ohne diese Programmteile hätte das Projekt nicht diesen ausführlichen Umfang erreicht.

2. Einführung

Motivation

Im Labor für Rechnerarchitektur der technischen Informatik gibt es 6 Mikrorechner Arbeitsplätze. Die Mikrorechner basieren auf einem 6502 Prozessor und werden in Assembler programmiert. Diese µP-Systeme, kurz MOSES genannt, werden bei den Praktika im Labor verwendet. Neben diesen Mikrorechnern verfügt das Labor noch über eine "Speicherprogrammierbare Steuerung" (SPS) vom Typ Simantic S7 – 300. Um die SPS mit ins Praktikum zu integrieren, soll eine Möglichkeit zum Datenaustausch zwischen den beiden Systemen geschaffen werden. Da der MOSES in dieser Hinsicht nur über sehr eingeschränkte digitale Ressourcen verfügt, ist die Schnittstelle als Bit-Serielle Schnittstelle in Anlehnung an RS 232C zu realisieren. Die Schnittstelle ist für "Halb Duplex" Betrieb auszulegen, also Senden und Empfangen müssen nicht gleichzeitig realisiert werden!

Aufgabenstellung und Zielsetzung

Realisierung einer Schnittstelle (halb Duplex) im heterogenen Verbund zwischen einer Simatic S7-300 SPS und einer Mikroprozessorsteuerung (MOSES).

*Die Schnittstelle ist nach Möglichkeit ohne zusätzliche Takt-Leitung(en) zu realisieren!
(Asynchron Serielle Schnittstelle mit fest definiertem Zeichenrahmen und Baud Rate)*

Beispielhafte Realisierung:

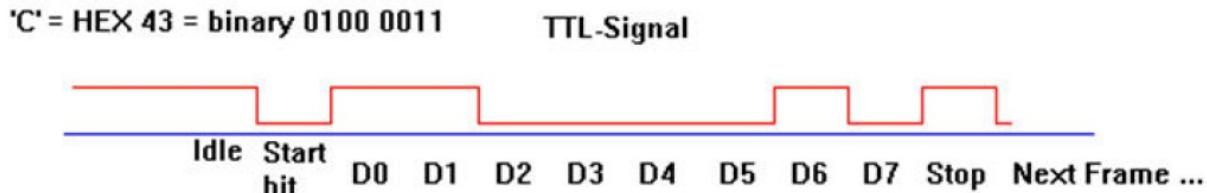


Abbildung 1 Beispiel einer Realisierung

Beim Moses sollen die Schnittstellenprogramme via Library, wie auch z.B. DigAus, in das Anwenderprogramm eingebunden werden können.

Nach Aufruf des Sendeprogramms ("JSR SendSPS") soll dann das im AKKU stehende Byte an die SPS gesendet werden. Anschließend wird das Sendeprogramm mit einer im AKKU stehenden positiven oder negativen Empfangsquittung verlassen.

Beim Moses wird der Port B für diese Schnittstelle benutzt, und zwar CB2 als Sendeleitung zur SPS und CB1 als Empfangsleitung von der SPS.

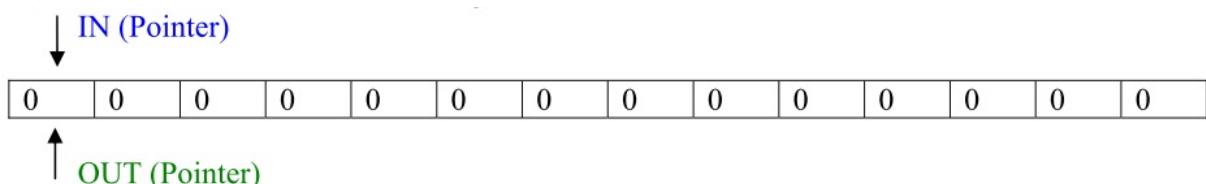
D.h., die fallende Flanke des Start-Bits löst einen Interrupt beim Kommunikationspartner aus und startet dort die definierte Abtastzeit von z.B. 20ms. Der Sender sendet in diesem Zeitraster dann simultan die Einzelbits des Bytes, bzw. des Zeichenrahmens (Frames).

Der Empfänger quittiert den Empfang des Bytes immer mit einer positiven (ACK) oder negativen (NACK) Empfangsquittung. Diese muss innerhalb eines definierten Time Out's empfangen werden, ansonsten interpretiert der Sender dies als "not connected"

Für den Empfang soll auf beiden Seiten ein Empfangsspeicher als Ringpuffer von ca. 100 Byte fest definiert werden. Diese FIFO's sollen nach dem Prinzip der umlaufenden Zeiger realisiert werden (d.h. die Zeiger dürfen sich gegenseitig nicht überholen). Sind die Zeiger gleich, ist das Empfangs-FIFO leer. Für das Auslesen der FIFO's wird jeweils ein Standard-Unterprogramm geschrieben. (ReadBuf). Nach JSR ReadBuf (beim Moses) steht im Akku die empfangene Date aus dem FIFO. War das FIFO leer, so wird dies gemeldet. Beim Moses z.B. durch eine Kennung im X-oder Y-Register. Ist das FIFO voll, werden keine Daten vom Partner mehr angenommen. In diesem Fall wird ebenfalls eine negative Empfangsquittung (NAK2) an den Sender zurückgemeldet. Darüber hinaus soll im MOSES noch ein Unterprogramm "InitBuf" zur Verfügung gestellt werden, mit dem jederzeit das FIFO gelöscht und initialisiert werden kann.

Beispiel einer FIFO:

(Startzustand nach der Initialisierung)



Arbeitsschritte

- Erstellung eines Pflichten- bzw. Lastenheftes.
- Realisierung der Pegelanpassung mit vorhandenen Optokopplern.
- Anfertigung eines einfachen Hardwareplans (Stromlaufplan) der Schnittstelle.
- Erstellung der Schnittstellenspezifikationen.
- Softwareentwicklung für die µP-Steuerung.
- Softwareentwicklung für die SPS.
- Funktionstest
- Dokumentationserstellung für die Präsentation und die Bedienung der Schnittstellenbausteine (Library).
- Inbetriebnahme

3. Konzept zur Realisierung

MOSES

Verwendete Unterprogramme:

- SPSIni: Initialisierung des IRQT und CB1 / CB2
- SPAus: Ausgabe des Bytes aus dem Akku
- SPSEin: Empfang eines Bytes von der SPS
- initFIFO: Initialisierung des Empfangs-FIFO's
- countFIFO: Zählt die Anzahl der belegten Bytes
- receiveCheck Temporäre Empfangsfreigabe

SPSIni:

Muss zur Voraussetzung der Kommunikation einmal aufgerufen werden.

Hier erfolgt die Initialisierung des Timers mit 20ms in der Betriebsart 2. Vorherige Initialisierungen werden überschrieben.

CB2 wird als Ausgang parametriert und auf 1 gesetzt (Idle). Für C1 wird der Interrupt für fallende Flanke freigegeben. Der Empfang kann anhand eines Parameters blockiert werden.

SPAus:

Das im Akku übergebene Byte wird zuerst in den 11 Bit breiten Zeichenrahmen konvertiert und für IRQT bereitgestellt. Danach wird die Ausgabeanforderung für IRQT gesetzt. Mit jedem IRQT wird dann (also alle 20ms) ein Bit-Status ausgegeben. Sind alle Bits gesendet, wird im IRQT die Anforderung zurückgesetzt und dadurch SPAus signalisiert, dass die Übertragung beendet ist. Jetzt muss innerhalb eines WaitBits und FinalBits ein IRQ an CB1 ausgelöst werden (Empfangsquittung). Ansonsten wird SPAus mit negativer Sendequittung im Akku verlassen. Ein nach dem FinalBit ausgelöster Empfangsinterrupt wird dann als normaler Empfang von der SPS interpretiert und das Zeichen nach der Prüfung und Konvertierung in den Empfangspuffer geschrieben (FIFO).

Nach dem Empfangsinterrupt muss zuerst nach $T/2 = 10\text{ms}$ ein Timer IRQ ausgelöst werden, zum Einlesen des Start Bits. Danach dann alle 20ms.

Treten nach der Empfangsinitialisierung weitere Hardware IRQ's an CB1 auf, wird die Übertragung abgebrochen. Nach einer abgeschlossenen Kommunikation wird dem Empfänger die Möglichkeit gegeben eine Übertragung zu starten. Hierfür ist das SwapBit zuständig.

Jetzt wird in Idle gewechselt bis eine neue Kommunikation gestartet wird. Die Zeit für einen Zeichenrahmen mit 14 Bits ist durch die Baudrate definiert.

$$T_{\text{ges.}} = 14 * 20\text{ms} = 280\text{ms}$$

Datenübertragungsrate = 50 Bits/s

Da bei binären Übertragungen Bitrate = Baudrate entspricht gilt:

Die Baudrate entspricht hier also 50 Baud

SPSEin:

Im Grunde prüft dieses Programm nur, ob Empfangsdaten im Empfangs FIFO stehen und gibt diese über den Akku aus. Wenn Empfangspuffer leer, dann Akku = \$FF.
Alles weitere findet in den IRQ Routinen statt.

InitFIFO:

Die 2 umlaufenden Zeiger der FIFO werden gesetzt. Mit der Konstanten fifoSize kann die Größe der FIFO bestimmt werden.

CountFIFO:

Zählt die Anzahl der belegten Bytes in der FIFO und gibt das Ergebnis im Akkumulator zurück.
Es werden keine Veränderungen der FIFO durchgeführt, Interrupts müssen hier also nicht gesperrt werden!

receiveCheck:

Mit dieser Subroutine kann bei Bedarf kurzzeitig der SPS die Sendemöglichkeit bzw. Sendefreigabe gegeben werden. Nach Verlassen des Unterprogramms ist immer nur das Senden an die SPS freigegeben und initialisiert.

Speicherprogrammierbare Steuerung

Für das SPS-Programm sind keine besonderen Vorgaben an die einzelnen Funktionen und Funktionsbausteinen definiert worden. Jedoch ist darauf zu achten, dass das Programm möglichst modular gestaltet wird, um ein gewisses Maß an Übersichtlichkeit sowie Wiederverwendbarkeit zu erhalten. Folgende Grundfunktionalitäten sollten unabhängig voneinander und ohne spezieller Abhängigkeiten zueinander implementiert werden:

- Kommunikation mit dem Mikrorechner MOSES, weiter unterteilt in
 - Funktion oder Funktionsbaustein Senden
 - Funktion oder Funktionsbaustein Empfang
- Generelle Fehlerbehandlung
- FIFO - Struktur
 - Initialisierungs- und Verwaltungsroutinen
 - Auf Wiederverwendbarkeit achten, da Sende- und Empfangs-FIFO benötigt
- Unabhängige Bausteine zur Organisation der Hardware-Baugruppen (z.B. zur Behandlung von IRQs und Ausgängen)
- Schnittstellen zwischen Kommunikationsbausteinen und FIFO-Strukturen
- Visualisierung mit WinCC (nicht zwingend nötig)

Einige Funktionen lassen sich sehr gut mithilfe von Schrittketten realisieren. Dort wo dies möglich ist, sollten Schrittketten zur Verbesserung der Übersicht eingesetzt werden.

4. Einführung MOSES-Programm

Übersicht der relevanten Funktionen

Nachfolgend eine kleine Übersicht, über das Aufrufen der verschiedenen Funktionen. Hierbei handelt es sich um eine schematische Darstellung, um die mögliche Reihenfolge der Funktionsaufrufe zu verdeutlichen.

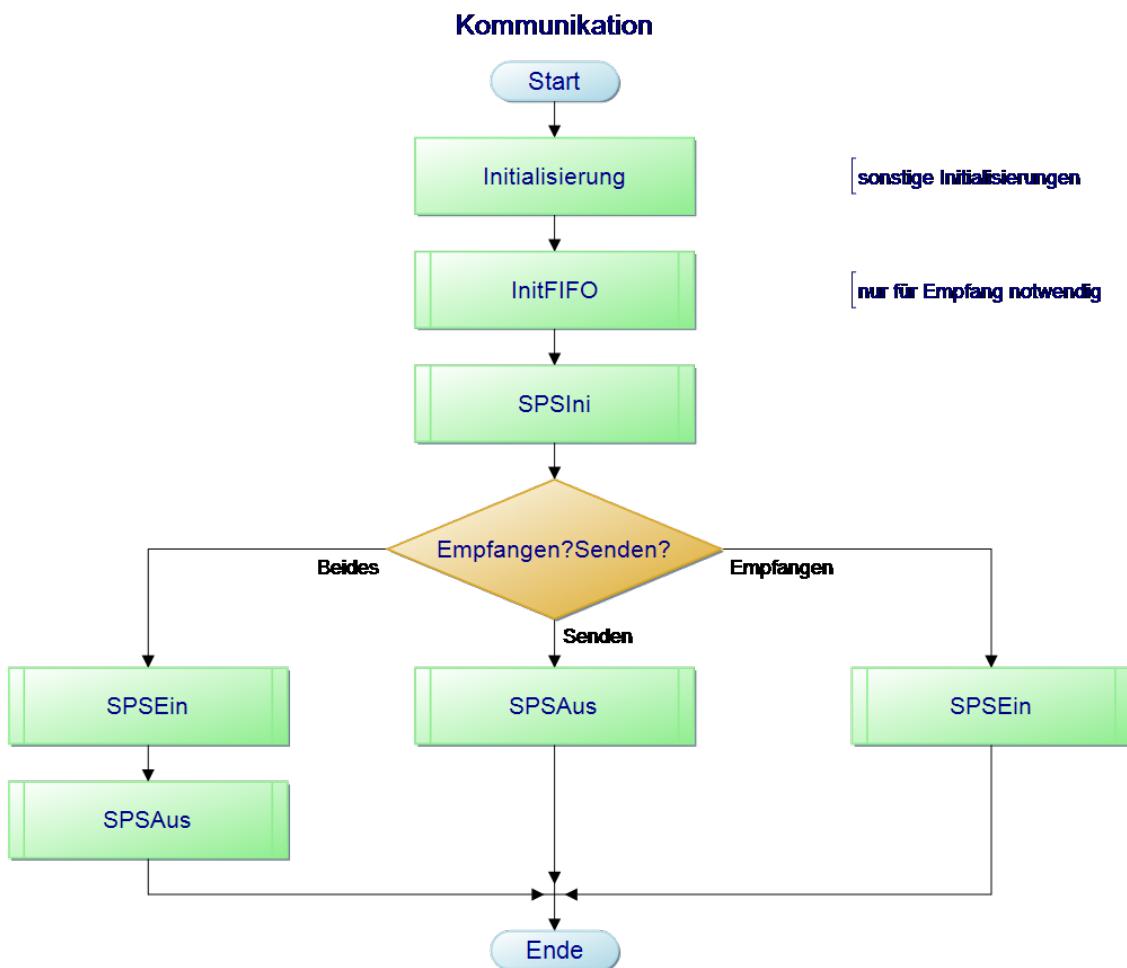


Abbildung 2 Grafische Darstellung des Programmablaufes des Moses

Beachte: Hierbei handelt es sich nicht um einen Programmablaufplan im klassischen Sinne, sondern lediglich um eine Übersicht über die mögliche Reihenfolge der Funktionsaufrufe im Anwenderprogramm.

SPSIni

Wird zum Initialisieren der Kommunikation verwendet.

Parameter

Der Parameter wird im Akkumulator übergeben.

- | | |
|---------|--|
| 0 | -> Löschen aller Einstellungen der seriellen Schnittstelle |
| 1 | -> Senden erlauben / jedoch den Empfang verbieten |
| sonst | -> Senden sowie Empfang erlauben |
| MSB = 1 | -> Verwendung von Leitung 0 des PORT B als Eingang |

Der Eingang CB1 ist ein Flankengesteuerter Eingang. Es ist daher nicht möglich den aktuellen Zustand des Einganges zu lesen. In manchen Fällen kann es passieren das eine Flanke nicht erkannt worden ist. Daher ist es möglich, dass zusätzlich die Leitung 0 des PORT B als Eingang verwendet wird. So erhält man eine höhere Sicherheit, dass der richtige Zustand eingelesen wird. Gesteuert wird das zu und abschalten der Leitung 0 über das MSB.

Rückgabewert

Keiner

initFIFO

Initialisiert die FIFO mit den Startwerten für die umlaufenden Zeiger.

Parameter

Keine

Die FIFO-Größe kann nicht dynamisch geändert werden, sondern muss vor der Assemblierung im Quellcode festgelegt werden. Die maximale Größe ist jedoch auf 255 Bytes beschränkt. Die Konstante fifosize kann in der Library ggf. angepasst werden.

Rückgabewert

Keiner

SPSAus

Sendet ein Byte an den Kommunikationspartner

Parameter

Senden eines Bytes

Parameter:

- | | |
|----------------|--------------------------------------|
| im Akku: | Datenbyte |
| im X-Register: | Warte auf KP(0), Nicht warten(sonst) |

Rückgabewert (X-Register)

- | | |
|---|---|
| 0 | -> kein Fehler |
| 1 | -> Fehler vom Empfänger erkannt bzw. vorzeitige Beendigung
des Sendevorganges (Empfänger wechselt zu früh aus Empfangsmodus) |
| 2 | -> Noch selbst im Empfangsmodus |
| 3 | -> Empfänger nicht empfangsbereit (es wird nicht gewartet) |

SPSEin

Liest einen Wert aus der FIFO und schreibt den Wert in den Akku. Fehlerkennungen werden in das X-Register geschrieben.

Parameter

Keine

Rückgabewert (X-Register)

- | | |
|---|---|
| 0 | Konnte Wert erfolgreich aus der FIFO lesen |
| 1 | FIFO ist leer, konnte keinen Wert entnehmen |

receiveCheck

Diese Subroutine prüft, ob die SPS ein Datenbyte übertragen möchte und empfängt dieses ggfs. Der Durchlauf der Subroutine kann bis zu 500ms dauern. Daher sollte diese Routine nur bei Bedarf verwendet werden.

Parameter

Keine

Rückgabewert

Keiner

Nach Aufruf der Subroutine wird intern Senden und Empfang (von der SPS) freigegeben. Danach wird eine Zeit (<500ms) gewartet und der SPS die Übertragung eines Bytes ermöglicht. Sollte die SPS innerhalb dieser Zeit Daten übertragen, so werden diese im Empfangs-FIFO eingetragen. Vor dem Verlassen des Unterprogramms wird der Empfang von der SPS wieder gesperrt. Das Senden an die SPS bleibt initialisiert, d.h. SPSAus kann jederzeit weiterhin aufgerufen werden.

Im Anschluss an den Aufruf von "receiveCheck" kann dann mittels "SPSEin" auf das Empfangs-FIFO zugegriffen werden.

5. Spezifikation der Schnittstelle

Grundlagen

Vorgaben

Für die detaillierte Spezifikation der seriellen Schnittstelle sind zunächst folgende Vorgaben zu beachten:

1. Übertragung erfolgt Bit-seriell
2. Ein Übertragungsvorgang umfasst ein Datenbyte
3. Halb-Duplex Betrieb
4. Nach Möglichkeit nur zwei Kommunikationsleitungen -> Asynchronbetrieb ohne Taktleitung
5. Verwendung von Start- und Stop-Bit
6. Verwendung eines Paritätsbits zur Fehlererkennung
7. Quittierung durch Empfänger
8. Fest definierter Zeichenrahmen und Baudrate (die Baudraten werden an beiden Kommunikationsschnittstellen auf den gleichen Wert eingestellt, die Baudrate selbst sollte leicht geändert werden können)
9. Die Geräte auf beiden Seiten des Kommunikationskanals sind nicht vorgeschrieben (z.B. möglich: MOSES - SPS, MOSES - MOSES, MOSES - PC, etc.)

Pegelanpassung

Da der PIAT-Baustein des MOSES, die Ausgangsbaugruppen der SPS S7-300 und die serielle Schnittstelle des PCs allesamt unterschiedliche Spannungsspegel zur Repräsentation der Zustände High und Low verwenden, müssen entsprechende Pegelanpassungen vorgenommen werden. Für eine Kommunikation zwischen MOSES und SPS wird eine Pegelanpassung am sichersten mit Hilfe Optokoppler bewerkstelligt.

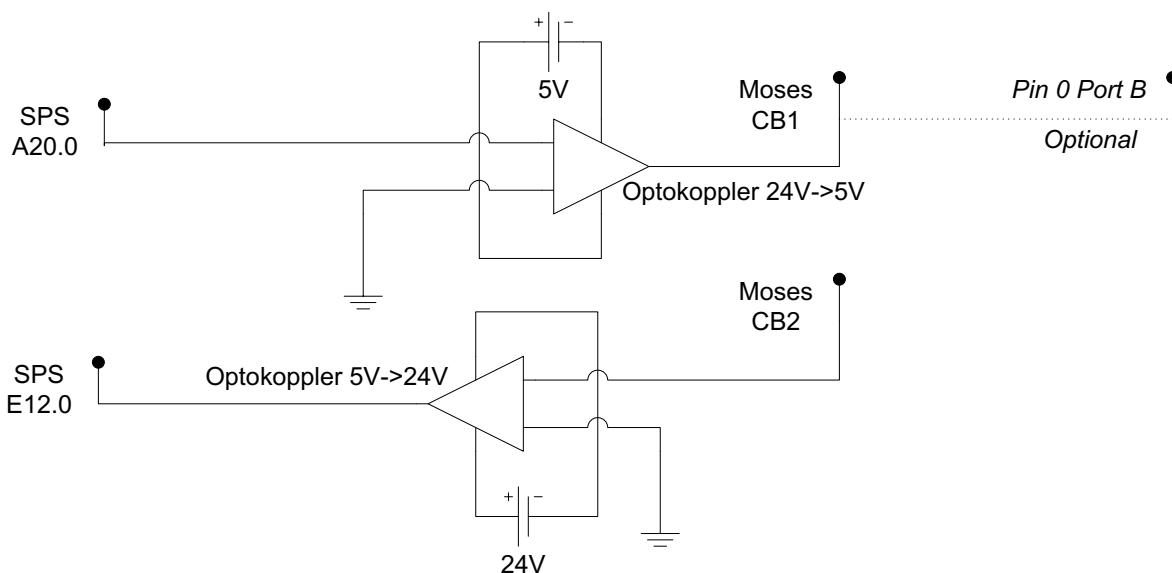


Abbildung 3 Verdrahtungsplan SPS - Moses

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

Für eine Kommunikation zwischen PC (RS232) und MOSES wird ein TTL-RS232-Wandlerbaustein verwendet (weiter unten diskutiert).

Die letzte mögliche Pegelanpassung, zwischen PC und SPS, wurde nicht realisiert, jedoch sollte dies auch mit Hilfe von Optokopplern möglich sein.

Im Verdrahtungsplan gibt es eine Optionale Verbindung. Diese wird benötigt wenn besondere Anforderungen an die Sicherheit gestellt werden. Der Moses kann mit Hilfe der zusätzlichen Leitung den aktuellen Status von seinem Eingang lesen.

Periodendauer

Die Periodendauer ist die kleinstmögliche Zeiteinheit während einer Übertragung. Hierdurch werden die weiter unten erläuterten Zustandsbits (Startbit, Datenbits, Stopbit etc.) definiert. Während einer Periode sind die Spannungspegel auf der Sendeleitung eines Kommunikationspartners konstant. Dies ist für eine erfolgreiche Kommunikation absolut notwendig. Nach dem Ablauf einer Periode können Sender und Empfänger das jeweils nächste Zustandsbit auf ihre Sendeleitungen schreiben. Durch eine Verschiebung der Timer beider Kommunikationspartner um eine halbe Periodendauer wird gewährleistet, dass die aktuellen Zustände vom Gegenüber korrekt eingelesen werden können. Für die Kommunikation zwischen MOSES und SPS hat sich ein Wert von 20ms für eine Periode als sinnvoll erwiesen (deutlich kleinere Werte sind aufgrund der ständig wechselnden Zykluszeiten der SPS nicht zu empfehlen).

Initiierung eines Kommunikationsvorganges

Hat einer der beiden Kommunikationspartner ein Datenbyte, das an den Gegenüber übertragen werden soll, so wird dieser versuchen einen Sendevorgang zu initiieren.

Voraussetzungen aus Sicht des Senders:

- Die Empfangsleitung muss im Zustand Idle sein (logisch 1 -> kann als grundsätzliche empfangsbereitschaft interpretiert werden)
- Die Sendeleitung muss auch im Zustand Idle sein, damit überhaupt eine fallende Flanke möglich ist
- Der potentielle Empfänger darf selbst nicht noch in einem Empfangs- bzw. Sendemodus sein (kann durch die Verschiebung beider Kommunikationspartner um die halbe Periodendauer durchaus relevant sein) -> Das Hängt eng mit dem ersten Stichpunkt zusammen
- Es darf kein Empfangsvorgang aktiv sein (Diese laufen z.B. beim MOSES durch Timer-IRQs im Prinzip im Hintergrund)

Sind die Voraussetzungen erfüllt, so muss beim Gegenüber mit einer fallenden Flanke auf die geplante Übertragung aufmerksam gemacht werden (Start-Bit). Im Regelfall wird der Empfänger auf das Startbit reagieren und in den Empfangsmodus wechseln. Dies ist aber nicht zwangsläufig der Fall. Ist beispielsweise der Empfangspuffer des Gegenübers vollständig belegt, so ist es sinnvoll, wenn dieser nicht in den Empfangsmodus wechselt würde. Eine andere Möglichkeit wäre in den Empfangsmodus zu wechseln und anschließend eine negative Quittung an den Sender zu übermitteln.

Sendemodus

Nach Initiierung des Sendevorganges (s.o.) befindet sich der entsprechende Kommunikationspartner im Sendemodus. In diesem befindet sich der Sender solange die Übertragung noch nicht abgeschlossen und der Gegenüber im Empfangsmodus ist.

Empfangsmodus

Wird das Startbit des Senders auf der Empfangsleitung des Gegenübers erkannt und setzt dieser daraufhin seine Sendeleitung auf logisch 0, so ist er in den Empfangsmodus gewechselt. Wichtig ist die Einhaltung der Reihenfolge der Vorgänge. Ein Wechsel in den Empfangsmodus bevor das Startbit überhaupt erkannt wurde (auch wenn ein Übertragungsvorgang erwartet wird) ist nicht möglich und auch nicht sinnvoll.

Selbstverständlich darf auch nicht in den Empfangsmodus gewechselt werden, wenn bereits im Sendebetrieb gearbeitet wird.

Signalisierung der Empfangsbereitschaft

Es gibt zwei Möglichkeiten um die Empfangsbereitschaft (Empfangsfreigabe) zu signalisieren:

1. Die grundsätzliche Empfangsbereitschaft wird durch den Zustand logisch 1 auf der Sendeleitung signalisiert. Dementsprechend kann die Empfangsfreigabe durch den Zustand logisch 0 wieder zurückgenommen werden. Dies hat einen Nachteil: Durch das Entfernen der Empfangsfreigabe wird beim Kommunikationspartner ein IRQ ausgelöst (da fallende Flanke) und ggfs. in den Empfangsmodus gewechselt, obwohl dies nicht beabsichtigt ist. Im Regelfall wird der Empfangsvorgang fehlschlagen, sodass keine weiteren Auswirkungen zu erwarten sind, ganz ausgeschlossen sind diese jedoch nicht. Sollte im Programm an vielen Stellen die Empfangsfreigabe entfernt und anschließend wieder gesetzt werden, so ist von diesem Verfahren abzuraten. Im MOSES-Programm wird daher die folgende Variante verwendet.

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

2. Ist keine Empfangsfreigabe vorhanden, wird bei einer Sendeanfrage (fallende Flanke auf der Empfangsleitung) nicht in den Empfangsmodus gewechselt, die Sendeleitung verbleibt im Zustand logisch 1. Dies kann der Kommunikationspartner erkennen und den Sendevorgang vorzeitig beenden. Nach einer gewissen Zeit kann erneut versucht werden einen Sendevorgang zu initiieren. Die Wartezeit zwischen zwei Sendever suchen sollte jedoch ausreichend groß gewählt werden um zu viele Signalwechsel auf der Sendeleitung zu vermeiden (kann bei den rein flankengesteuerten Empfangsleitungen des MOSES problematisch werden).

Ist die Empfangsleitung des Senders im Zustand Idle, so kann dies als grundsätzliche Empfangsbereitschaft des Gegenübers interpretiert werden. Wechselt der Gegenüber nach dem Startbit auch wirklich in den Empfangsmodus, so kann dies als tatsächliche Empfangsbereitschaft aufgefasst werden (vgl. notwendige und hinreichende Bedingung).

Festlegung des Zeichenrahmens

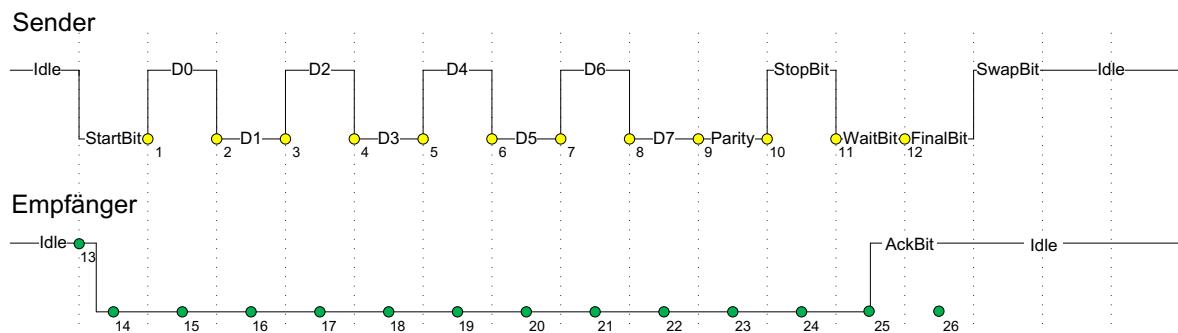


Abbildung 4 Zeitablauf einer Übertragung

In der obigen Grafik ist ein Übertragungsvorgang für ein vorgegebenes Datenbyte (01010101) über die Zeit grafisch dargestellt. Es werden jeweils die Sendeleitungen beider Kommunikationspartner betrachtet. Einzelne Zeitpunkte sind mit Kreisen markiert und werden weiter unten betrachtet.

- Idle: Kein Kommunikationsvorgang aktiv. Beide Seiten zeigen eine grundsätzliche Empfangsbereitschaft an.
- StartBit: Der Sender teilt dem Empfänger über eine fallende Flanke mit, dass eine Übertragung gestartet werden soll.
Der Empfänger sollte darauf mit einer fallenden Flanke und anstehendem 0 Signal antworten.
- D0 - D7: Das eigentliche Datenbyte. Beachte: Es wird zuerst das LSB (Least significant Bit) übertragen.
- Parity: Das Parity-Bit ergibt sich aus der ungeraden Parität der übertragenen Datenbits (D0-D7) und dient der späteren Fehlererkennung.

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

- StopBit: Über eine steigende Flanke wird mitgeteilt, dass die Übertagung beendet ist. Dieses Bit, ist für die Behandlung bestimmter Fehler / Störfälle wichtig (s.u.)
- WaitBit: Dem Empfänger wird eine Zeitperiode eingeräumt, um die empfangen Daten zu prüfen (Parität, Start- bzw. Stop-Bit) und anschließend in den Empfangs-FIFO zu schreiben.
- FinalBit: Nach der Quittierung des Empfängers muss ein wenig gewartet werden damit die zeitliche Verschiebung zwischen Sender und Empfänger ausgeglichen werden kann. Dadurch wird verhindert, dass der Sender sofort mit einer neuen Übertragung beginnt.
- AckBit: Das Acknowledge-Bit dient der Quittierung der empfangen Daten durch den Empfänger. War die Übertragung aus Sicht des Empfängers erfolgreich, so ist das AckBit logisch 1, ansonsten logisch 0.
- SwapBit: Das SwapBit wird benötigt, damit der ursprüngliche Empfänger zum Sender werden kann. Der Sender wechselt zu diesem Zeitpunkt bereits vom Sendemodus in den Zustand Idle. Jedoch kann während des SwapBits kein neuer Sendevorgang durch diesen initiiert werden. Dadurch wird gewährleistet, dass der ursprüngliche Empfänger in den Sendemodus wechseln kann, falls zu übertragende Daten vorhanden sind. Wenn beide Kommunikationspartner viele Daten übertragen müssen, kann eine Übertragung in beide Richtungen stets abwechselnd erfolgen (gleichberechtigte Kommunikationspartner). Im Gegensatz zu den anderen Bits, muss das SwapBit nicht eine volle Periodendauer anhalten (nämlich genau dann, wenn ein Wechsel stattfindet).

Beachte: Lediglich die acht Datenbits sowie das Paritäts-Bit unterscheiden sich bei den einzelnen Übertragungsvorgängen. Ändern sich die anderen Bits / Zustände, so führt dies zwangsläufig zu einem Fehler.

Ablauf einer Übertragung

Beachte: Grüne Punkte in der Grafik beschreiben Zeitpunkte an den Aktionen des Empfängers durchgeführt werden, gelbe dementsprechend Zeitpunkte an den Aktionen des Sender stattfinden.

Wie man gut sehen kann, sind die gelben und grünen Punkte jeweils um eine halbe Periode verschoben.

- 13: Sender löst beim Empfänger einen IRQ durch eine fallende Flanke aus. Der Empfänger wechselt anschließend in den Empfangsmodus.
 - 14: Das StartBit wird durch den Empfänger ausgelesen. Dies ist nicht zwingend notwendig, erhöht aber die Sicherheit ein wenig, weil ausgeschlossen werden kann, dass die erkannte fallende Flanke nur ein Störimpuls oder Ähnliches war.
- 1 - 11: Der Sender liest seine Empfangsleitung ein und prüft ob der Empfänger tatsächlich in den Empfangsmodus gewechselt ist (Leitung im Zustand logisch 0).
- 1 - 8: Die einzelnen Datenbits werden in umgekehrter Reihenfolge auf die Leitung geschrieben.
- 9 - 12: Die oben erläuterten Bits werden vom Sender geschrieben

- 15 - 22: Der Empfänger liest die übertragenen Datenbits vom Sender ein.
- 23: Der Empfänger bekommt das Paritäts-Bit.
- 24: Der Empfänger liest das StopBit, die Übertragung des Senders ist damit beendet. Zu diesem Zeitpunkt muss der Empfänger aktiv werden und die empfangen Daten prüfen (StartBit, Parität und StopBit).
- 25: Nach der Validierung der empfangen Daten wird das berechnete AckBit (Erfolg aus Sicht des Empfängers) geschrieben.
- 12: Der Sender liest das AckBit ein und kann somit auf den Erfolg der Übertragung schließen.
- 26: Der Empfänger wechselt vom Empfangsmodus wieder in den Zustand Idle. Spätestens zu diesem Zeitpunkt wird dessen Leitung also wieder in den Zustand logisch 1 wechseln.

Anschließend wechselt der Sender in den Idle Modus und wartet dann einen Moment ab (SwapBit). Hiernach kann der Sender einen neuen Übertragungsvorgang beginnen.

Beachte: Aufgrund der Synchronisierung zwischen Sender und Empfänger sind die Zahl der Perioden bis zum Abschluss der Übertragung für beide unterschiedlich. Der Empfänger benötigt genau 13 Perioden, der Sender zwischen 13 und 14.

Besonderheit des Wait- und Final-Bits

Im Wait- und Final-Bit werden keine Informationen codiert, beide haben lediglich eine Synchronisierungsfunktion. Auf den ersten Blick scheint der Zustand der Sendeleitung also irrelevant zu sein, logisch 0 oder logisch 1 wären also denkbar. Tatsächlich ist aufgrund der zeitlichen Verschiebung von Sender und Empfänger (der Empfänger wechselt noch vor dem Sender in den Idle-Zustand) eine Belegung zumindest des Final-Bits mit logisch 0 nötig. Dadurch kann erst zu Beginn des SwapBits eine neue Übertragung durch den ursprünglichen Empfänger erfolgen (grundsätzliche Empfangsbereitschaft).

Der Empfänger muss daher als zusätzliche Sicherheit zum Zeitpunkt 25 und 26 seine Empfangsleitung prüfen. Eine negative Quittierung ist aber nur noch zum Zeitpunkt 25 möglich.

Berechnung der Parität

Berechnet wird die Parität über das Datenbyte (8 Bit). Das Startbit wird also nicht mit einbezogen (ist auch nicht sinnvoll, da dieses immer logisch 0 ist $\rightarrow y \text{ xor } 0 = y$). Verwendet wird das Verfahren der geraden Parität, bei einer ungeraden Anzahl an Einsen wird also das Paritätsbit gesetzt. Die Berechnung wird also folgendermaßen durchgeführt:

$$p = \overline{D_0 \leftrightarrow D_1 \leftrightarrow D_2 \leftrightarrow D_3 \leftrightarrow D_4 \leftrightarrow D_5 \leftrightarrow D_6 \leftrightarrow D_7}$$

Wird die Paritätsberechnung direkt über alle acht Datenbits durchgeführt, so kann beispielsweise die Anzahl der Eins-Bits gezählt werden. In unserem Fall wird dies mit Hilfe des nachfolgenden Algorithmus gemacht. Die Besonderheit liegt darin, dass die Anzahl der Schleifendurchläufe von der Zahl der gesetzter Einsen in dem Datenbyte abhängt. Bei nur zwei gesetzten Bits werden z.B. auch nur zwei Schleifendurchläufe benötigt.

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

C Code:

```
while(x) {           //Anzahl der gesetzten Bits bestimmen
    count++;
    x &= (x - 1);   // x = x & (x - 1)
}
p = count & 1;       //Parität bestimmen
```

Beispiel: Gerade Parität der Dezimalzahl 6

$$110_2 \triangleq 6_{10}$$

Schritt 1:	$110 - 1 = 101$
Schritt 2:	$101 \& 110 = 100$
Schritt 3:	$100 - 1 = 011$
Schritt 4:	$011 \& 100 = 000$ Fertig

Man kann erkennen, dass immer das niederwertigste 1-Bit entfernt wird. Die Schleife wird zwei mal durchlaufen, also sind zwei Eins-Bits in der Zahl enthalten. Damit ist die Belegung des Paritäts-Bits sofort ersichtlich.

Optimierung der Paritätsberechnung: Die Parität wird erst nach vollständiger Übertragung aller Datenbits berechnet. Bei hohen Übertragungsraten (kleinen Periodendauern) hat sich dies als Nachteil herausgestellt, da die Berechnung in diesem Fall zum Flaschenhals des gesamten Übertragungsvorganges wird. Sinnvoller ist es, die Parität direkt nach jedem empfangen Bit anzupassen.

Mögliche Übertragungsfehler und deren Behandlung

Gleich zu Beginn sollte erwähnt werden, dass nicht alle denkbaren Fehler erkannt oder behoben werden können. Teilweise sind Erkennung und Behebung von Fehlern auch geräteabhängig. Beispielsweise können die beim MOSES verwendeten, rein flankengesteuerten, Leitungen Probleme verursachen, die bei der SPS nicht denkbar sind. Auch können Kombinationen von Fehlern möglicherweise dazu führen, dass sie nicht erkannt werden können. Dennoch wird versucht, möglichst viele Fehler zu erkennen und diese ggfs. zu beheben.

Es werden an dieser Stelle nur zufällige Fehler betrachtet. Falsche

Konfigurationseinstellungen des Anwenders und Ähnliches werden weiter unten diskutiert.

Daten-Byte falsch gelesen (Empfangsfehler)

Falls der Empfänger eines der Bits nicht richtig einliest (oder eine ungerade Anzahl) wird bei der Berechnung der Parität ein Fehler auftreten. Mit einer geraden Anzahl von fehlerhaften Bits kann durch die Paritätsprüfung mit einem Paritäts-Bit kein Fehler festgestellt werden.

Lösung / Korrektur:

Siehe Paritätsprüfung

Falsches Paritäts-Bit gelesen (Empfangsfehler)

Falls der Empfänger ein falsches Paritäts-Bit einliest, wird die Paritätsprüfung einen Fehler melden, wenn nicht noch weitere Datenbits betroffen sind.

Lösung / Korrektur:

Siehe Paritätsprüfung

Paritätsprüfung fehlgeschlagen (Empfangsfehler)

Die Paritätsprüfung schlägt immer dann fehl, wenn die berechnete Parität über das Datenbyte nicht mit dem zusätzlichen Paritäts-Bit übereinstimmt. Dies kann verschiedene Gründe haben, die Fehlerquelle kann hier nicht zwangsläufig ausgemacht werden.

Es ist davon auszugehen, dass die Berechnung der Parität auf den jeweiligen Geräten immer korrekt ist.

Lösung / Korrektur:

Der Empfänger erkennt den Paritätsfehler bevor das Acknowledge-Bit auf die Leitung geschrieben wird. Daher kann dem Sender dies auch mitgeteilt werden. Ggf. kann die Übertragung also wiederholt werden.

Start-Bit nicht erkannt (Empfangsfehler)

Die fallende Flanke des Start-Bits wird nicht erkannt, der Empfänger wechselt nicht in den Empfangsmodus. Dessen TxD-Leitung wird also im Zustand HIGH verbleiben.

Lösung / Korrektur:

Da der Sender nach dem Start-Bit prüft, ob der Empfänger auch tatsächlich in den Empfangsmodus gewechselt ist (LOW-Signal auf dessen TxD-Leitung), kann der Fehler frühzeitig erkannt werden. Die Übertragung muss in diesem Fall vom Sender abgebrochen werden und kann nach einer gewissen Wartezeit wiederholt werden.

Falsches Stop-Bit gelesen (Empfangsfehler)

Das Stop-Bit muss immer den Wert Eins besitzen. Aus verschiedenen Gründen kann es passieren, dass der Empfänger den falschen Wert einliest.

Lösung / Korrektur:

Die Prüfung des Stop-Bits ist fester Bestandteil der Empfangsroutine. Der Fehler kann also erkannt und mithilfe des ACK-Bits dem Sender mitgeteilt werden.

Wechsel in den Empfangsmodus wird nicht erkannt (Sendefehler)

Es ist denkbar, dass der Empfänger bei erkanntem StartBit zwar in den Empfangsmodus wechselt, der Sender (MOSES) aber die fallende Flanke des Empfängers nicht registriert.

Demzufolge wird der Empfänger den Übertragungsvorgang zu Ende führen, der Sender bricht die Übertragung aber nach dem Start-Bit ab.

Die Bits, die der Empfänger jetzt einließt sind vom anschließenden Verhalten des Senders abhängig. Wird sofort, oder in kurzen Zeitabständen versucht einen neuen Sendevorgang zu initiieren, kann es durchaus sein, dass die Übertragung aus Empfängersicht erfolgreich abgeschlossen wird (wenn auch unwahrscheinlich). Auch weitere komplexere Fehler wären vorstellbar, die sich auch über mehrere Übertragungen fortziehen könnten.

Lösung / Korrektur:

Um die obigen Probleme zu vermeiden, sollte der Sender nach einer gescheiterten Sendeinitiierung vor einem neuen Sendevorgang die volle Übertragungszeit abwarten. Die Sendeleitung würde also nach dem Start-Bit wieder in den Zustand HIGH wechseln. Dadurch liest der Empfänger für alle Datenbits sowie für das Paritäts-Bit eine 1 ein. In dem Fall wird die Paritätsprüfung den Fehler erkennen und das Datenbyte durch den Empfänger verworfen. Weiterhin wird das Wait-Bit durch den Empfänger geprüft. Dies muss laut Definition logisch 0 sein.

Beim MOSES ist auch eine (optionale) weitere Eingangsleitung vorgesehen, sodass nicht mehr auf Flankenwechsel reagiert werden muss, sondern immer der aktuelle Zustand gelesen werden kann.

Verlassen des Empfangsmodus wird nicht erkannt (Sendefehler)

Wie im oberen Fall, ist es möglich, dass der Sender die letzte steigende Flanke des Empfängers (ACK - Bit bzw. Wechsel nach Idle) verpasst. Zum Einen ist nicht klar, ob die Empfangsquittierung (ACK-Bit) positiv oder negativ war. Zum Anderen kann kein neuer Sendevorgang initiiert werden, da die grundsätzliche Empfangsbereitschaft des Gegenübers scheinbar nicht vorhanden ist. Erst eine fallende und wieder steigende Flanke auf der RxD-Leitung kann wieder den korrekten Zustand herstellen. Das Problem liegt darin, dass der Empfänger diesen Fehler nicht erkennen kann. Ohne weitere Flankenwechsel die eine Blockierung des Senders bedeutet.

Lösung / Korrektur:

Eine wirklich praxistaugliche Lösung gibt es vermutlich nicht. Denkbar wäre aber folgendes:
Sollte der Empfänger über einen längeren Zeitraum keine Sendeanfragen erhalten, könnte er einen kurzen 0-Impuls auf seine TxD-Leitung geben. Der Gegenüber würde durch das vermeintliche Start-Bit in den Empfangsmodus wechseln, das gelesene Datenbit anschließend jedoch verwerfen. So würde aber erreicht, dass wieder der korrekte Zustand der Leitung vorliegt. Diese Lösung ist aber nur möglich, wenn die Rollen von Sender und Empfänger klar zugewiesen sind und sich nicht ändern.

Denn: Sollte der ursprüngliche Empfänger in den Sende-Modus wechseln, kann dies fatale Folgen haben, da ein Flankenwechsel vom ursprünglichen Sender benötigt würde um die Zustände wieder richtig zu beurteilen.

Falls besonderer Wert auf Sicherheit (MOSES) gelegt wird, kann auch zur Vermeidung dieses Fehlers der optionale zusätzliche Eingang benutzt werden.

*Diese beiden Fehler treten (wahrscheinlich) nur auf, wenn der Sender ein MOSES-Gerät ist.
 Dessen Empfangsleitung (CB1) ist rein flankengesteuert, daher ist das Einlesen des aktuellen Wertes im Standard-Betrieb nicht möglich.*

Flanken während der Übertragung werden nicht erkannt (Empfangsfehler)

Dieser Fehler tritt nur beim MOSES auf, da die C1 Leitung flankengesteuert ist.

Dies kann zu Paritätsfehlern führen oder zu Fehlern die nicht erkannt werden können.

Siehe FAQ.

Lösung / Korrektur:

Der Fehler kann umgangen werden indem zusätzlich ein Pin des Datenports verwendet wird.

Empfänger verlässt Empfangsmodus zu früh

Der Empfänger kann den Empfangsmodus aus irgendwelchen Gründen zu früh verlassen.

Beispielsweise könnte durch den Anwender oder das Programm die Empfangsfreigabe mit sofortiger Wirkung (wenn auch nicht empfehlenswert) entzogen werden. In jedem Fall muss von einer Störung ausgegangen werden.

Lösung / Korrektur:

Der Sender prüft zu allen Zeitpunkten seine RxD-Leitung. Sollte diese den falschen Signal-Pegel (HIGH) führen, so muss die Übertragung unverzüglich abgebrochen werden, da der Empfänger wieder im Idle-Zustand sein könnte. Da der Empfänger nicht alle Daten-Bits bzw. Stop-Bits gelesen hat, ist auch davon auszugehen, dass das Datenbyte nicht korrekt ist. Eine Wiederholung der Übertragung durch den Sender ist daher ggfs. nötig.

Keine Korrektur falls:

der Empfangsmodus während des Wait-Bits oder nach dem Wait-Bit verlassen wird, da das Datenbyte bereits quittiert wurde und die Übertragung aus Sendersicht erfolgreich war.

Die serielle Schnittstelle sollte so implementiert werden, dass eine Entziehung der Empfangsfreigabe sich nicht auf eine aktuelle Übertragung auswirkt.

Sender verlässt Sendemodus zu früh

Im Regelfall kann der Sendebetrieb nicht ohne weiteres unterbrochen werden. Geschehen könnte dies durch Ausschalten des Gerätes oder Programmabsturz etc. Der Fehler kann vom Empfänger nur dann mit Sicherheit erkannt werden, wenn die RxD-Leitung logisch 1 ist. In dem Fall würde der Fehler spätestens beim Wait-Bit oder Final-Bit erkannt.

Ist RxD jedoch logisch 0, so kann der Fehler nur erkannt werden, wenn der Sendebetrieb vor dem Stop-Bit beendet wurde, wodurch der Empfänger ein falsches Stop-Bit liest.

Negative Quittierung

Erhält der Sender eine negative Quittierung für ein übertragenes Datenbyte, wird der Empfänger in der Regel einen Fehler im Datenbyte oder Ähnliches erkannt haben.

Lösung / Korrektur:

Da der Empfänger das Datenbyte ohnehin nicht in sein Empfangs-FIFO eintragen wird, kann der Sender die Übertragung wiederholen.

Falsches ACK-Bit gelesen (Sendefehler)

Sollte der Sender einen falschen Wert für das Acknowledge-Bit lesen, ist die Erkennung des Fehlers nahezu ausgeschlossen, da dies für den Empfänger im Prinzip die einzige Möglichkeit darstellt, ein Datenbyte zu quittieren. Eine weitere Fehlererkennung und -korrektur auf höheren Schichten könnte eine Option sein.

Empfänger wechselt nicht in Empfangsmodus

Wenn der Empfänger nicht in den Empfangsmodus wechselt, dann muss dies nicht zwangsläufig auf einen Fehler hindeuten (Empfang kurzzeitig durch Programm blockiert o.Ä.).

Lösung / Korrektur:

Wie Abschnitt Start-Bit nicht erkannt.

Leitung TxD des Senders führt falschen Spannungspiegel während Wait- und Final-Bit

Während Wait- und Final-Bit muss an der TxD-Leitung des Senders ein LOW-Signal anliegen. Sollte dies nicht der Fall sein (einige Gründe wurden bereits angeführt), ist davon auszugehen, dass auch das empfangene Datenbyte nicht korrekt ist.

Lösung / Korrektur:

Sollte der Fehler bereits beim Wait-Bit auftreten, kann das empfangene Datenbyte ohne Probleme verworfen werden, da über das ACK-Bit dem Sender der Fehler mitgeteilt werden kann. Weitere Überlegungen hierzu sind also nicht nötig.
 Sollte aber erst beim Final-Bit die TxD-Leitung des Senders im Zustand logisch 1 sein, kann dies dem Sender nicht gemeldet werden, da das ACK-Bit bereits gesendet wurde. Eine Fehlerbehandlung wird also schwierig. Es muss also offen gelassen werden, ob das empfangene Datenbyte akzeptiert oder verworfen wird.

Vollständig belegtes Empfangs-FIFO

Unter Umständen kann es passieren, dass das Empfangs-FIFO sehr schnell gefüllt wird. Ist das letzte Byte belegt, so können keine weiteren Werte aufgenommen werden.

Lösung / Korrektur:

1. Solange das Empfangs-FIFO voll ist wird nach jedem empfangen Datenbyte eine negative Quittierung herausgegeben (NACK)
2. Es wird beim Sendeversuch nicht in den Empfangsmodus gewechselt

Mögliche Optimierung der Schnittstellendefinition

Nach Implementierung der Schnittstellen auf MOSES und SPS und anschließenden Tests haben sich einige Punkte herauskristallisiert, die optimiert werden können. Dafür müssen jedoch Teile der Schnittstellendefinition angepasst werden.

Wie sich herausgestellt hat, ist die Paritätsberechnung beim Empfänger bei hohen Übertragungsraten zu langsam. Trotz WaitBits (das Zeit für Berechnungen geben sollte) ist dieses Programmstück der Flaschenhals im MOSES-Programm. Außerdem werden Start- und StopBit erst zum Ende des Empfangsvorganges geprüft. Sinnvoller ist es alle Bits direkt beim Einlesen zu prüfen. Die Parität würde also bei jedem Abtastvorgang sukzessive berechnet. Dadurch würde das WaitBit überflüssig und das Acknowledge-Bit könnte sofort nach Einlesen des StopBits auf die Leitung geschrieben werden.

Wird für das SwapBit eine volle Periode festgelegt, wird das FinalBit auch nicht mehr nötig sein, da gewährleistet ist, dass der Sender nicht zu früh einen neuen Sendeversuch startet.

Durch diese beiden Änderungen wird zum Einen der komplette Übertragungsvorgang um 1/7 verkürzt. Zum Anderen werden zwischen zwei MOSES-Geräten vermutlich noch höhere Übertragungsraten (kleinere Periodendauern) möglich sein, da der Flaschenhals der Paritätsberechnung etc. auf die gesamte Übertragung aufgeteilt wird.

6. Funktionsweise der FIFO

Allgemein

Bei einer FIFO-Datenstruktur (**First In First Out**) handelt es sich um eine klassische Warteschlange. Werte die als erstes in diese Datenstruktur geschrieben werden, müssen auch wieder als erstes ausgelesen werden. Theoretisch kann solch eine Warteschlange unendlich viele Werte aufnehmen. In der Realität ist die Größe der FIFO beschränkt. Eine mögliche Implementierung, einer in der Größe beschränkten FIFO, stellt der sogenannte Ringpuffer dar. Hierbei handelt es sich um eine Datenstruktur mit zwei umlaufenden Zeigern. Ein Zeiger (IN-Pointer) zeigt auf das erste freie Datenelement, der zweite Zeiger (OUT-Pointer) zeigt auf das erste belegte Element im Puffer.

Minimaler Funktionsumfang der FIFO:

1. Schreiben von Bytes in die FIFO
2. Lesen von Bytes aus der FIFO
3. Berechnung der Anzahl an belegten Speicherstellen in der FIFO
4. Berechnung der Anzahl an freien Speicherstellen in der FIFO

Vergleich zwischen SPS und MOSES

MOSES

Für den MOSES sind zwei Versionen einer FIFO vorhanden. Die erste Version erfüllt nur den Minimalumfang, ist aber deutlich schneller als die zweite. Daher sollte diese auch im Regelfall für die Kommunikation verwendet werden.

Mit der zweiten Version kann eine Liste von FIFOs von beliebiger Größe erzeugt werden. So kann man z.B. eine Sende- und eine Empfangs-FIFO (wenn dies sinnvoll erscheint) oder mehrere Sende-FIFOs unterschiedlicher Priorität verwenden.

Generell wird für den MOSES jedoch nur ein Empfangs-FIFO benötigt, da ein Sendevorgang ohnehin das laufende Programm blockiert und erst wieder zurückkehrt, wenn die Übertragung abgeschlossen ist (keine Parallelität möglich).

SPS

Die FIFO der SPS verfügt über ein paar zusätzlicher Informationen wie z.B. Datum und Zeit der Übertragung.

Die Größe der FIFO wird vom Anwender vorher festgelegt.

Bei einer vollständigen Füllung der FIFO werden die ersten eingetragenen Bytes wieder überschrieben.

Die SPS besitzt 2 FIFO's, ein Sende- und ein Empfangs-FIFO.

Die weiteren Funktionsweisen der FIFO von SPS und MOSES sind weitestgehend übereinstimmend. Für einen detaillierten Einblick sollte der Quellcode zu Rate gezogen werden.

7. MOSES-Programm

Verwendung der Bibliothek

Alle Funktionalitäten, die zur erfolgreichen Nutzung der seriellen Schnittstelle nötig sind, wurden in einer Library hinterlegt, die leicht in ein bestehendes Assembler-Programm eingebunden werden kann. Im Gegensatz zu Bibliotheken wie die KONST.LIB oder PIAT.LIB enthält die SPSMOS.LIB nicht ausschließlich Konstanten, sondern auch den kompletten Assembler-Code. Bei der Verwendung in einem eigenen Anwenderprogramm muss also darauf geachtet werden, dass keine Überschneidung von Adressbereichen stattfindet. Notfalls können die Adressbereiche in der Bibliothek jedoch relativ einfach angepasst werden. Im Regelfall reicht es, wenn die erste Zeile (Konstante progAddr) angepasst wird. Standardmäßig wird die Library an der Speicheradresse \$3900 im MOSES abgelegt. Die Adresse des Empfangs-FIFOs muss auch nicht vom Anwender festgelegt werden. Standardmäßig folgt das Datenfeld direkt auf die Programmanweisungen der seriellen Schnittstelle. Die genaue Adresse der empfangenen Daten ist nicht zwangsläufig nötig. Da alle nötigen FIFO-Operationen bereitgestellt werden, kann bei Bedarf nachgeschlagen werden (Konstante _fifoAddr). Sollte die FIFO-Größe einmal angepasst werden müssen (max. 255 Elemente), reicht es wenn die entsprechende Konstante (_fifoSize) geändert wird.

Ansonsten kann, wenn alle Standardparameter beibehalten werden, das Anwenderprogramm ohne Probleme an der üblichen Adresse \$4000 beginnen.

Zu Beginn der Datei werden alle nötigen Konstanten und Adressen definiert. Von einer Änderung der Konstanten (außer den weiter oben genannten) sollte ohne genaue Kenntnis des Programmes abgesehen werden, da dies unter Umständen fatale Folgen nach sich ziehen würde.

Grundsätzliches

Subroutinen

Generell sind alle Subroutinen so geschrieben, dass keine Speicherbereiche verändert werden (für temporäre Variablen wird der Stack verwendet). Lediglich zwei Byte werden für Verwaltungsinformationen benötigt. Auch der Akkumulator und das X- und Y-Register behalten beim Verlassen der Routine (sofern nicht als Rückgabewert verwendet) ihre ursprünglichen Werte. So werden Missverständnisse und unerklärliche Fehler vermeiden.

Benötigte PIAT-Ressourcen

Für die serielle Schnittstelle werden die Leitungen CB1 (RxD) und CB2 (TxD) benötigt. Ansonsten kann der Port B durch den Anwender genutzt werden. Jedoch sollten die acht Daten-Pins allesamt als Ausgänge verwendet werden, da durch Lesevorgänge auf das Datenregister auch der IRQ-B quittiert wird. Daher ist bei der Verwendung von Pins als Eingänge äußerste Vorsicht geboten. Sollten Eingänge nötig sein, kann der Port A ohne Einschränkung genutzt werden. Unter Umständen muss auf den Pin 0 des Ports B verzichtet werden (siehe SPSIni).

Für die serielle Schnittstelle werden außerdem die Timer-Funktionen des PIAT-Bausteins verwendet. Daher sollten Schreibvorgänge auf folgende Register bei der seriellen Schnittstelle vermieden werden: Upper Latch (UL, ULEC), Lower Latch (LL), Counter Mode Control Register (CMCR), Control Register B (CRB).

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

Weiterhin kann das Einlesen des Data Registers B (LDA DRB) unter Umständen zu Fehlern führen (besonders kritisch ist die Verwendung in kurzen Schleifen).

Weitere Ressourcen

Neben den bereits oben benötigten Speicherbereichen werden unter Umständen auch vier Bytes der Zero-Page belegt. Für den Vektor IRQAVek ist die Adresse \$30 und für IRQTVek die Adresse \$32 festgelegt worden (können ggfs. im Quelltext schnell angepasst werden).

Für temporäre Variablen werden außerdem von einigen Routinen kurzzeitig wenige Bytes auf dem Stack benötigt.

Die Anwenderanzeige und die Eingabeschalter am MOSES werden im Programm nicht verwendet.

Verwendung von Interrupts im eigenen Anwenderprogramm

Grundsätzlich können auch Interrupts zusammen mit dieser Bibliothek benutzt werden. Sollen diese ohne Einschränkung nutzbar sein, so muss die Subroutine SPSIni mit dem Parameter zur Löschung aller Einstellungen aufgerufen werden. Anschließend können alle möglichen IRQs verwendet werden. Selbstverständlich sind dann keine Kommunikationsvorgänge über die serielle Schnittstelle möglich. Außerdem müssen die benötigten PIAT-Register vom Anwender neu initialisiert werden, selbst wenn dies an vorherigen Stellen im Programm bereits vorgenommen wurde.

Während der Verwendung der seriellen Schnittstelle kann lediglich der IRQ-A des PIAT in vollem Umfang benutzt werden. Die Benutzung des Timer-Interrupts ist nur sehr eingeschränkt möglich und sollte daher nicht gleichzeitig mit der seriellen Schnittstelle verwendet werden (obwohl dies auch im Quelltext vorgesehen ist). Der IRQ-B wird von der seriellen Schnittstelle benötigt und kann überhaupt nicht für andere Zwecke verwendet werden.

Verwendung des IRQ-A

Soll der IRQA des PIAT-Bausteins verwendet werden, so darf der Vektor UIRQVek NICHT vom Anwender umgebogen werden. Stattdessen wird der Vektor UIRQAVek bereitgestellt. So können die Funktionalitäten des Ports A im vollen Umfang genutzt werden. Anweisungen, die sich auf den Port B beziehen, sollten jedoch in der Behandlungsroutine vermieden werden (z.B. LDA DRB) wenn deren Wirkung nicht zu 100% klar ist. Am Ende der Behandlungsroutine darf kein RTS - Befehl stehen. Als letzte Anweisung sollte JMP IRQAEnde verwendet werden. Der Befehl JMP IRQEnde wird aber auch funktionieren.

Verwendung des IRQ-T

Für die Verwendung des IRQ-T sollte sichergestellt werden, dass während der Benutzung durch den Anwender keine Empfangsvorgänge stattfinden. Dies kann z.B. durch den Aufruf von SPSIni mit dem entsprechenden Parameter geschehen. Weiterhin sollten keine Daten versendet werden (kann vom Anwender kontrolliert werden), da sonst die entsprechenden Timer-Register umgeschrieben werden. Werden diese beiden Punkte beachtet, kann der Timer beliebig genutzt werden ohne das später eine Neu-Initialisierung mit SPSIni erfolgen muss um die Funktionen der seriellen Schnittstelle zu nutzen. Die Timer-Register können also beliebig beschrieben und gelesen werden. Es muss lediglich darauf geachtet werden, dass wie beim IRQ-A nicht der Vektor UIRQVek umgebogen wird, sondern in diesem Fall UIRQTVek. Auch hier darf am Ende der Behandlungsroutine kein RTS - Befehl stehen, als letzte Anweisung sollten JMP IRQTEnde oder JMP IRQEnde benutzt werden.

Zusätzliche Informationen

Parallele Verwendung der Zusatzleitung am MOSES?

Da der Eingang CB1 rein flankengesteuert ist, oft aber der aktuelle Zustand der Leitung benötigt wird, müssen die einzelnen Flankenwechsel registriert und aufgezeichnet werden. Unter Umständen können Flanken verpasst werden, sodass falsche Zustände der Leitung zwischengespeichert sind. Aus diesem Grund ist es möglich, einen weiteren Pin des Port B zu verwenden, um dieses Manko zu umgehen.

Vom Anwender sind dazu keine besonderen Änderungen im Quelltext nötig. Lediglich der Parameter der Subroutine SPSIni muss entsprechend angepasst werden.

Blockierung von Interrupts

Es gibt einige kritische Programmberiche, die nicht unterbrochen werden dürfen, wie zum Beispiel die FIFO-Operationen Lesen und Schreiben. Sollte z.B. beim Auslesen eines Wertes die Operation unterbrochen und ein Schreibvorgang ausgeführt werden, können die Zeiger der FIFO falsche Werte annehmen.

Generell ist es erforderlich IRQ's an Stellen zu blockieren, an denen gemeinsam genutzte Betriebsmittel (Variablen) gelesen / verändert werden. Dadurch ergeben sich atomare Operationen, die aus mehreren Assembler-Anweisungen bestehen.

Keine Subroutine für den Empfangsmodus

Der Grund dafür, dass dem Anwender keine Routinen (außer receiveCheck) für den Empfang zur Verfügung stehen, ist ganz einfach: Ein Kommunikationsvorgang wird immer vom Sender initiiert. Der Empfänger kann lediglich auf eine Anfrage reagieren. Der Zeitpunkt einer Übertragung wird also immer vom Sender festgelegt. Der Empfangsvorgang läuft daher komplett im Hintergrund über IRQs ab und das Anwendungsprogramm muss von Zeit zur Zeit das Empfangs-FIFO prüfen.

Kommunikation zwischen Subroutinen / IRQ-Behandlungsroutine

Für die Kommunikation von Routinen untereinander werden Variablen benötigt. Die Kommunikationsroutinen verwenden im wesentlichen ein Flag-Byte (_serFlags) um Informationen für andere Routinen bereitzustellen. Das Byte wird für den Empfang sowie für das Senden benötigt. Bei der Fehlersuche ist die Untersuchung dieses Bytes nötig.

Auch die einfache FIFO benötigt ein Flag-Byte, sodass Informationen zwischen den einzelnen Routinen ausgetauscht werden können.

Realisierung der Swap-Time

Während des Swap-Bits wird dem Empfänger ein gewisser Zeitraum eingeräumt, um selbst eine neue Übertragung starten zu können. Dazu muss beim (ursprünglichen) Sender ein IRQ durch eine Flanke ausgelöst werden. Um dies zu erreichen wird beim Sender das Programm durch eine Zeitschleife am Ende von SPSAus blockiert. Die Verwendung vom PIAT-Timer ist also nicht möglich.

Die Wartezeit ergibt sich aus der Zahl der Schleifendurchläufe, der Taktfrequenz des MOSES (4MHz) und die Anzahl an nötigen Takten pro Schleifendurchlauf. Zur (eher geringen) Erhöhung der Übertragungsrate kann daher die Zahl der Schleifendurchläufe reduziert werden.

8. SPS Programm

Programmstruktur

OB1

Symbol	Adresse	Kommentar
fbSimulation	FB 1	Für Testzwecke und Simulation
fcComMoses	FC 100	Kommunikation mit MOSES µP-Steuerung
fcComWinCC	FC 2	Kommunikation mit WinCC
fcSendReceiveHost	FC 70	Kommunikation mit HOST SPS
fcFaultRecognition	FC 20	Fehlerbehandlung

Organisationsbaustein 1

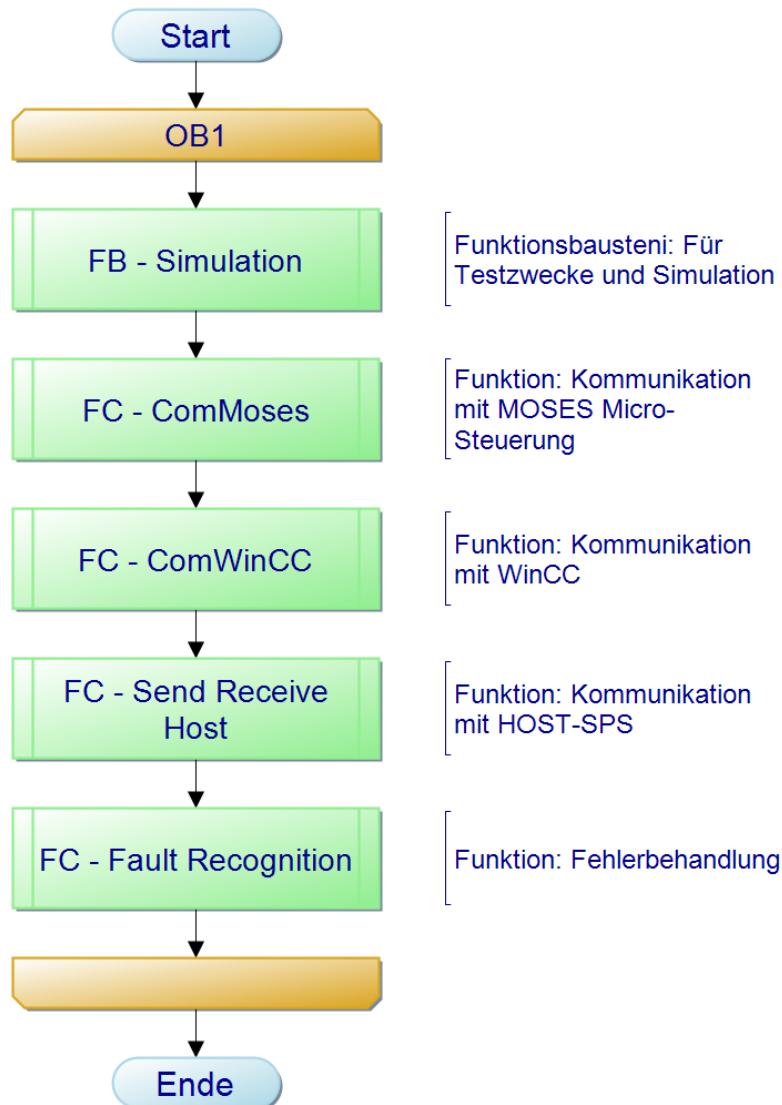


Abbildung 5 Ablauf des Organisations-Bausteins OB1

ComMOSES

Symbol	Adresse	Kommentar
fcRcvMoses	FC 110	Datenempfang von MOSES µP-Steuerung bearbeiten
fcSndMoses	FC 115	Daten Senden zur MOSES µP-Steuerung bearbeiten
fcOrgTxD_Line	FC 35	Organisation zur Ansteuerung der TxD Leitung

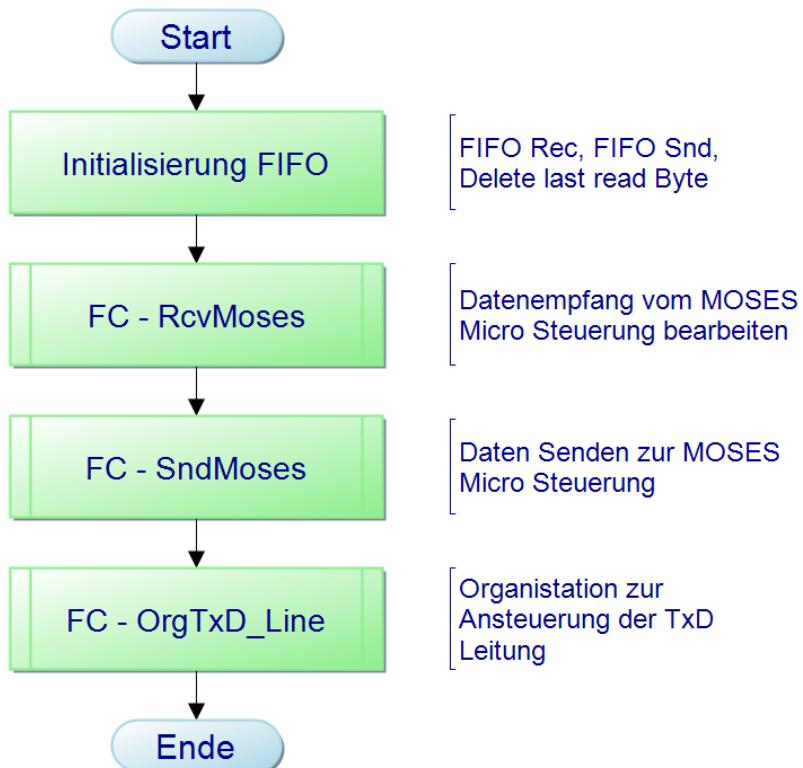
ComMoses

Abbildung 6 Ablauf des ComMoses FB's

RcvMOSES

Symbole	Adresse	Kommentar
fbReceiveMOS	FB 20	Datenempfang vom MOSES bearbeiten
fcOrgFIFO_Rec	FC 50	Organisation mit FIFO für Datenempfang

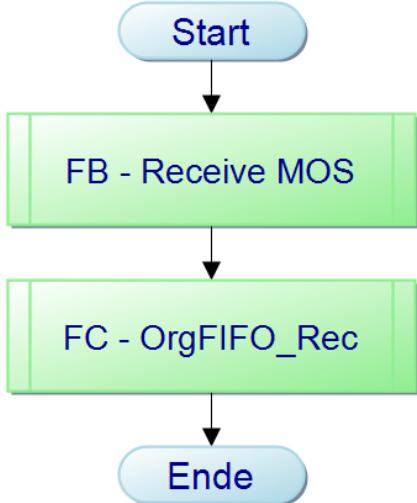
FC - RcvMoses

Abbildung 7 Ablauf der Funktion RcvMoses

SndMOSES

Symbol	Adresse	Kommentar
fcGetSendData	FC 40	Zu sendendes Byte in Sende-FIFO eintragen
fcSendData	FC 41	Senden an MOSES µP-Steuerung steuern
fbSendMOS	FB 21	Senden eines Bytes an die MOSES µP-Steuerung
fcOrgFIFO_Snd	FC 51	Organisation mit FIFO für sendedaten an MOSES µP-Steuerung

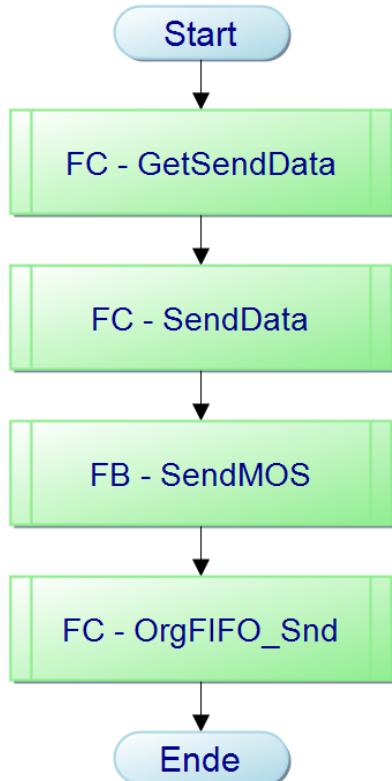
FC - SndMoses

Abbildung 8 Ablauf der Funktion SndMoses

Beschreibung der wichtigsten Funktionen, Funktions- und Organisationsbausteine

Detaillierte Informationen sollten dem SPS Quellcode entnommen werden. Die Beschreibung dient lediglich der Übersicht.

Funktion einer Schrittkette

Eine Schrittkette besteht aus mehreren Schritten die jeweils eine Bedingung erfüllen müssen.

Ist die Bedingung erfüllt, wird in den nächsten Schritt der Schrittkette gewechselt.

So wird erreicht, dass die einzelnen Schritte nacheinander abgearbeitet werden, also eine Sequenz von Schritten durchlaufen wird. Das Überspringen eines Schrittes/ Zustandes ist im Regelfall nicht möglich (nötig). Außerdem bedingt dieser Aufbau das mit dem ersten Schritt gestartet werden muss.

FB – ReceiveMOS

Der Funktionsbaustein ist nach dem Prinzip einer Schritt-kette aufgebaut. Der Funktionsbaustein ist für das Einlesen der Leitung, des Einganges 12.0 zuständig.

Das Einlesen geschieht nach einer Schritt-kette. Die Schritt-kette geht in den nächsten Zustand über, wenn ein Timer abgelaufen ist. So kann die Synchronität zwischen Sender und Empfänger realisiert werden.

Außerdem wird das Start und StopBit kontrolliert. Anschließend wird noch für das Übertragene Byte die Parität berechnet.

Abschließend wird kontrolliert ob alle Bedingungen für eine Korrekte Übertragung erfüllt worden sind. Hier wird dann das Ergebnis der Kontrolle an den Baustein FC-OrgTxLine übergeben.

FB – SendMOS

Nach dem Prinzip einer Schritt-kette aufgebaut, siehe oben(ReceiveMOS).

Der Funktionsbaustein ist für die Übertragung von der SPS zum MOSES zuständig.

Der Baustein bekommt von der FIFO ein Byte übertragen welches er Senden soll.

Als erstes wird ein Startbit über den Ausgang 20.0 gesendet. Antwortet der MOSES darauf wird das Byte übertragen. Während der Übertragung wird kontrolliert ob der Empfänger 0 Singal auf Leitung Eingang 12.0 legt. Sollte das Signal auf 1 wechseln versucht der Sender das Byte erneut zu übertragen.

FC – ReadFrameBit

Der Baustein wurde in SCL geschrieben, eine Programmiersprache speziell für die SPS entwickelt.

Er trägt den Status aller empfangenen Bits in das Zeichenrahmen-Empfangs-Array ein. Sind alle 11 Bits ins ARRAY eingetragen worden, quittiert der Baustein den Empfang und extrahiert das Nutzdatenbyte aus dem Zeichenrahmen. Danach trägt er dieses im Empfangs-DB ein.

FC-RecvMos

Der Baustein wird aufgerufen falls eine fallende Flanke (StartBit) vom OB40 erkannt wird.

Während des Empfangs wird die Funktion ReadFrameBit aufgerufen.

Vor jedem Aufruf von FC-ReadFrameBit wird der Timer neu gestartet(20ms).

Während des WaitBits werden die Kontrollberechnungen durchgeführt.

Anschließend wird das ACK Bit auf die Leitung gegeben.

Damit ist die Übertragung abgeschlossen.

FC – SendData

Der Baustein ist als Schritt-kette aufgebaut.

Im ersten Schritt prüft der Baustein ob ein Datenbyte im Sende-FIFO vorhanden ist.

Wenn ein Datenbyte vorhanden ist dann wird die Sendefreigabe gegeben, Bit gesetzt für FB-SendMos.

Jetzt wird gewartet bis die Übertragung beendet ist. Falls ein Fehler bei der Übertragung gemeldet wurde, wird die Übertragung wiederholt. Bei korrekter Übertragung wird das Datenbyte aus der Sende-FIFO entfernt.

OB-40 Prozessalarm

Is für das Detektieren des Startbits zuständig. Reagiert auf eine Fallendeflanke an Eingang E12.0.

Bei einer fallenden Flanke des Startbits wird in den Empfangsmodus gewechselt, wenn eine Empfangsfreigabe besteht.

Fallende Flanken während einer Übertragung werden ignoriert.

9. Oberfläche WinCC

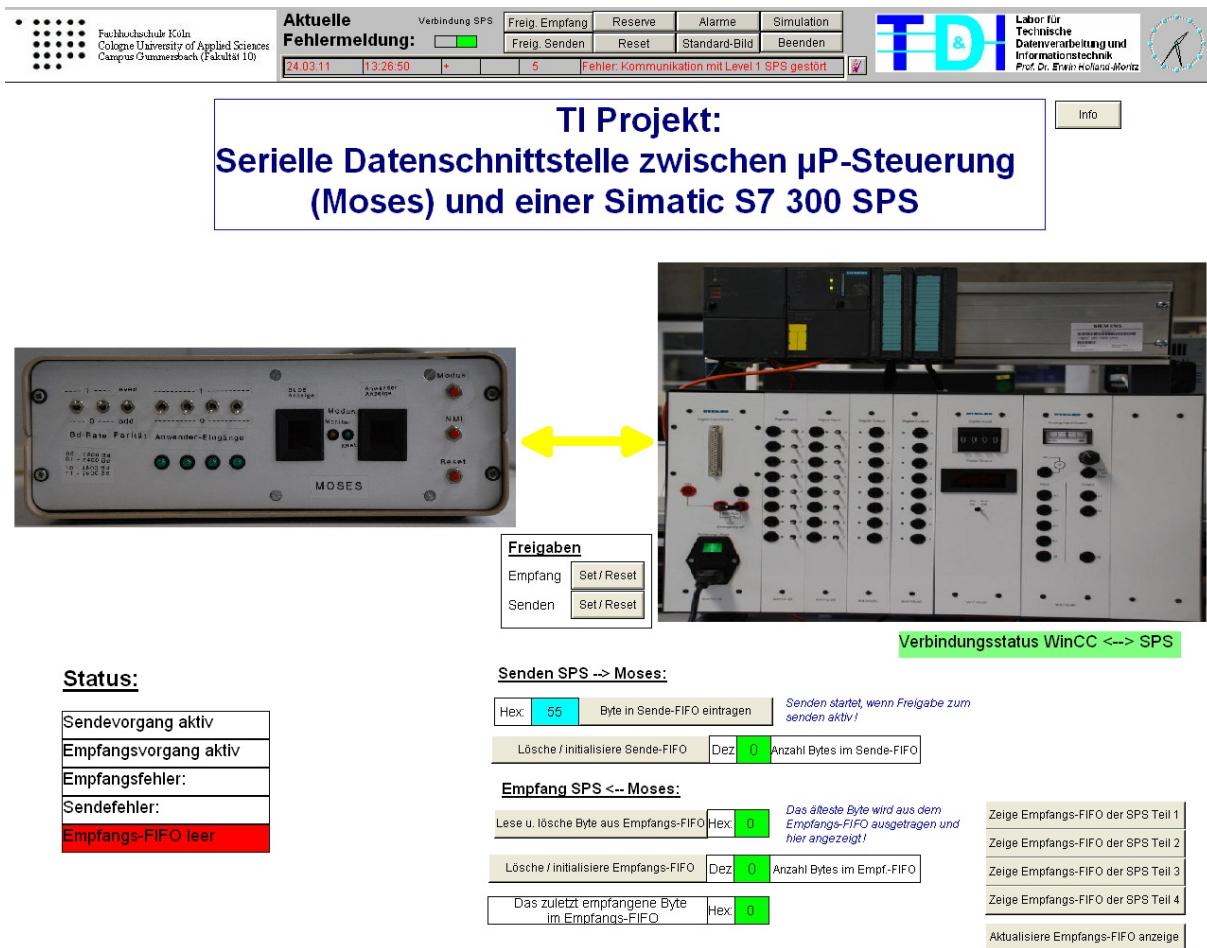


Abbildung 9 Startbildschirm WinCC

WinCC bietet eine grafische Oberfläche mit der Informationen der SPS visualisiert werden können. In der von uns erstellten Oberfläche werden verschiedenste Funktionen bereitgestellt, die eine Übertragung visualisieren. Folgende Funktionen sind vorhanden:

Freigaben und Statusmeldungen

Status:



Die Freigaben für Empfang bzw. Senden kann über zwei Buttons eingestellt werden. Ist die entsprechende Freigabe gesetzt, leuchtet der jeweilige Button grün auf. Die Empfangs- und Sendefreigabe können auch direkt hardwaremäßig an der SPS eingestellt werden und haben Vorrang gegenüber den WinCC-Einstellungen.

In der Statusanzeige wird der aktuelle Zustand grün hinterlegt. Außerdem werden Empfangs- und Sendefehler gezählt und ausgegeben. Ist die Empfangs-FIFO leer, so wird das entsprechende Feld rot hinterlegt.

Ist eine Verbindung zwischen SPS und WinCC aufgebaut, wird der Verbindungsstatus grün hinterlegt.

Empfangs- und Sende-FIFO der SPS

Senden SPS --> Moses:

Hex:	<input type="text" value="55"/>	Byte in Sende-FIFO eintragen	Senden startet, wenn Freigabe zum senden aktiv!	
<input type="button" value="Lösche / initialisiere Sende-FIFO"/>		<input type="text" value="0"/> Dez	Anzahl Bytes im Sende-FIFO	

Empfang SPS <-- Moses:

Lese u. lösche Byte aus Empfangs-FIFO	<input type="text" value="0"/> Hex	Das älteste Byte wird aus dem Empfangs-FIFO ausgetragen und hier angezeigt!		<input type="button" value="Zeige Empfangs-FIFO der SPS Teil 1"/>
<input type="button" value="Lösche / initialisiere Empfangs-FIFO"/>		<input type="text" value="0"/> Dez	Anzahl Bytes im Empf.-FIFO	<input type="button" value="Zeige Empfangs-FIFO der SPS Teil 2"/>
Das zuletzt empfangene Byte im Empfangs-FIFO	<input type="text" value="0"/> Hex			<input type="button" value="Zeige Empfangs-FIFO der SPS Teil 3"/>
				<input type="button" value="Zeige Empfangs-FIFO der SPS Teil 4"/>
				<input type="button" value="Aktualisiere Empfangs-FIFO anzeigen"/>

Im Bereich „Senden SPS -> MOSES“ können HEX-Werte im Byte-Format in die Sende-FIFO eingetragen werden. Weiterhin ist ein Löschen und Initialisieren der FIFO möglich, hier werden außerdem die in der FIFO noch vorhandenen Bytes angezeigt.

Im Bereich „Empfang SPS <- Moses“ können HEX-Werte aus der FIFO ausgelesen und Gelöscht werden. Weitere Funktionen sind das Löschen und Initialisieren der FIFO und die Anzeige des zuletzt empfangenen Bytes welches in die FIFO eingetragen wurde.

Es stehen 4 Buttons zu Verfügung die, die gesamte FIFO anzeigen können und ein Button, der die Werte aktualisiert.

Fehlermeldungen und Resetfunktion

Im oberen Teil des Fensters finden sich einige Buttons, mit denen die aktuelle Ansicht geändert oder andere Funktionen durchgeführt werden können.

10. C++-Programm

Grundsätzliches

Um die Sende- bzw. Empfangsfunktionalitäten von MOSES und SPS zu testen, hat es sich als hilfreich erwiesen, ein Programm mit einer grafischen Oberfläche auf dem PC zu verwenden. Hiermit kann das grundsätzliche Verhalten der Kommunikationspartner überprüft und mögliche Fehler in der Implementierung erkannt werden. Ein großer Vorteil des Programmes liegt darin, dass für viele Tests kein Oszilloskop benötigt wird, da das Programm Aufzeichnungen über die Zustände der Leitungen vornehmen kann.

Weiterhin liegt der Aufwand zur Realisierung eines C++-Programmes, das die serielle Schnittstelle (RS232) des PCs verwendet, deutlich unter dem zur Realisierung eines Assembler-Programmes für den MOSES oder eines Programmes für die SPS S7-300. Das liegt daran, dass es sich bei C++ um eine Hochsprache handelt, weiterhin können viele Komponenten aus bereits bestehenden Bibliotheken entnommen werden (FIFO, RS232, GUI).

Die einzelnen Klassen sind so konzipiert, dass sie einfach in neuen Projekten verwendet werden können, sollte einmal der PC als Kommunikationspartner dienen.

Da die serielle Schnittstelle bipolare Signalpegel verwendet (positive wie negative Spannungen), muss eine Pegelanpassung erfolgen. In diesem Fall wird ein einfacher RS232-TTL-Wandlerbaustein der Firma Pollin verwendet, sodass mit einfachen TTL-Pegeln (0V, 5V) gearbeitet werden kann.

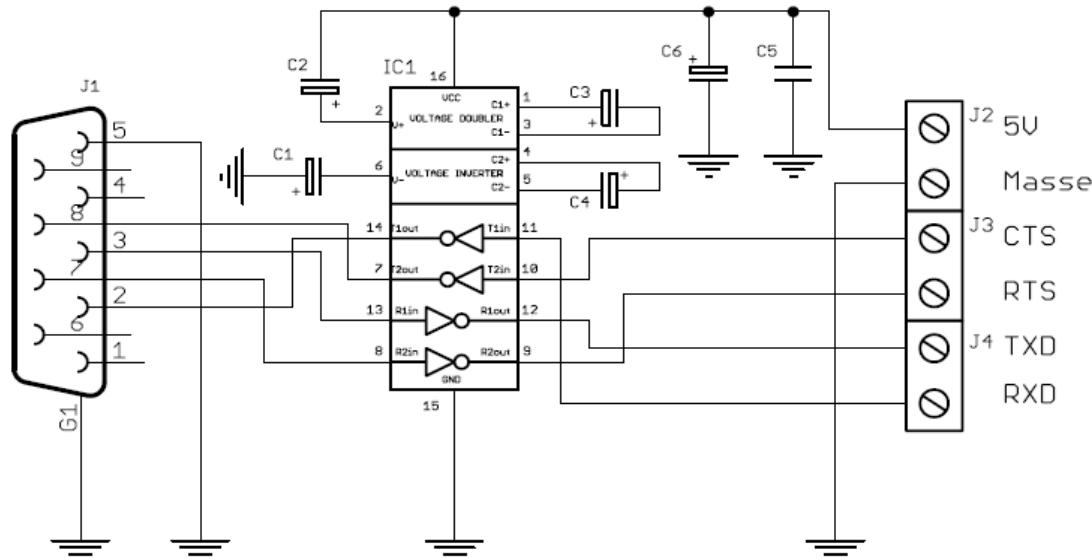


Abbildung 10 Verdrahtungsplan für das Interface PC-SPS/Moses

Es gibt jedoch einige Besonderheiten, die beachtet werden sollten. Da eine echte Flankensteuerung nicht ohne weiteres realisiert werden konnte (eigene Treiber müssten geschrieben werden), musste eine etwas andere Methode verwendet werden. Hierzu wird die Empfangsleitung in geeigneten Zeitintervallen (ca. 1 - 2ms) auf Signalwechsel überprüft und ggfs. ein Ereignis ausgelöst. Da es sich bei Betriebssystemen wie Windows um keine Echtzeitbetriebssysteme handelt, sollte vor der Verwendung des Programmes die Timer-Funktionen auf Ihre Genauigkeit getestet werden, da dies unter Umständen zu Problemen führen könnte. Im Regelfall ist jedoch eine hohe Genauigkeit

gegeben. Ansonsten kann eine Verbesserung durch die Erhöhung der Prozess-Priorität erreicht werden.

Funktionen des Testprogramms

- Auswahl des COM-Ports (falls mehrere vorhanden)
- Empfangsfreigabe: Falls aktiviert, werden Daten des Kommunikationspartners akzeptiert
- Sendefreigabe: Falls aktiviert, werden Daten sequentiell dem Sende-FIFO entnommen und übertragen
- FIFO-Operationen Empfangs-FIFO: Auslesen eines Bytes, Auslesen der kompletten FIFO, Löschen aller Elemente, Anzeige der FIFO-Größe
- FIFO-Operationen Sende-FIFO: Eintragen von Testbyte (0x55), Testfolge (Zahlen 0x0 - 0x63) oder eines beliebigen Bytes, Löschen aller Elemente, Anzeige der FIFO-Größe
- Genauigkeits-Tests der Timer
- Übertragung von Test-Frames: Jedes einzelne Bit des Test-Frames kann vom Anwender festgelegt werden. So können z.B. Fehler bei der Paritätsberechnung des Gegenübers festgestellt werden. Außerdem wird die Reaktion des Kommunikationspartners aufgezeichnet.
- Statistiken: Hier kann man Informationen zu gesendeten / empfangen Bytes, Anzahl und Art der Fehler während der Übertragung und sonstige Informationen entnehmen.

Weitere Informationen können dem ausführlich kommentierten Quelltext im Anhang entnommen werden. Es werden jedoch nur die selbst geschriebenen Klassen aufgeführt.

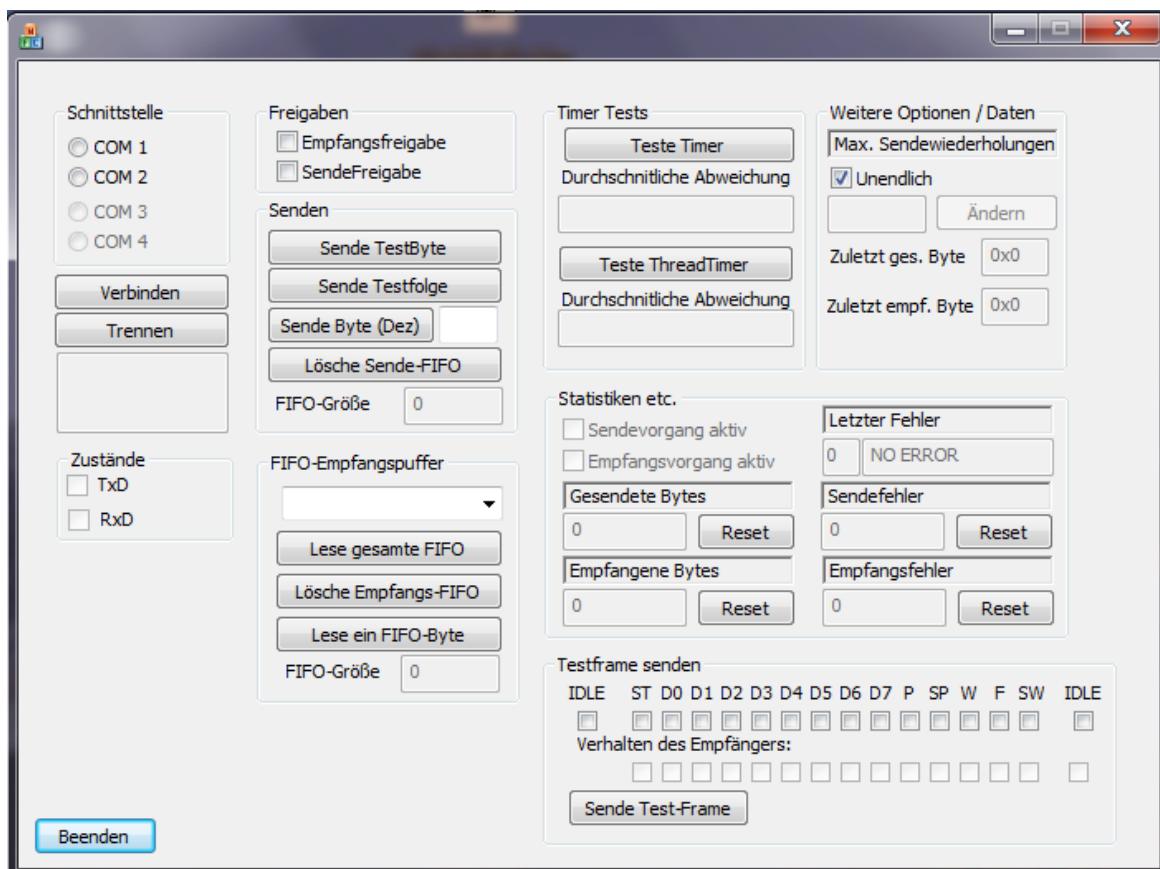


Abbildung 11 Bildschirm C++-Programm

11. Tests

Übertragungsgeschwindigkeiten

MOSES zu MOSES

Um die maximale Übertragungsrate zwischen zweier MOSES-Geräte zu ermitteln, wurden insgesamt 256 (Bytes 0-255) Pakete übertragen. Die minimale Übertragungsdauer beträgt für diesen Fall ziemlich genau eine Sekunde(1.004s). Daraus ergibt sich eine Baudrate von knapp 3600 Baud (in unserem Fall 3600 Bit/s).

Bei einer höheren Übertragungsrate kann der Empfänger die Paritätsberechnung nicht schnell genug durchführen, sodass das ACK-Bit nicht rechtzeitig auf die Leitung geschrieben wird. Der Sender würde dies daher als negative Bestätigung auffassen, obwohl der Empfänger das Byte richtig empfangen und in sein Empfangs-FIFO eingetragen hat. Dies könnte bei einer Sender-seitigen Fehlerbehandlung zu Mehrfachübertragungen führen.

Die maximale Übertragungsrate zwischen zwei MOSES wird im Wesentlichen durch die Dauer der Paritätsberechnung begrenzt.

Die Dauer der Paritätsberechnung kann die Periodendauer überschreiten. Dies führt zu einer Sender-seitigen Fehlinterpretation, da der Empfänger das ACK-Bit zu spät setzt.

SPS zu SPS

Die Übertragungsgeschwindigkeit für eine Kommunikation zwischen zwei SPS-Stationen hängt im Wesentlichen von den Zykluszeiten der beiden Geräte ab. Da diese unter Umständen relativ groß werden können (teilweise >5ms), sollten die Periodendauern 20ms nicht unterschreiten.

Überprüfung der Schnittstellenfunktionalitäten

Nach der Implementierung der Sende- und Empfangsfunktionalitäten wurden umfangreiche Tests durchgeführt. Diese sollen an dieser Stelle beispielhaft am MOSES-Programm dargestellt werden. Für die SPS wurden die Tests in gleicher Weise durchgeführt.

Die meisten Tests wurden für unterschiedliche Frequenzen durchgeführt. Für die weiter unten beschriebenen Tests wird eine Periodendauer von 20ms angenommen.

Durchführung der Tests

Die Übertragung der Test-Frames wurde stets mit dem Oszilloskop aufgezeichnet, um mögliche Fehlerquellen schnell eingrenzen zu können. Da zuerst der Empfänger implementiert und getestet wurde, musste ein kleines Testprogramm geschrieben werden, das nur einige wenige Sendefunktionen übernehmen konnte. Nachdem die Empfangsfunktionen getestet und korrigiert wurden, konnte der Sendevorgang implementiert werden.

Für die Tests des MOSES-Programmes ist das verwendete Flag-Byte von besonderem Interesse, da dies fast alle nötigen Informationen enthält.

Außerdem werden Fehlerkennungen ausgegeben.

Das übertragene Datenbyte entspricht im Regelfall den hexadezimalen Wert 0x55.

Empfang: Allgemeiner Test

Im ersten Test wurde ein Frame ohne die Simulierung irgendwelcher Fehler übertragen. Das empfangene Datenbyte wird in der Variable `_serBuf` abgelegt.

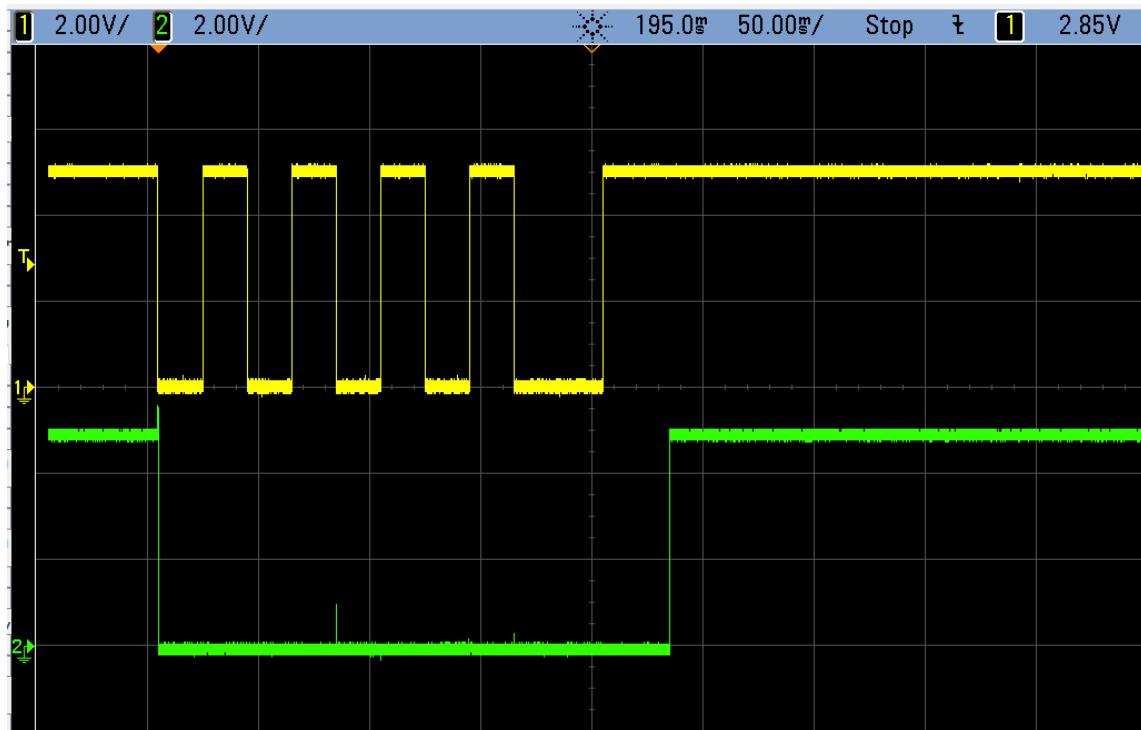


Abbildung 12 Übertragung gesamt

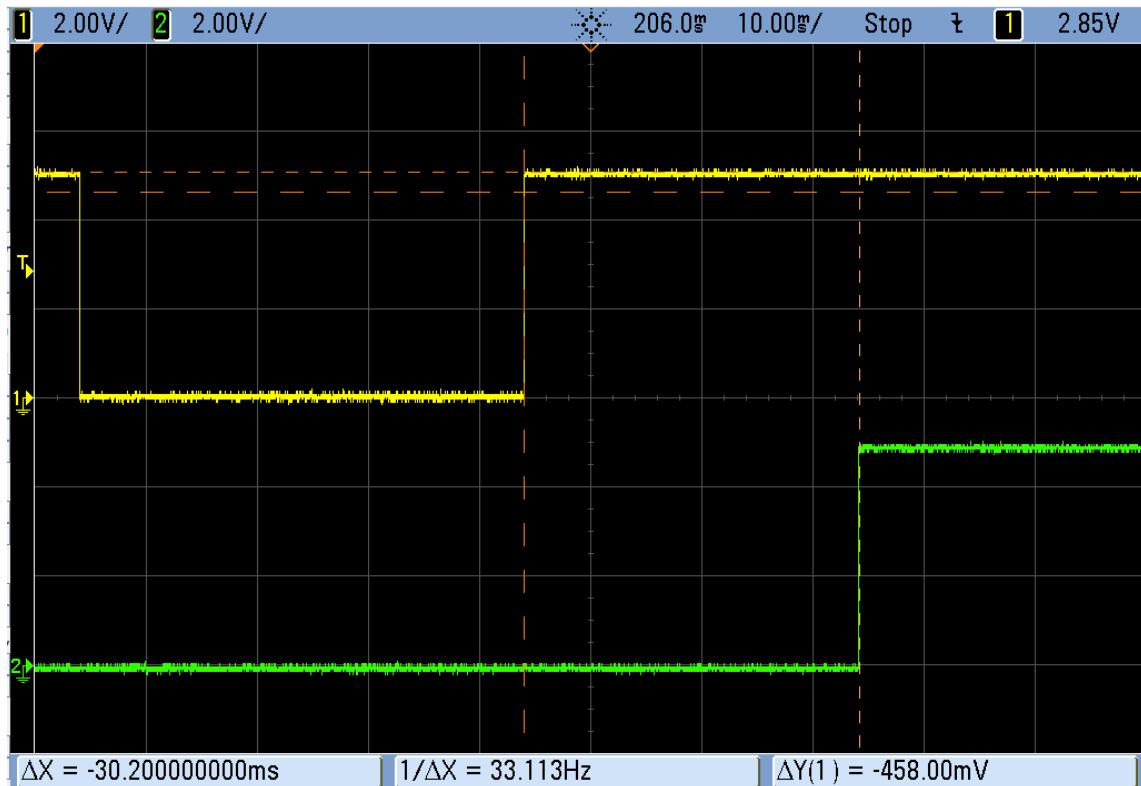


Abbildung 13 Zeitmessung zwischen Stop- und ACK-Bit

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

Nach dem Einlesen der einzelnen Datenbits und der Berechnung der Parität muss das Acknowledge-Bit auf die Leitung geschrieben werden. Bei einer positiven Quittierung muss der Empfänger 30ms nach der steigenden Flanke des Stop-Bits auch seine Sendeleitung in den Zustand logisch 1 bringen (wie in der obigen Grafik zu sehen).

Status-Byte:

Bit	Wert	Beschreibung
0	1	Status Flag der CB1-Leitung
1	1	Empfangsmodus erlaubt
2	0	Empfangsmodus aktiv
3	0	Sendemodus aktiv
4	0	StartBit
5	1	StopBit
6	0	Parität
7	0	Fehlerkennung

Bit 0 gibt den aktuellen Zustand auf der Leitung CB1 an. Da das Status-Byte nach dem Übertragungsvorgang eingelesen wird, ist der Zustand von CB1 auch korrekt.

Bit 1 gibt an, dass der Empfang erlaubt wird, ohne dieses Bit würde ansonsten nicht in den Empfangsmodus gewechselt.

Zusammenfassung:

Übertragener Frame: 01010101001

_serFlags:	Hex: 23	Bin: 00100011
_serBuf:	Hex: 55	Bin: 01010101

Fehlermeldung: Keine

Die letzten Schritte einer Übertragung

An dieser Stelle soll noch kurz die letzten Schritte einer Übertragung anhand einer Oszilloskopaufnahme erläutert werden.

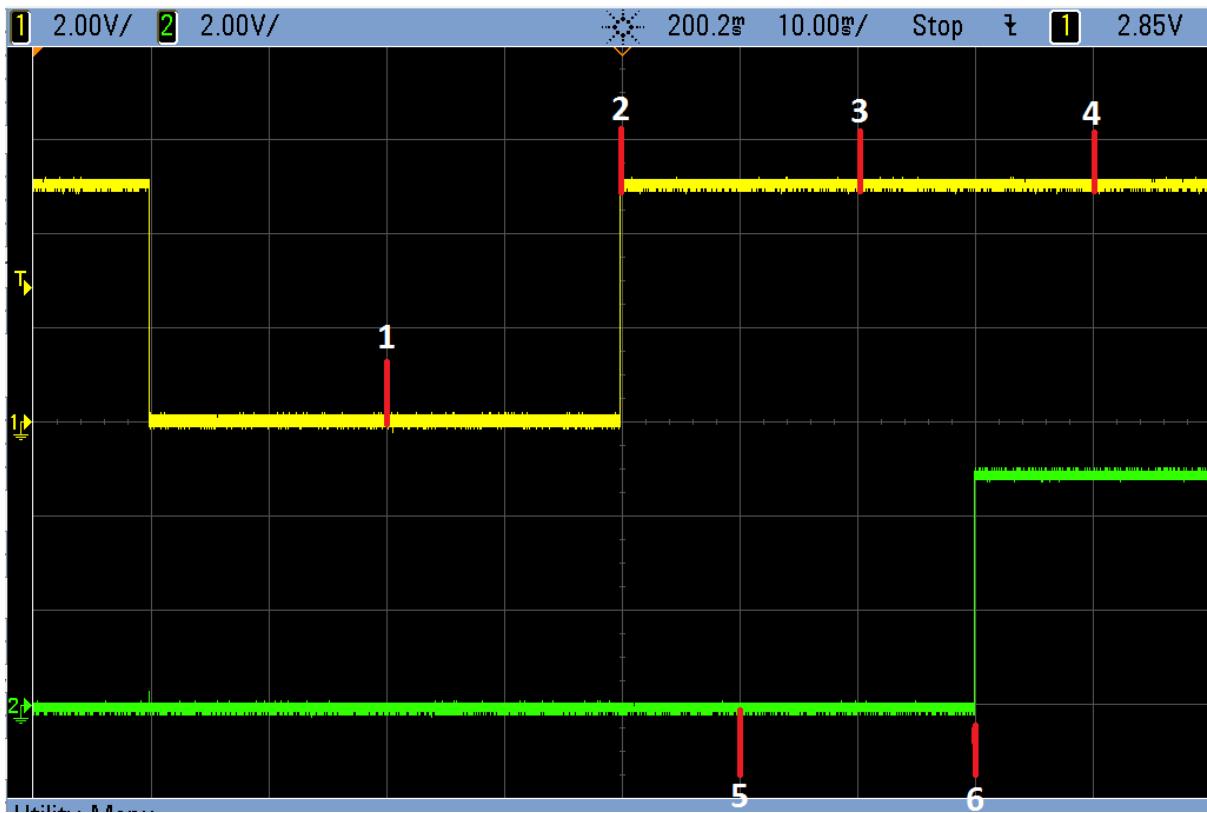


Abbildung 14 Letzen Schritte der Kommunikation

1. Sender schreibt Parität auf die Leitung
2. Sender schreibt StopBit auf die Leitung
3. Hier fragt der Sender seinen Eingang auf Null ab
4. Hier fragt der Sender den Erfolg der Übertragung ab (ACKBit)
5. Empfänger liest das StopBit
6. Empfänger schreibt den Erfolg der Übertragung auf die Leitung

Zeitlich korrekte Reihenfolge: 1, 2, 5, 3, 6, 4

Berechnung der Parität und andere Fehlerkontrollen muss der Empfänger zwischen 5-6 durchführen.

Empfang: falsche Parität

In diesem Fall wird vom Sender das Datenbyte 0x54 übertragen. Da das Paritätsbit für diesen Fall falsch belegt ist, muss der Empfänger dieses Datenpaket negativ quittieren.

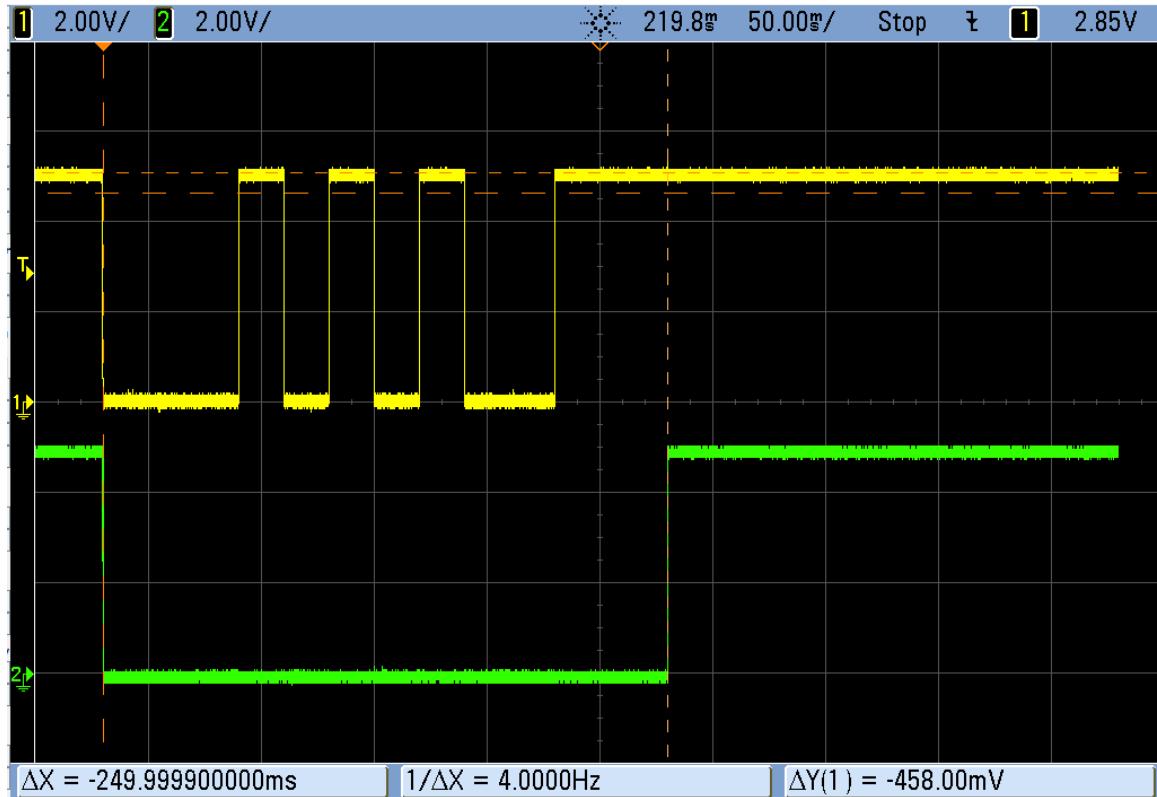


Abbildung 15 Übertragung eines Frames mit Paritätsfehler

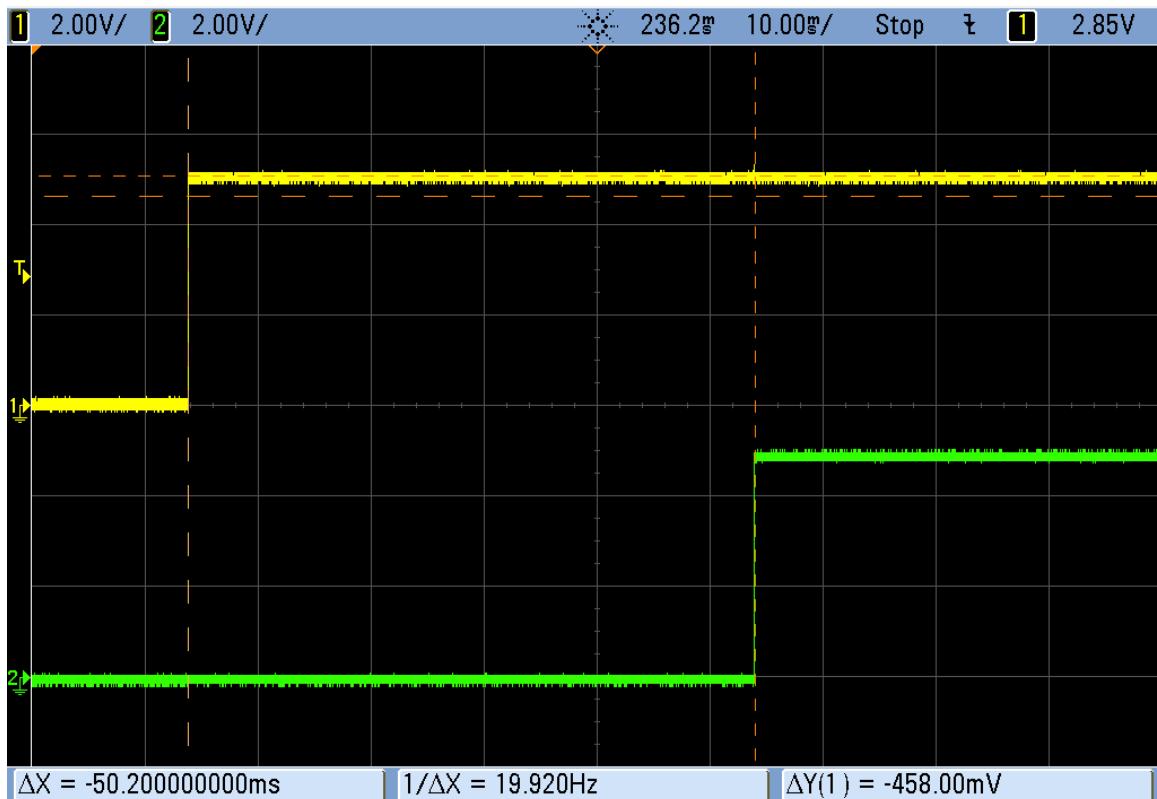


Abbildung 16 Negative Quittierung

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

Wie man obiger Abbildung entnehmen kann, wechselt der Empfänger erst 50ms nach steigender Flanke des Stop-Bits in den Zustand Idle. Das ACK-Bit ist also logisch 0 und der Paritätsfehler wurde demnach vom Empfänger erkannt.

Status-Byte:

Bit	Wert	Beschreibung
0	1	Status Flag der CB1-Leitung
1	1	Empfangsmodus erlaubt
2	0	Empfangsmodus aktiv
3	0	Sendemodus aktiv
4	0	StartBit
5	1	StopBit
6	0	Parität
7	1	Fehlerkennung

Damit die Übertragung korrekt verlaufen wäre, müsste das Paritäts-Bit auf 1 gesetzt werden. Richtigerweise wird daher im Bit 7 des Status-Bytes des Empfängers das Fehler-Flag gesetzt.

Zusammenfassung:

Übertragener Frame: 00010101001

_serFlags:	Hex: A3	Bin: 10100011
_serBuf:	Hex: 54	Bin: 01010100

Fehlernmeldung: Paritätsfehler

Empfang: Fehlerhaftes Stop-Bit

Das Stop-Bit dient auch zur Fehlererkennung, der eingelesene Wert muss also auch geprüft werden. In diesem Fall muss der Empfänger das falsche Stop-Bit erkennen und anschließend eine negative Quittierung absetzen.

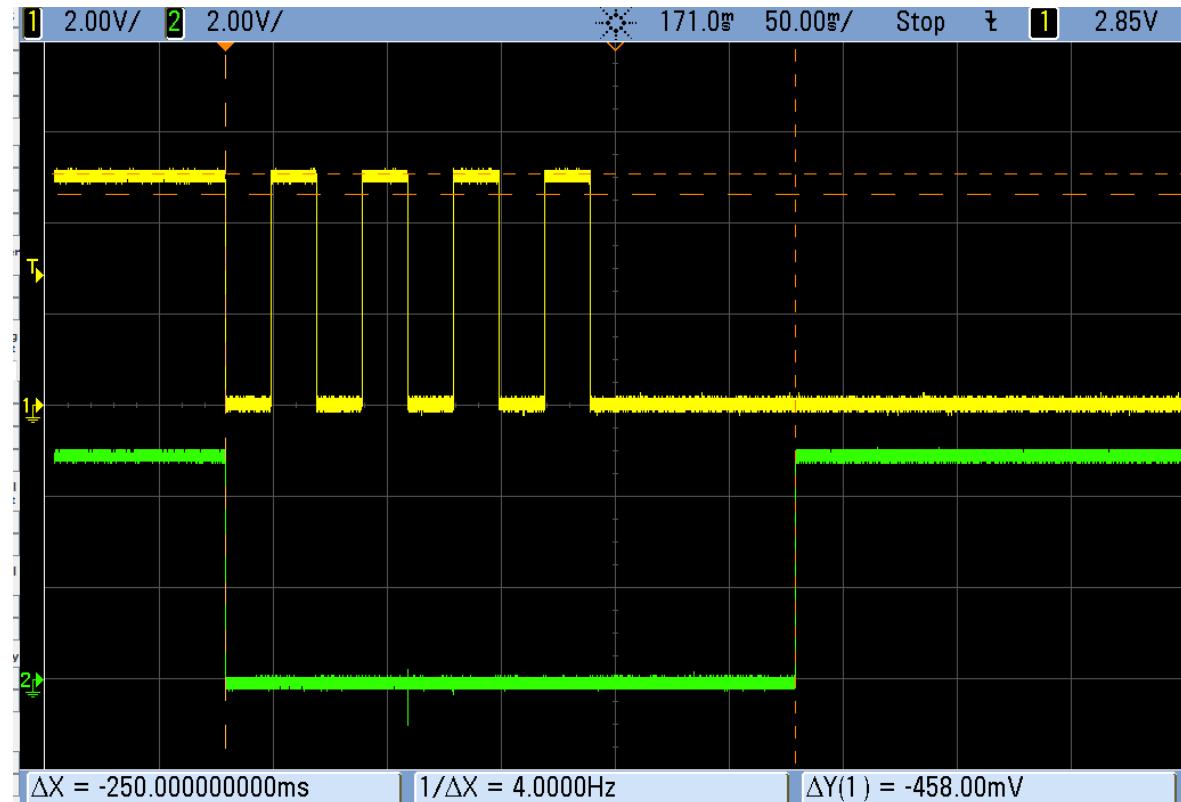


Abbildung 17 Fehlerhaftes Stop-Bit

Auch aus obiger Abbildung kann abgelesen werden, dass die Übertragung gescheitert ist. Die Zeit, die zwischen fallender und steigender Flanke des Empfängers vergeht, beträgt hier 250ms. Bei positiver Quittierung würde diese Zeit nur 230ms betragen. Das fehlerhafte Stop-Bit wurde also vom Empfänger erkannt. In diesem Fall wechselt der Sender auch gar nicht mehr in den Idle-Modus.

Status-Byte:

Bit	Wert	Beschreibung
0	0	Status Flag der CB1-Leitung
1	1	Empfangsmodus erlaubt
2	0	Empfangsmodus aktiv
3	0	Sendemodus aktiv
4	0	StartBit
5	0	StopBit
6	0	Parität
7	1	Fehlerkennung

Zusammenfassung:

Übertrager Frame: 01010101000

_serFlags: Hex: 82 Bin: 10000010

_serBuf: Hex: 55 Bin: 01010101

Fehlernmeldung: Stop-Bit ist auf 0

Empfang: Falsches Startbit

In der Regel ist es kaum möglich, dass ein fehlerhaftes Start-Bit eingelesen wird. Lediglich Störimpulse oder ähnliches können zu diesem Fehler führen. Nichtsdestotrotz sollte der Empfänger trotzdem das eingelesene Start-Bit auf den korrekten Wert prüfen.

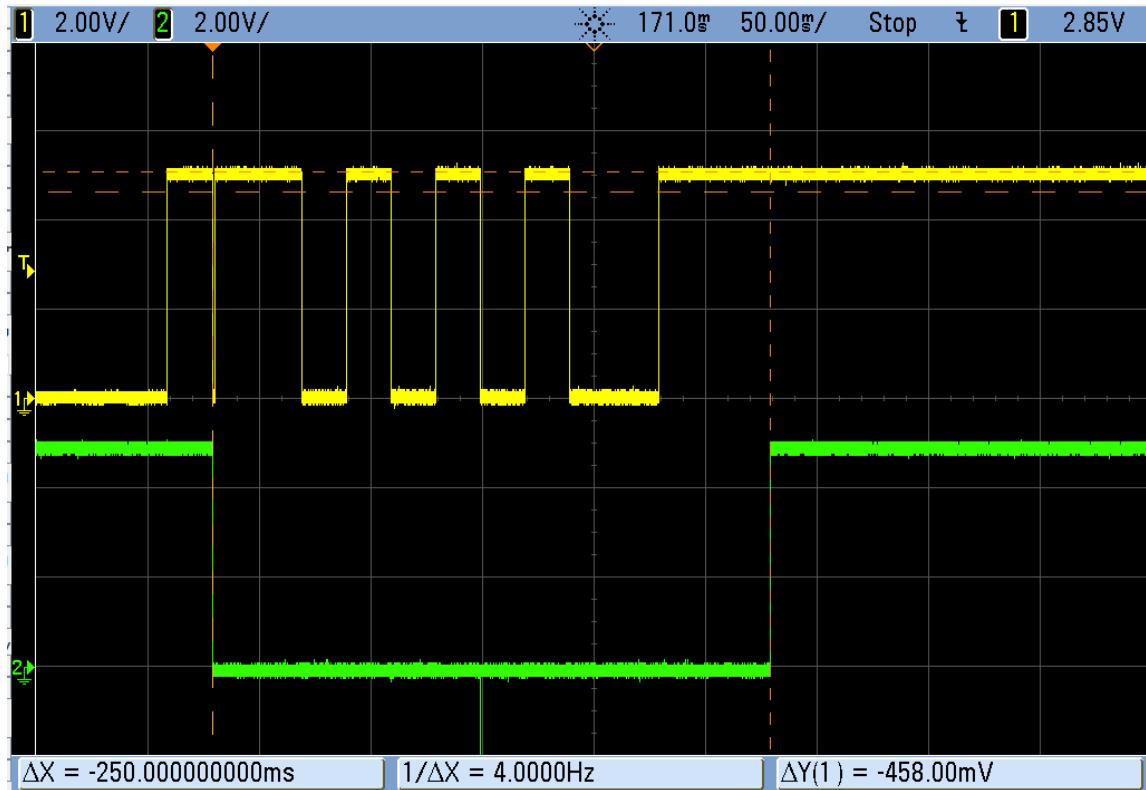


Abbildung 18 Fehlerhaftes Start-Bit

In der obigen Grafik wurde solch ein fehlerhaftes Start-Bit simuliert. Auf die erste fallende Flanke des Senders folgt sofort wieder eine steigende. Der Empfänger liest also zwangsläufig einen falschen Wert ein. Da auch hier die Zeit zwischen fallender und wieder steigender Flanke des Empfängers 250ms vergehen, ist die Quittierung negativ. Das fehlerhafte Start-Bit wurde erkannt.

Status-Byte:

Bit	Wert	Beschreibung
0	1	Status Flag der CB1-Leitung
1	1	Empfangsmodus erlaubt
2	0	Empfangsmodus aktiv
3	0	Sendemodus aktiv
4	1	StartBit
5	1	StopBit
6	0	Parität
7	1	Fehlerkennung

Zusammenfassung:

Übertragener Frame: 11010101001

_serFlags: Hex: B3 Bin: 10110011

_serBuf: Hex: 55 Bin: 01010101

Fehler: Startbit ist gleich 1

Empfang: Vollständig belegtes Empfangs-FIFO

Sollte das Empfangs-FIFO bereits vollständig gefüllt sein, so muss entsprechend reagiert werden.

Dazu gibt es zwei Möglichkeiten:

1. Nach der fallenden Flanke des Start-Bits wird nicht in den Empfangs-Modus gewechselt.
2. Nach der Übertragung der Datenbits wird anhand des ACK-Bits eine negative Quittierung auf die Leitung gegeben.

Für das MOSES-Programm wurde sich für die zweite Variante entschieden. Für den Test wurde ein Empfangs-FIFO mit einer maximalen Größe von 60 Elementen definiert. Beim 61. gesendeten Datenbyte muss also eine negative Quittierung erfolgen.

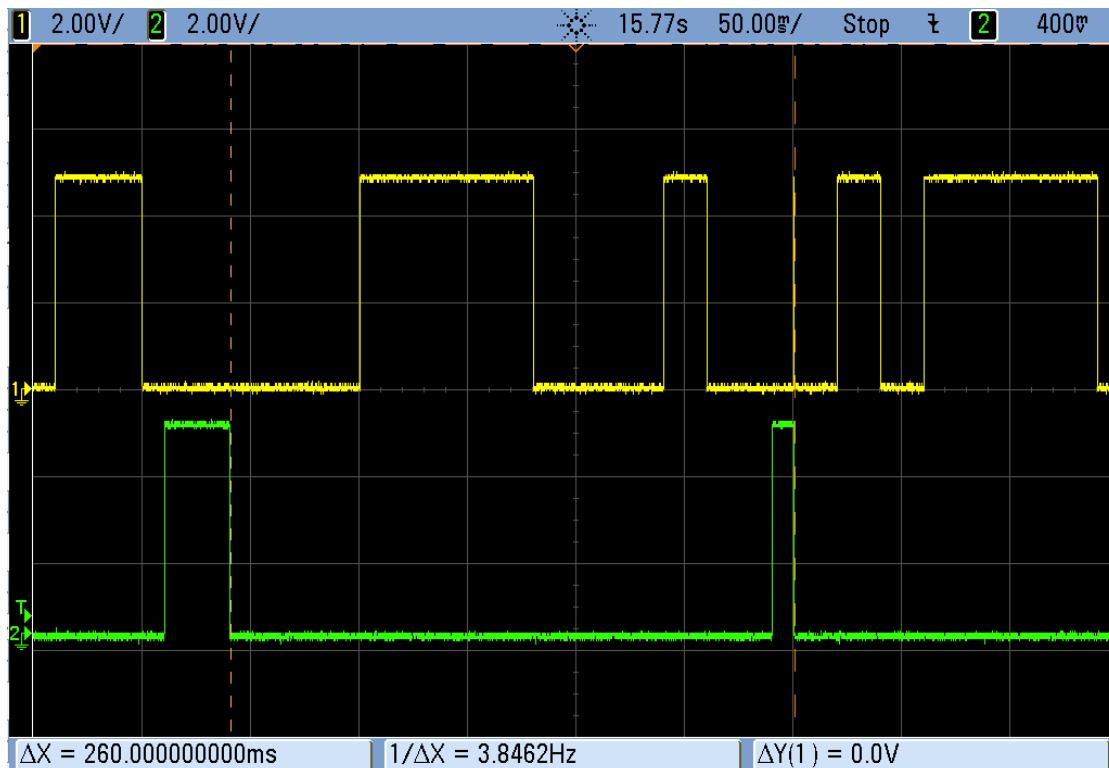


Abbildung 19 Vollständig belegtes Empfangs-FIFO

Status-Byte:

Bit	Wert	Beschreibung
0	1	Status Flag der CB1-Leitung
1	1	Empfangsmodus erlaubt
2	0	Empfangsmodus aktiv
3	0	Sendemodus aktiv
4	0	StartBit
5	1	StopBit
6	0	Parität
7	1	Fehlerkennung

Zusammenfassung:

Übertrager Frame: 00011110001

_serFlags: Hex: A3 Bin: 10100011

_serBuf: Hex: 3C Bin: 00111100

Fehler: Empfangs-FIFO kann keinen weiteren Wert aufnehmen

Senden: Allgemeiner Test

Beim Sender müssen nicht ganz so viele Fehler-Fälle geprüft werden. Der Empfänger sollte zu diesem Zeitpunkt bereits fehlerfrei arbeiten. Für Sender und Empfänger wurde jeweils ein MOSES verwendet.

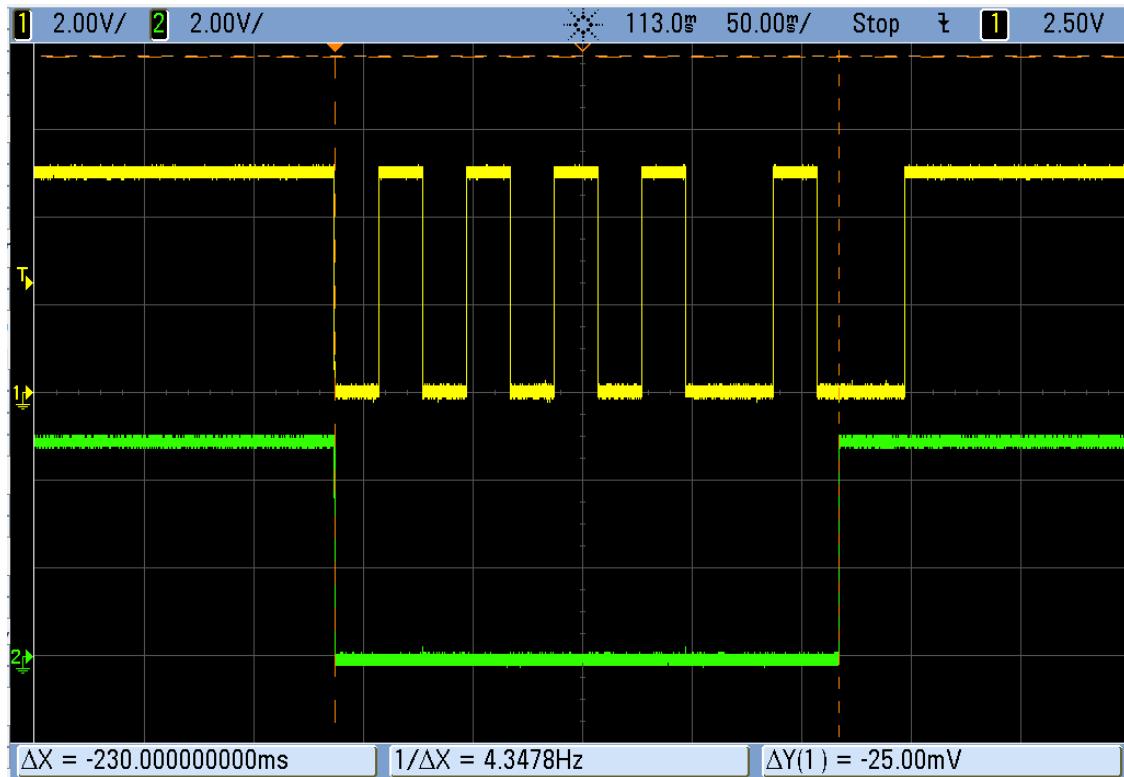


Abbildung 20 Allgemeiner Test: Vollständiger Sendevorgang

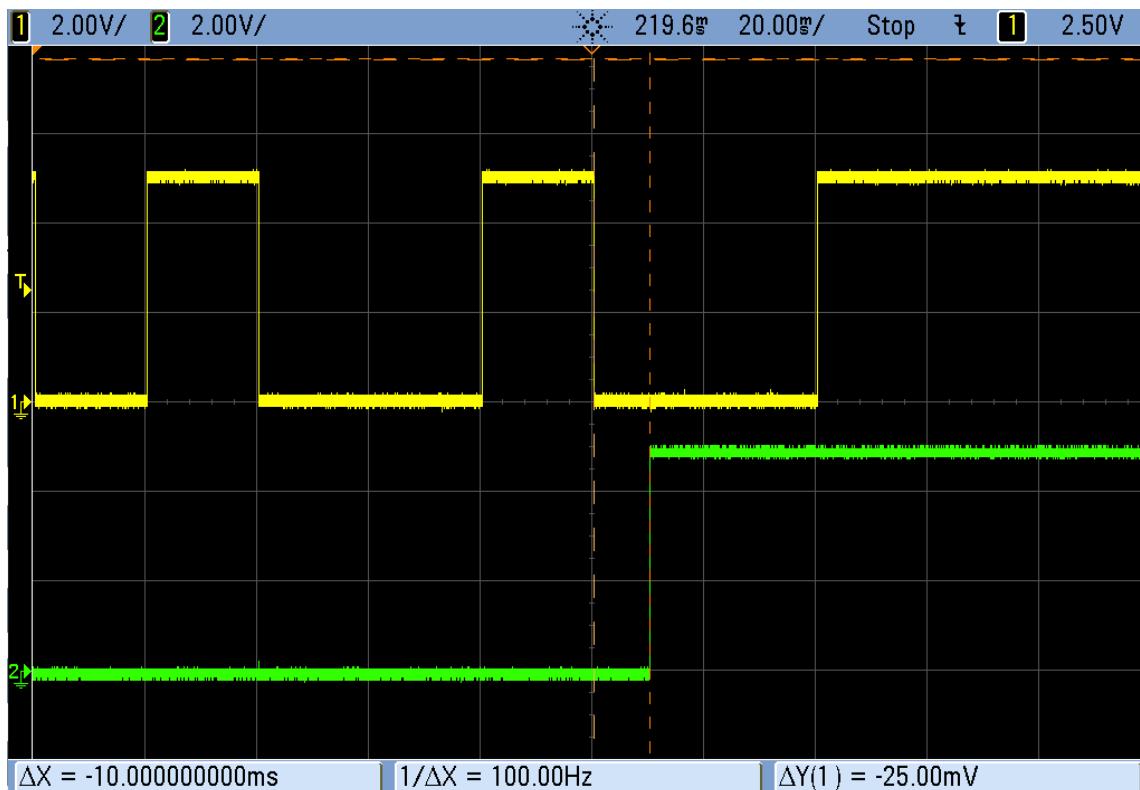


Abbildung 21 Zoom zum letzten Teil der Übertragung

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

Der Empfänger gibt eine positive Quittierung auf seine Leitung, der Sendevorgang ist also aus Empfängersicht korrekt verlaufen. Die Betrachtung der Status-Flags kann noch weitere Details liefern:

Status-Byte des Senders:

Bit	Wert	Beschreibung
0	1	Status Flag der CB1-Leitung
1	1	Empfangsmodus erlaubt
2	0	Empfangsmodus aktiv
3	0	Sendemodus aktiv
4	0	StartBit
5	1	StopBit
6	0	Parität
7	0	Fehlerkennung

Zusammenfassung Sender:

Übertragener Frame: 01010101001
 _serFlags: Hex: 23 Bin: 00100011
 _serBuf: Hex: 55 Bin: 01010101
 Fehlermeldung: Keine

Status-Byte des Empfängers:

Bit	Wert	Beschreibung
0	1	Status Flag der CB1-Leitung
1	1	Empfangsmodus erlaubt
2	0	Empfangsmodus aktiv
3	0	Sendemodus aktiv
4	0	StartBit
5	1	StopBit
6	0	Parität
7	0	Fehlerkennung

Zusammenfassung Empfänger:

_serFlags: Hex: 23 Bin: 00100011
 _serBuf: Hex: 55 Bin: 01010101
 Fehlermeldung: Keine

Die Status-Flags von Sender und Empfänger stimmen überein, ein Fehler wurde in beiden Fällen nicht festgestellt. Auch die Bits 4, 5 und 6 sind wie vorgesehen belegt.

Beachte: Die Flags müssen nicht in zwangsläufig übereinstimmen (siehe z.B. Bit 1).

Senden: Verpasstes / Ignoriertes Start-Bit

Der Empfänger muss nicht zwangsläufig auf eine Sende-Anfrage reagieren, er muss also nicht in den Empfangsmodus wechseln. Der Sender muss also prüfen, ob der Empfänger tatsächlich nach der fallenden Flanke des Start-Bits in den Empfangsmodus wechselt. Ist dies nicht der Fall, muss die Übertragung sofort abgebrochen werden. Dies ist notwendig, da der Empfänger sonst weitere fallende Flanken der Datenbits als Start-Bit interpretieren würde.

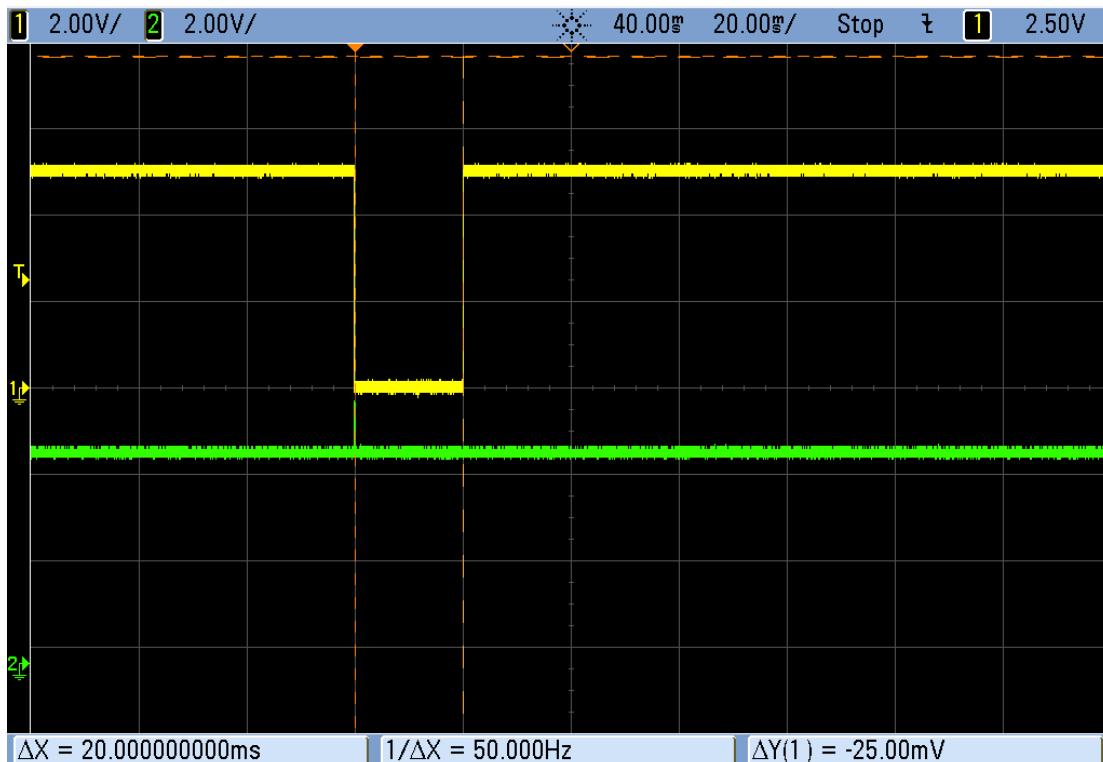


Abbildung 22 Verpasstes / Ignoriertes Start-Bit

Nach dem Start-Bit prüft der Sender, ob der Empfänger in den Empfangsmodus wechselt. Dies ist nicht der Fall, der Sender wechselt also sofort wieder in den Zustand Idle.

Status-Byte des Senders:

Bit	Wert	Beschreibung
0	1	Status Flag der CB1-Leitung
1	1	Empfangsmodus erlaubt
2	0	Empfangsmodus aktiv
3	0	Sendemodus aktiv
4	0	StartBit
5	0	StopBit
6	0	Parität
7	1	Fehlerkennung

Zusammenfassung:

Übertragener Frame: 01010101001

_serFlags: Hex: 83 Bin: 10000011

_serBuf: Hex: 55 Bin: 01010101

Fehlermeldung: Empfänger wechselt nicht in Empfangsmodus oder beendet diesen zu früh

Senden: Frühzeitiges Verlassen des Empfangsmodus

Ein weiteres Problem kann darin bestehen, dass der Empfänger den Empfangsmodus zu früh verlässt. Wechselt die TxD-Leitung des Empfängers vor dem ACK-Bit wieder in den Zustand logisch 1, muss der Sender dies als Fehler behandeln und die Übertragung abbrechen.

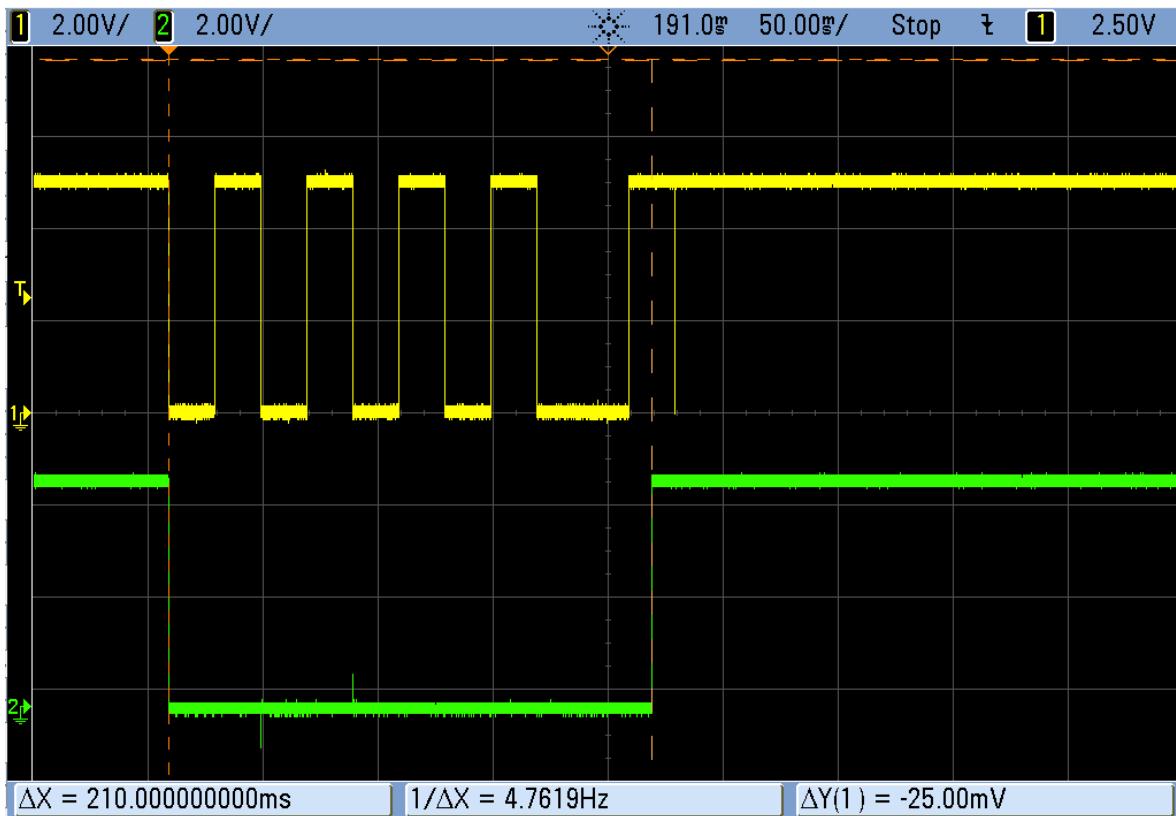


Abbildung 23 Frühzeitiges Verlassen des Empfangsmodus (1)

Wie man obiger Abbildung entnehmen kann, beträgt die Zeit zwischen fallender und steigender Flanke auf der TxD-Leitung des Empfängers 210ms. Laut Schnittstellendefinition ist dies jedoch nicht möglich. Es müssen mindestens 230ms zwischen diesen Flanken liegen. Der Sender muss also davon ausgehen, dass der Empfänger den Empfangsmodus vorzeitig verlassen hat, die Übertragung ist gescheitert.

In diesem Bild ist aber noch ein kleiner Fehler des Senders zu erkennen, der korrigiert werden musste: Nach dem Stop-Bit ist noch ein kurzeitiger Impuls auf der TxD-Leitung zu erkennen. Damit dies nicht zu Problemen führt, wurde das Programm nachträglich entsprechend angepasst (s.u.).

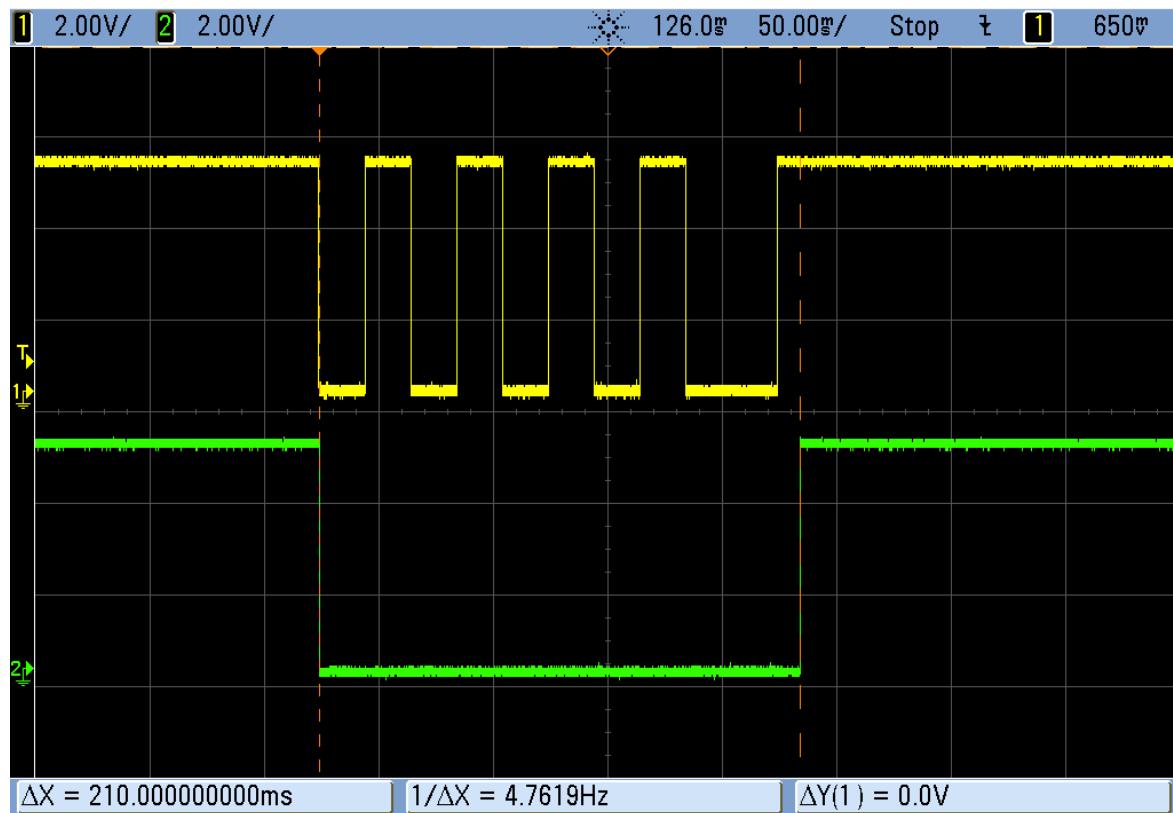


Abbildung 24 Frühzeitiges Verlassen des Empfangsmodus(2)

Nach der Korrektur des Programmes tritt der störende Impuls nicht mehr auf.

Status-Byte des Senders:

Bit	Wert	Beschreibung
0	1	Status Flag der CB1-Leitung
1	1	Empfangsmodus erlaubt
2	0	Empfangsmodus aktiv
3	0	Sendemodus aktiv
4	0	StartBit
5	1	StopBit
6	0	Parität
7	1	Fehlerkennung

Zusammenfassung:

Übertragener Frame: 01010101001

_serFlags: Hex: A3 Bin: 10000011

_serBuf: Hex: 55 Bin: 01010101

Fehlermeldung: Empfänger wechselt nicht in Empfangsmodus oder beendet diesen zu früh.

Senden: Negative Quittierung des Empfängers

Sollten beim Empfänger Fehler während der Übertragung auftreten (falsche Parität etc.), so meldet er dies mit einer negativen Quittierung (NACK) mithilfe des ACK-Bits. Der Sender muss dies feststellen können, um auf die Fehlermeldung entsprechend reagieren zu können.

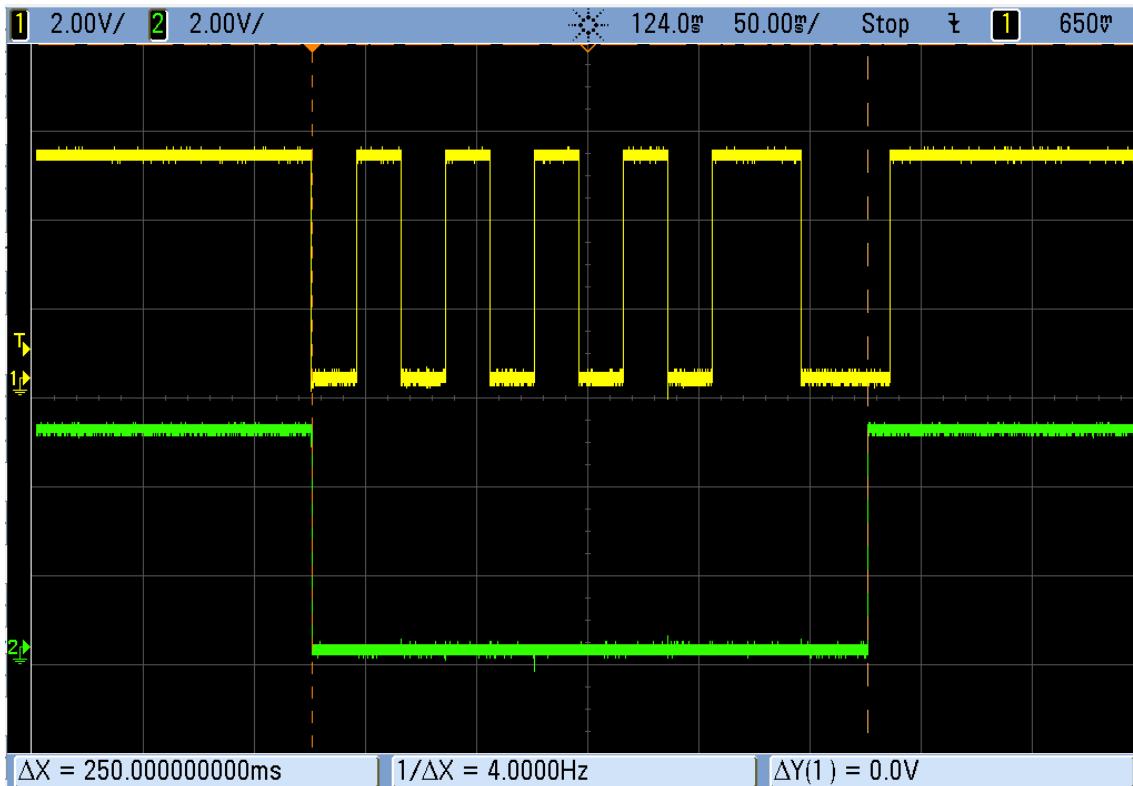


Abbildung 25 Empfänger meldet NACK (Negative Quittierung)

Die TxD-Leitung des Empfängers ist genau für 250ms auf logisch 0, die Quittierung ist demnach negativ. Hier wurde vor der Übertragung des Senders das Paritäts-Bit manipuliert, sodass beim Empfänger ein Paritätsfehler auftritt.

Status-Byte des Senders:

Bit	Wert	Beschreibung
0	1	Status Flag der CB1-Leitung
1	1	Empfangsmodus erlaubt
2	0	Empfangsmodus aktiv
3	0	Sendemodus aktiv
4	0	StartBit
5	1	StopBit
6	1	Parität
7	1	Fehlerkennung

Zusammenfassung:

Übertragener Frame: 01010101011

_serFlags: Hex: E3 Bin: 11100011

_serBuf: Hex: 55 Bin: 01010101

Fehlermeldung: Negative Quittierung des Empfängers

Senden / Empfangen: Test des Swap-Bits

Haben beide Kommunikationspartner eine größere Zahl an Bytes, die an den jeweils anderen übertragen werden sollen, so sieht die Schnittstellendefinition vor, dass nach jedem Übertragungsvorgang Sender und Empfänger ihre Rolle tauschen. Dies wird unter anderem mithilfe des Swap-Bits realisiert.

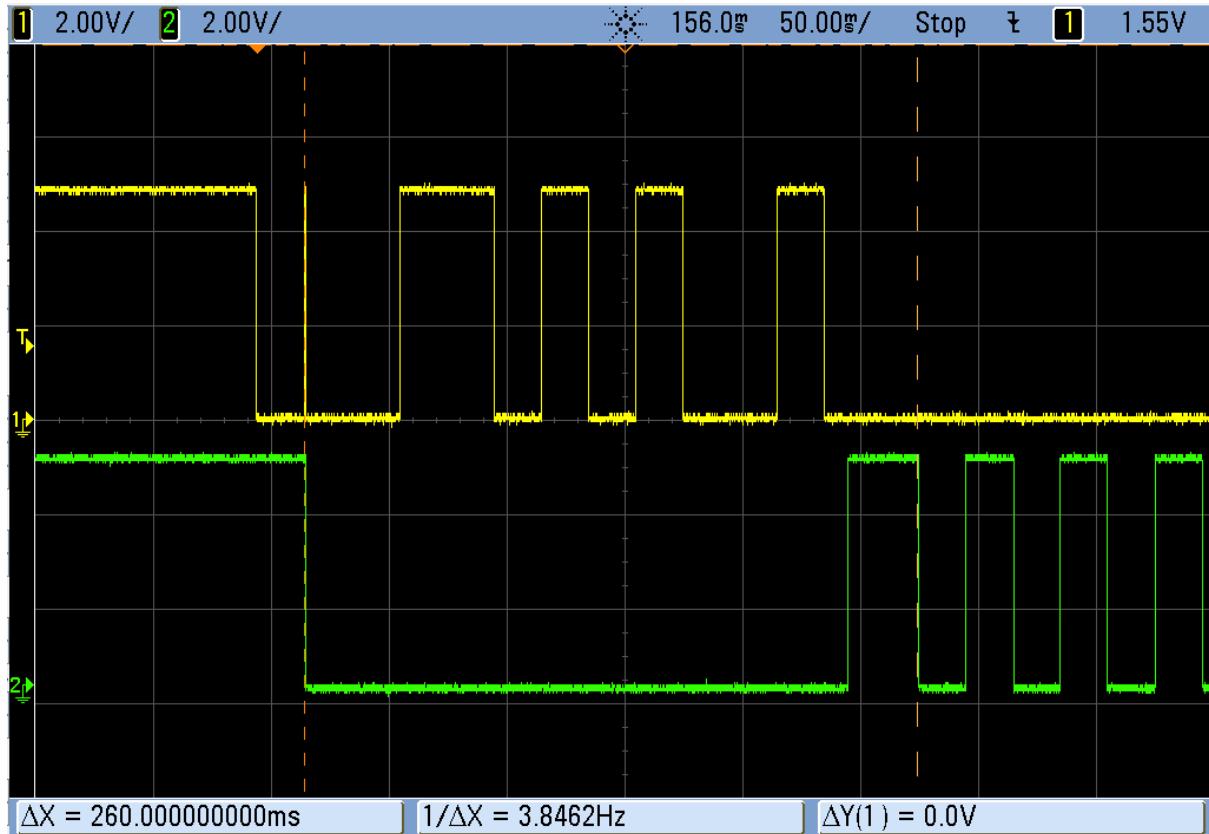


Abbildung 26 Abwechselndes Senden / Empfangen (1)

In der obigen Grafik beginnt Gelb mit der Übertragung des ersten Datenbytes. Die fallende Flanke des Start-Bits wird von Grün erst nach dem zweiten Anlauf erkannt. Im weiteren Verlauf treten aber keine weiteren Probleme bei der Übertragung durch Gelb auf.

Anschließend wechselt Grün in den Sendemodus und beginnt die Übertragung.

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

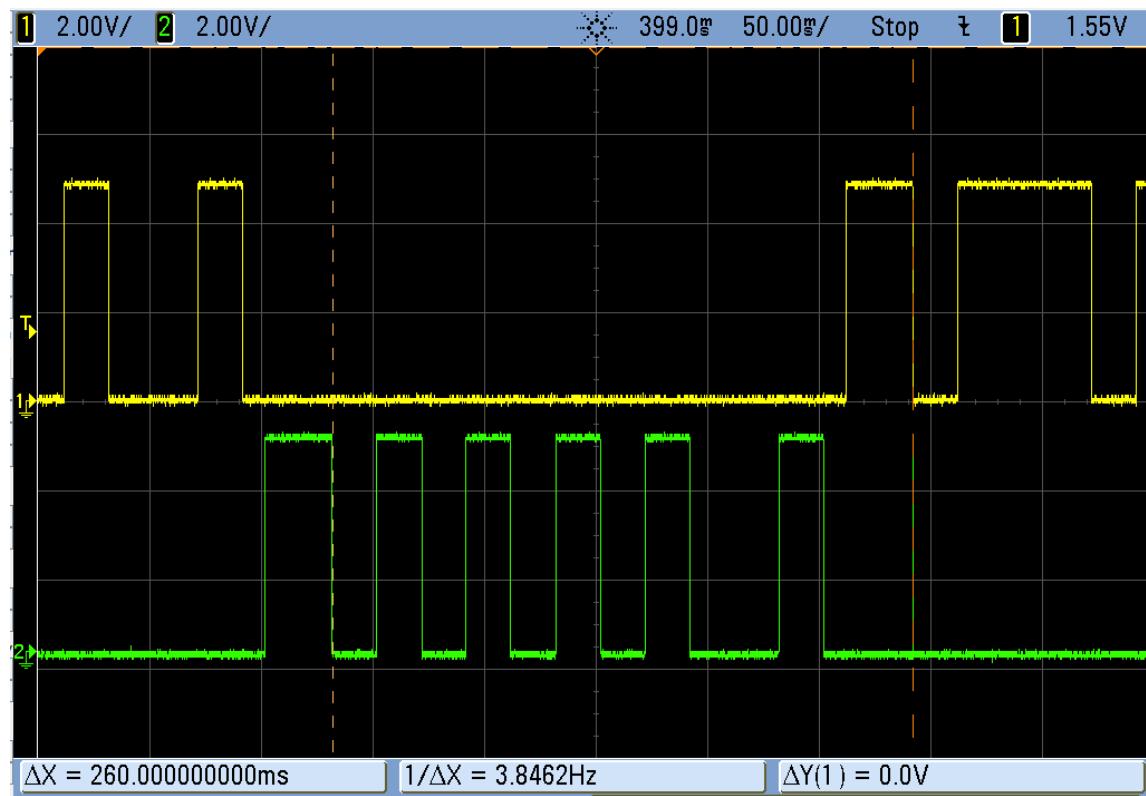


Abbildung 27 Abwechselndes Senden / Empfangen (2)

Nachdem das Datenbyte von Grün erfolgreich übertragen ist, wechseln beide wieder die Rollen.

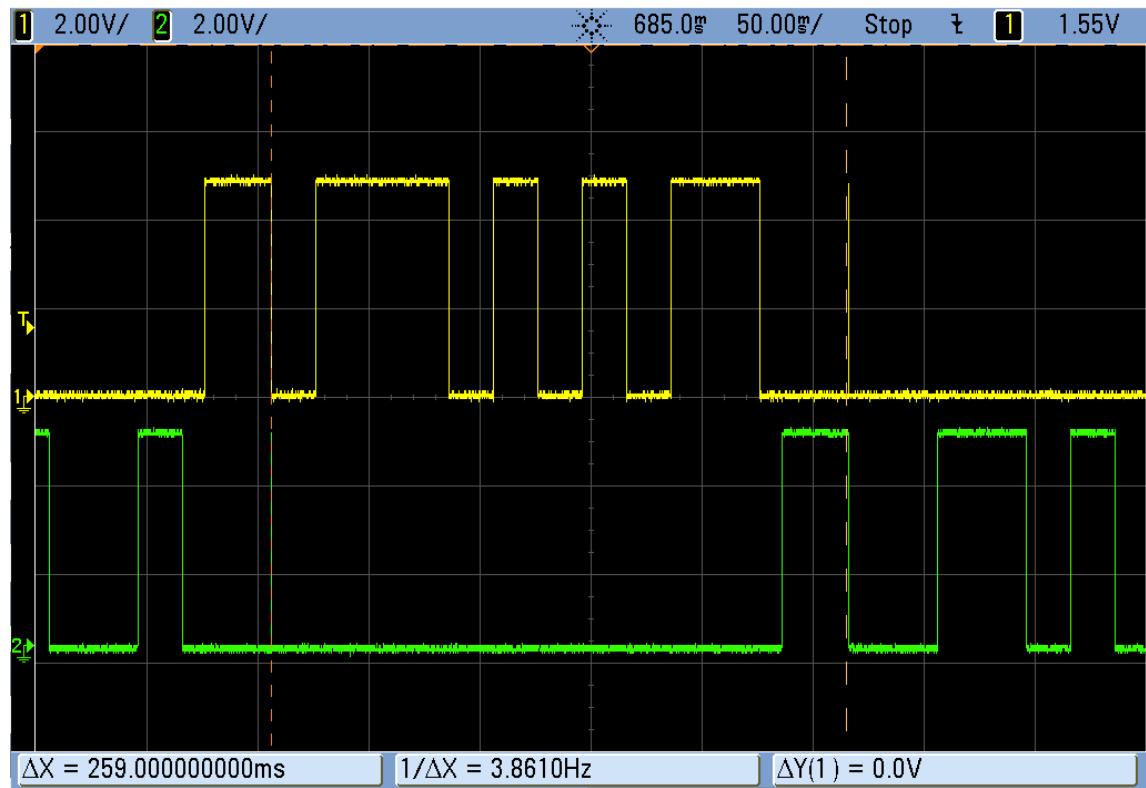


Abbildung 28 Abwechselndes Senden / Empfangen (3)

Das abwechselnde Senden wird ausgeführt, bis einer der beiden Kommunikationspartner über keine zu sendenden Daten mehr verfügt.

12. Anhang

Für die Assemblerprogramme des MOSES wurden auch Programmablaufpläne erstellt, für die einige Konventionen zu beachten sind:

- Abkürzungen für Akkumulator, X- und Y-Register: Akku, X, Y
- Konstanten werden mit vorangestellten Rauten (#) gekennzeichnet (z.B. #10)
- Konstanten können auch in verschiedenen Zahlensystemen angeben sein (z.B. #10 im Dezimal-, #%1010 im Dual und \$A im Hexadezimalsystem)
- Sonstige Bezeichner für Variablen
- Lade- und Transfer und Stackoperationen: Darstellung mit einem Pfeil (z.B. #5 → X, Akku → Stack, X → Y)
- Nachindizierte Adressierung: Verwendung von eckigen Klammern (z.B. hFeld[X])
- Indirekte Adressierung: *(p + X) für die vorindizierte Variante und (*p)[Y] für die nachindizierte Variante
- Die Adresse einer Variablen wird mit eckigen Klammern angegeben (z.B. [Variable])
- Inkrementierung und Dekrementierung angelehnt an die C-Syntax (z.B. variable++, X--)
- Additionen und Subtraktionen sowie UND-, ODER-, XOR-Operationen werden immer nur auf den Akkumulator ausgeführt, dies wird daher nicht noch extra angegeben
- Um Platz zu sparen, können mehrere Operationen in einem Vorgangs-Block enthalten sein

Weiterhin sind einige Programmablaufpläne (insbesondere der seriellen Schnittstelle) sehr allgemein gehalten, da die Übersichtlichkeit leiden würde. Das Verständnis solcher umfangreicher PAPs wäre ohnehin nicht ohne weiteres möglich.

Beispielanwendungen für MOSES und SPS

Beispiel für ein Sendeprogramm

Das nachfolgende Programm verdeutlicht das Zusammenspiel der einzelnen Funktionen. Es überträgt eine Folge von Zahlen von 0 - 99 (0x63). Ein Sendevorgang kann durch den Wechsel des hintersten Schalters (Bit 0) gestartet werden. Mit dem zweiten Schalter wird das Programm beendet. Außerdem werden die Anzahl an Sendefehlern mitgezählt, die nach Programmende ausgelesen werden können.

```
.LIB "C:\SHELL\arbeit\KONST.lib"
.LIB "C:\SHELL\arbeit\PIAT.lib"
.LIB "C:\SHELL\arbeit\SPSMos.lib"

.ORG $4000

; Setze FIFO zurück
JSR _initFIFO
LDA #$02
JSR SPSIni
```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

loop:
  LDA #0

  ;Startwert des Datenbytes
  STA _serSndByte
  JSR DigEin
  CMP #2
  BEQ PEnde

  ;Schalterstellung = 2 beendet Programm
  CMP #1
  BNE loop

  ;SchalterStellung = 1 startet Übertragung
  ;Testfolge senden
  JSR DigAus

sndLoop:
  LDA _serSndByte
  CMP #$64

  ;ABBRUCHBEDINGUNG
  BEQ endLoop
  LDX #0
  JSR SPSAus
  INC _serSndByte

  ;Fehler?
  TXA
  BEQ sndLoop
  INC _serNumErrs
  BNE sndLoop

endLoop:
  JSR DigEin
  CMP #0
  BEQ loop
  AND #2
  BEQ endLoop
PEnde:
  RTS

  .ORG $4800
;=====
;Debugs / Tests
_serNumErrs: .BYTE $00
_serSndByte: .BYTE $00
;/Debug/Tests
=====
```

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

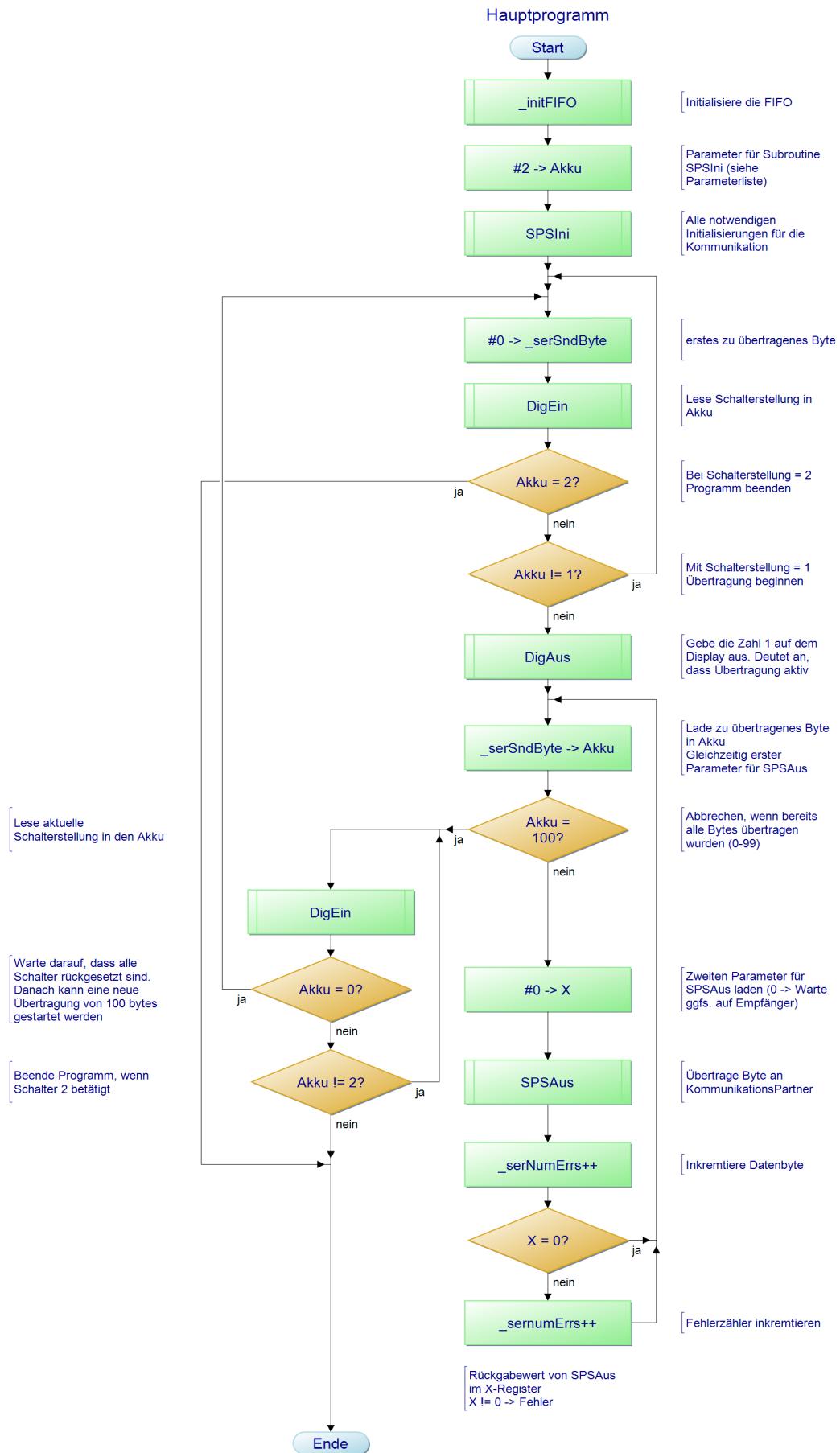


Abbildung 29 PAP Hauptprogramm MOSES

Beispiel für ein Empfangsprogramm

Mit dem folgenden Programm kann ein Empfänger simuliert werden. Auf die Anwenderanzeige wird die hintere Tetrade des zuletzt empfangen Bytes ausgegeben. Mit dem hintersten Schalter am MOSES kann das Programm beendet werden.

```
.LIB "C:\SHELL\arbeit\KONST.lib"
.LIB "C:\SHELL\arbeit\PIAT.lib"
.LIB "C:\SHELL\arbeit\SPSMos.lib"

.ORG $4000
;Setze FIFO zurück
JSR _initFIFO

LDA #$02
JSR SPSIni ;Empfang erlauben

loop:
JSR SPSEin
CPX #0      ;Prüfe, ob Empfangs-FIFO leer
BNE loop_1
JSR DigAus
loop_1:
JSR DigEin
CMP #1
BNE loop

RTS
```

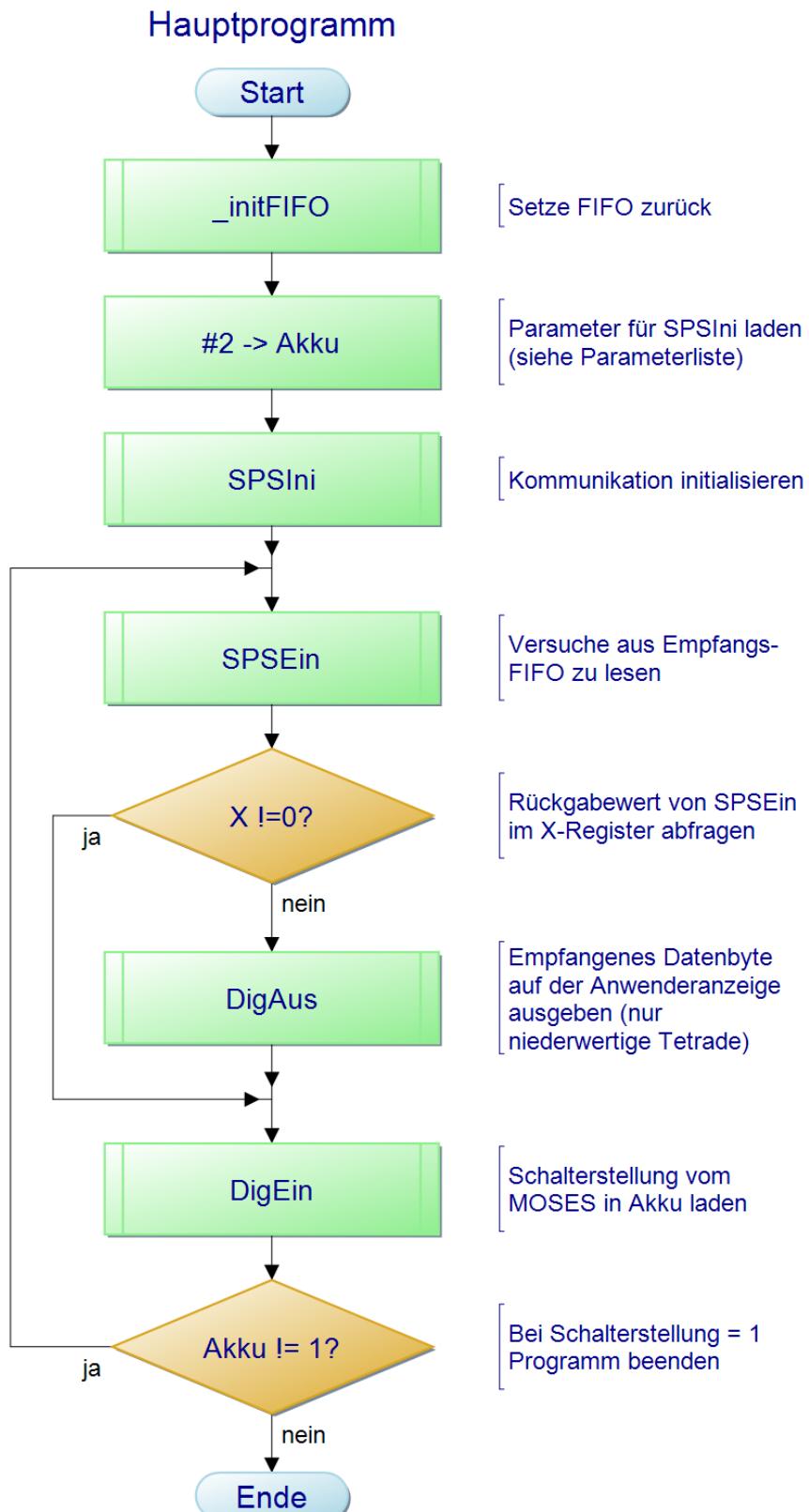


Abbildung 30 PAP Beispiel: Empfangsprogramm

Sendevorgänge mit Fehlerbehandlung

Diese Subroutine kann dazu verwendet werden, um ein Byte an den Kommunikationspartner zu senden. Der Unterschied zur Standardroutine besteht darin, dass bei fehlgeschlagener Übertragung max. 10 Wiederholungen stattfinden. Sollte auch der letzte Versuch fehlschlagen, wird eine Fehlerkennung im X-Register hinterlegt.

```

;=====
; Parameter:
;   Akku: zu übertragenes Datenbyte
; Rückgabewerte:
;   X-Register: 0 -> kein Fehler, ansonsten 1
;=====

SPSAusR:
    ;Sichere Akku auf den Stack
    PHA

    ;Sichere Y-Register auf den Stack
    TYA
    PHA
    LDY #0      ;Fehler-Zähler

_SPSAusR1:
    ;Hole Datenbyte in den Akku
    TSX
    LDA $0101, X

    ;Es soll auf den KP gewartet werden
    LDX #0
    JSR SPSAus

    ;Prüfe, ob Übertragung erfolgreich verlaufen ist
    CPX #0
    BEQ _SPSAusR2 ;kein Fehler, Verzweige

    ;Fehler
    INY
    CPY #10
    BEQ _SPSAusR3 ;Breche ab, wenn 10 Fehler erreicht
    BNE _SPSAusR1

_SPSAusR2:
    ;kein Fehler, Lade 0 ins X-Register
    LDX #0
    BEQ _SPSAusREnde

_SPSAusR3:
    ;Fehler, Lade Fehlerkennung ins X-Register
    LDX #1

_SPSAusREnde:
    ;Hole Y-Register vom Akku
    PLA
    TAY
    ;Hole gesicherten Wert vom Akku vom Stack
    PLA

    RTS
;=====

```

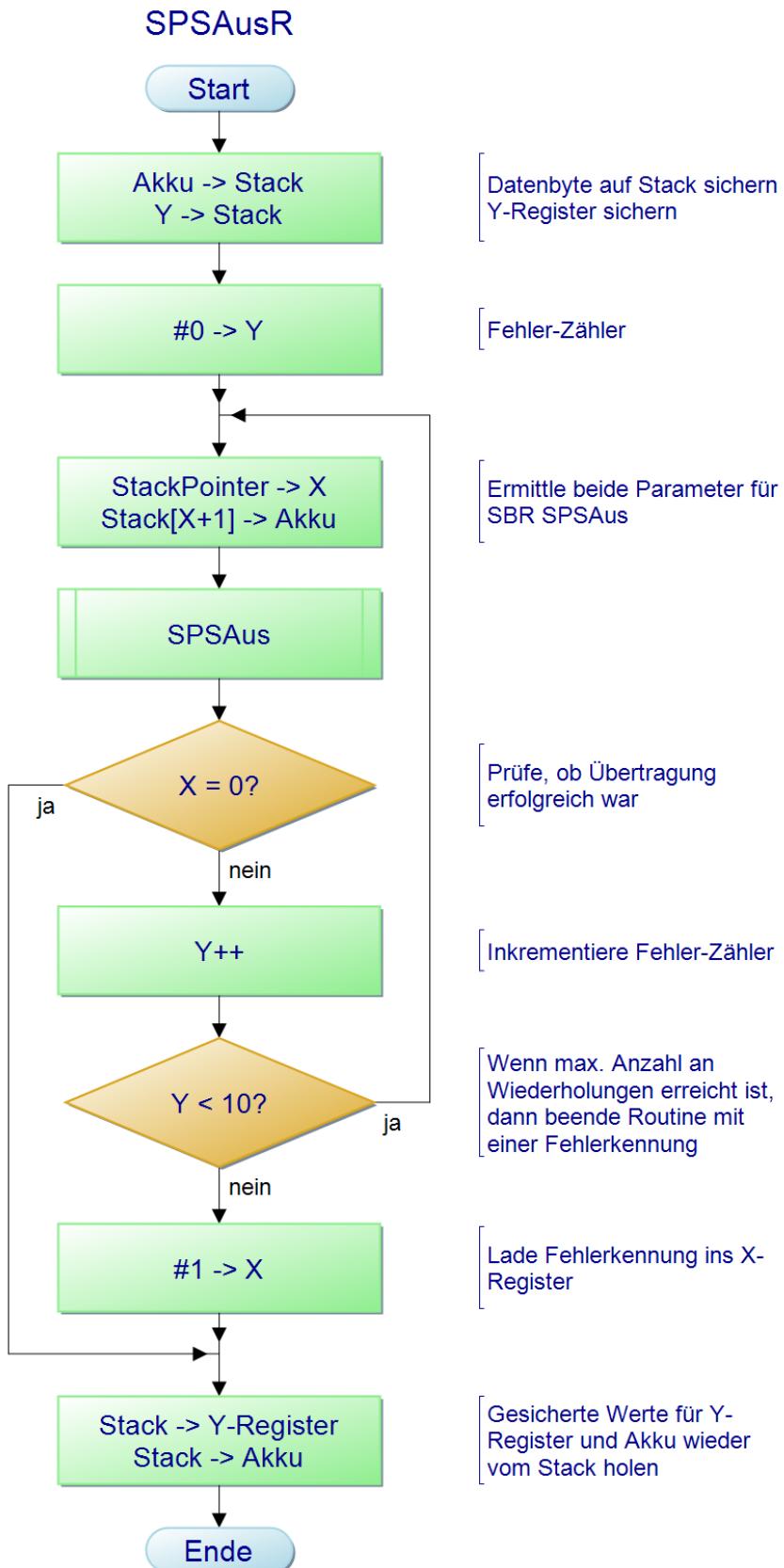


Abbildung 31 PAP Sichere Übertragung

Übertragung der aktuellen Schalterstellung am MOSES

Mit dem Aufruf der folgenden Subroutine wird die aktuelle Schalterstellung am MOSES an den Kommunikationspartner übertragen. Es werden maximal zehn Sendeversuche unternommen. Schlägt dies fehl, so bricht diese Routine mit einer Fehlerkennung im X-Register ab.

```

=====
; Parameter:
;   Keine
; Rückgabewerte:
;   X-Register: 0 -> kein Fehler, ansonsten 1
=====

SPSAusDigEin:
    ;sichere Akku auf den Stack
    PHA

    ;sichere Y-Register auf den Stack
    TYA
    PHA
    LDY #0 ;Fehler-Zähler

    _SPSAusDigEin1:
        JSR DigEin      ;Lese aktuelle Schalterstellung ein
        LDX #0
        JSR SPSAus      ;Versuche an KP zu senden

        ;Prüfe, ob Übertragung erfolgreich verlaufen ist
        CPX #0
        BEQ _SPSAusDigEinEnde ;kein Fehler, Verzweige

        ;Fehler
        INY
        CPY #10
        BEQ _SPSAusDigEin3 ;Breche ab, wenn 10 Fehler erreicht
        BNE _SPSAusDigEin1

    _SPSAusDigEin3:
        ;Fehler, Lade Fehlerkennung ins X-Register
        LDX #1

    _SPSAusDigEinEnde:
        ;Hole Y-Register vom Akku
        PLA
        TAY

        ;Hole gesicherten Wert vom Akku vom Stack
        PLA
        RTS
=====

```

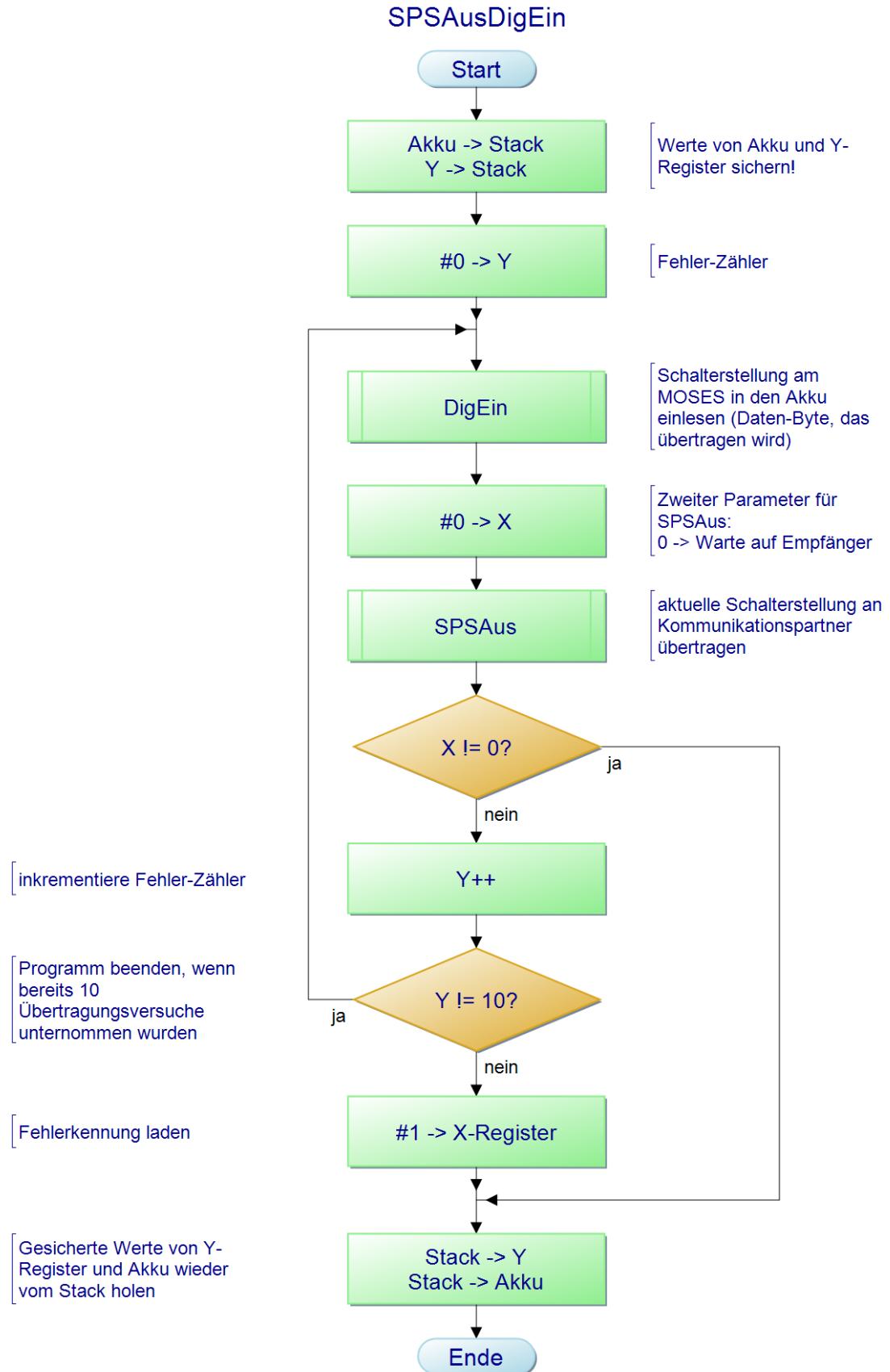


Abbildung 32 PAP SPSAusDigEin

Übertragung von Datenfeldern

Sendet ein ein Feld von Bytes an den Kommunikationspartner. Die Adresse des Feldes wird im Pointer `_serHP` hinterlegt. Das erste Bytes des Feldes enthält die Anzahl der zu übertragenen Bytes. Die Routine versucht jedes Byte max. 10-mal zu übertragen. Schlägt dies fehl, so bricht diese Routine mit einer Fehlerkennung im X-Register ab.

```

;=====
; Parameter:
;   keine
;Rückgabewerte:
;   X-Register: 0 -> kein Fehler, ansonsten Byte-Nr(>0) für die
;               Übertragung fehlgeschlagen ist
;=====

SPSAusFeld:
    ;sichere Akku auf den Stack
    PHA

    ;sichere Y-Register auf den Stack
    TYA
    PHA

    LDY #0 ;Schleifenvariable (Anzahl der Durchläufe)

    ;Lege Anzahl der zu übertragenen Bytes auf den Stack
    LDA (_serHP), Y
    PHA

    ;Fehler-Zähler als lokale Variable
    LDA #0
    PHA

_SPSAusFeld1:
    TSX
    ;Prüfe, ob alle Bytes übertragen wurden
    TYA
    CMP $0102, X
    BEQ _SPSAusFeld4

    INY ;Anzahl der übertragenen Bytes inkrementieren

    ;Setze Fehler-Zähler zurück
    LDA #0
    STA $0101, X
_SPSAusFeld2:
    LDA (_serHP), Y ;Datenbyte
    LDX #0           ;Warte auf Empfänger
    JSR SPSAus

    ;Prüfe, ob Übertragung erfolgreich verlaufen ist

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

CPX #0
BEQ _SPSAusFeld1 ;kein Fehler, verzweige

;Fehler
TSX
INC $0101, X
LDA #10
CMP $0101, X
BNE _SPSAusFeld2

_SPSAusFeld3:
;10 Fehler bei einer Übertragung, breche ab
TYA
TAX ;Fehlerkennung: Byte-Nr im hFeld
BNE _SPSAusFeldEnde

_SPSAusFeld4:
;Alle Bytes erfolgreich übertragen
LDX #0      ;Rückgabewert -> alles OK

_SPSAusFeldEnde:
;Hole lokale Variablen vom Akku
PLA
PLA

;Hole Y-Register vom Akku
PLA
TAY

;Hole gesicherten Wert vom Akku vom Stack
PLA
RTS
;-----

;=====
;Beispiel für ein Hauptprogramm
;=====
_serHP = $82

.ORG $4000
LDA #<hFeld
STA _serHP
LDA #>hFeld
STA _serHP + 1
JSR SPSAusFeld

.ORG $6000
hFeld: .BYTE $4, $1, $2, $3, $4, $5, $6, $7, $8, $9, $10
;-----

```

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

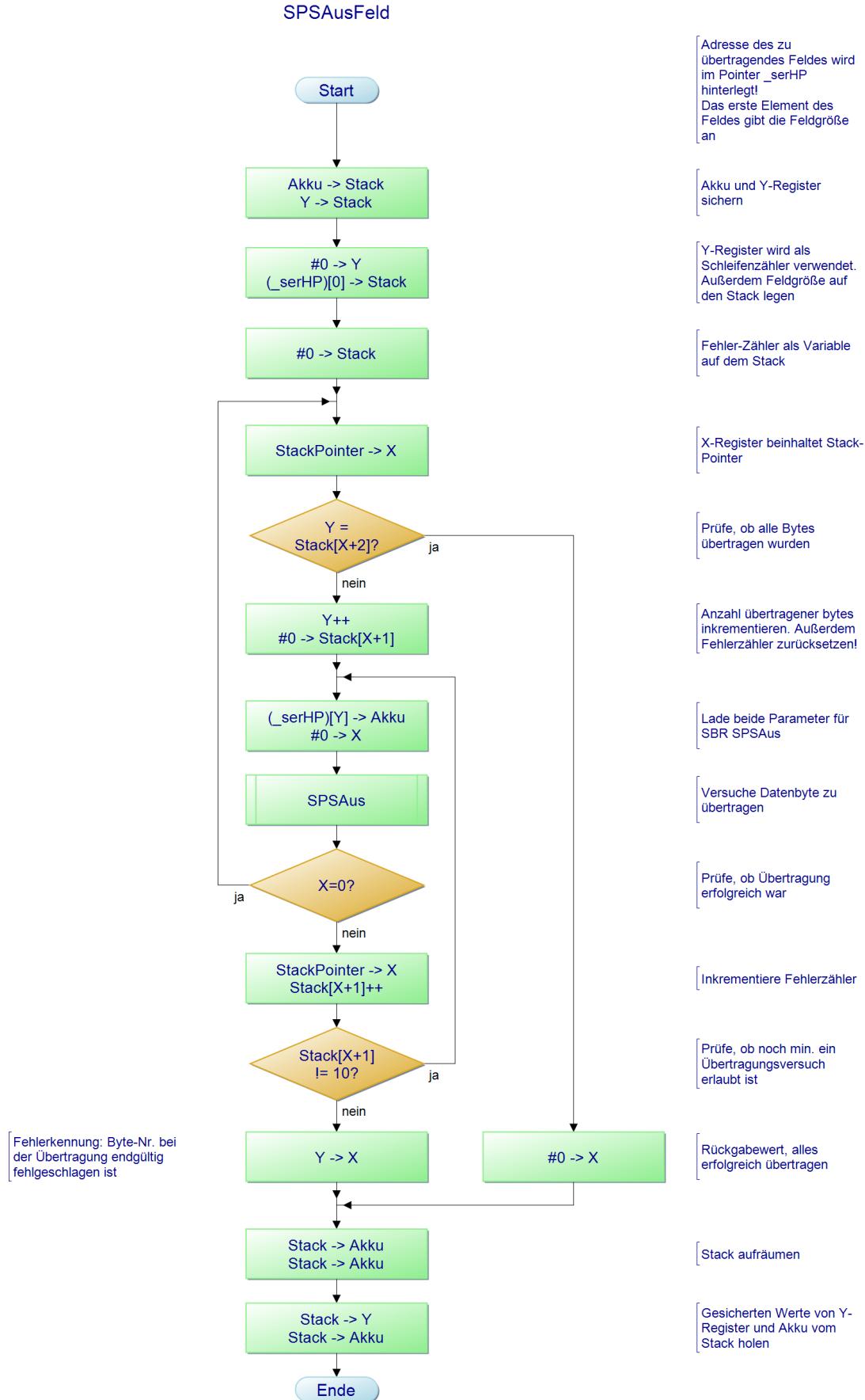


Abbildung 33 PAP SPSAusFeld

Verwendung von Sende-FIFOs

Unter Umständen kann es sinnvoll sein, ein Sende-FIFO zu verwenden, in das Werte eingetragen werden und zu einem gewissen Zeitpunkt zu versenden.

Mit den nachfolgenden Routinen kann ein Sende-FIFO verwendet werden, in das ganz einfach Datenbytes geschrieben und an die SPS übertragen werden können. Sollten mehrere Sende-FIFOs unterschiedlicher Priorität benötigt werden, so kann dieses Programm als Vorlage dienen.

Mit der Subroutine putFIFO kann ein Datenbyte in die Sende-FIFO eingetragen werden. Mit SendNum kann dann eine gewisse Zahl an Bytes aus der FIFO an den Kommunikationspartner übertragen werden.

Folgende nötige Subroutinen bitte den entsprechenden Bibliotheken entnehmen und in den Quelltext einfügen (in den Bereich der Subroutinen).

```
SPSAusR
  _initFIFO
  _deleteFIFO
  _readFIFO
  _readFirstFIFO
  _writeFIFO
  _getFIFOPointer
  _fifoMult
```

Mit nachfolgender Routine wird ein Daten-Byte in den Sende puffer geschrieben.

```
;=====
; Parameter:
;   Akku -> Datenbyte
; Rückgabewerte:
;   X-Register -> 2 (FIFO voll), 0 sonst
;=====

putFIFO:
    LDX #1      ; Sende-FIFO ist Nr. 1
    JSR _writeFIFO

    ; Im X-Register befindet sich bereits der Rückgabewert
    RTS
; -----
```

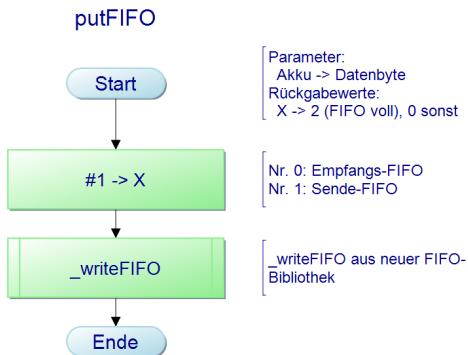


Abbildung 34 PAP putFIFO

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

Die Subroutine sendNum sendet eine gewisse Anzahl an Bytes an den Kommunikations-Partner. Die Werte werden der FIFO entnommen. Ist die FIFO leer, so wird diese Routine vorzeitig beendet. Im Fehlerfall wird jedes Byte max. 10x übertragen. Im endgültigen Fehlerfall bleibt das Datenbyte jedoch in der FIFO erhalten.

```

;=====
; Parameter:
;   Akku -> Max. Anzahl der zu übertragenen Bytes (max. FF)
; Rückgabewerte:
;   X-register:
;     0 -> Alle Datenbytes wurden übertragen (FIFO ist leer)
;     1 -> Max. Anzahl Datenbytes übertragen (FIFO ist nicht leer)
;     2 -> Fehler bei Übertragung eines Datenbytes (10x)
;=====

sendNum:
    PHA ;sichere max. Anzahl der Übertragungen
    TAX

    TYA ;sichere Y-Register auf den Stack
    PHA

    TXA
    TAY ;Anzahl der max. zu übertr. Bytes ins Y-Register
_sendNum1: ;Schleife, zur Übertragung der einzelnen Bytes

    LDX #1 ;Aus dem SendeFIFO lesen (noch nicht Löschen)
    JSR _readFirstFIFO
    CPX #2 ;X-Register abfragen (Prüfe, ob Leer)
    BEQ _sendNumEmpty ;Abbrechen, wenn FIFO leer

    JSR SPSAusR ;Versuche Datenbyte zu übertragen
    CPX #1 ;Frage X-Register ab
    BEQ _sendNumError ;Im Fehlerfall abbrechen

    INX ;X ist jetzt 1
    JSR _readFIFO ;alles erfolgreich, entferne aus FIFO

    DEY
    BNE _sendNum1 ;Breche ab, wenn max. Zahl erreicht

    ;FIFO kann leer sein, wenn im letzten Schleifendurchlauf
    ;das letzte Byte aus der FIFO entfernt wurde
    CPX #2
    BEQ _sendNumEmpty ;Verzweige, wenn FIFO leer ist

    ;FIFO ist noch nicht leer
    LDX #1 ;Rückgabewert
    BNE _sendNumend
  
```

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```
_sendNumEmpty:  
    LDX #0  
    BEQ _sendNumend  
  
_sendNumError:  
    LDX #2  
    ;BNE _sendNumend  
  
_sendNumend:  
    PLA  
    TAY ;Y-Register wiederherstellen  
  
    PLA ;Akku wiederherstellen  
    RTS  
;-----
```

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

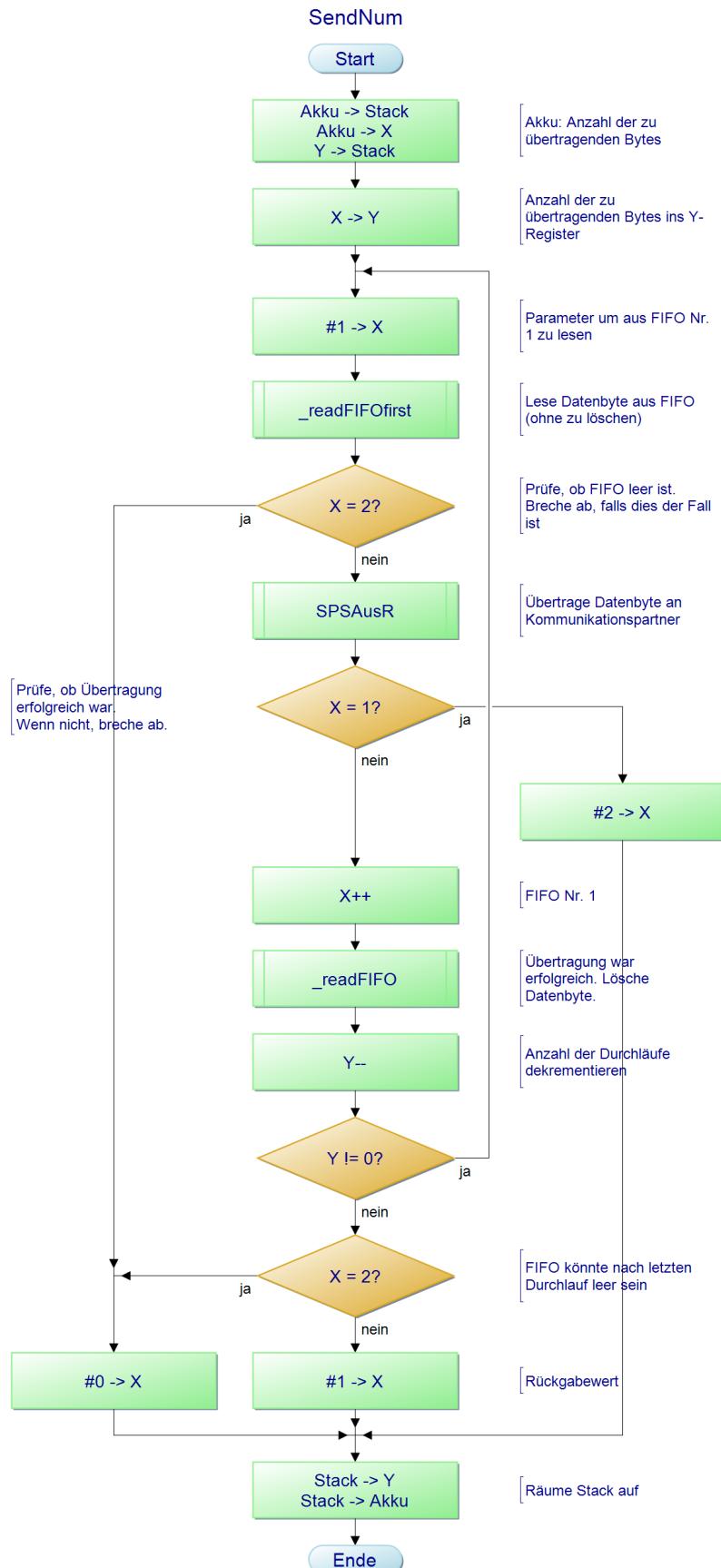


Abbildung 35 PAP SendNum

Komplexere FIFO-Struktur

Für einige der Beispielanwendungen wird diese etwas umfangreichere FIFO-Struktur verwendet. Hiermit kann eine Liste von FIFOs einer bestimmten Größe angelegt werden. So könnten beispielsweise Sende-FIFOs unterschiedlicher Priorität angelegt werden. Verwaltungsoperationen können jeweils auf einzelne FIFOs der Liste angewandt werden. Die einzelnen FIFOs werden zu diesem Zweck durchnummeriert (beginnend mit 0) und können so durch die unterschiedlichen Funktionen angesprochen werden. Das komplette Programm kann der Datei **newFIFO.asm** entnommen werden. Dort ist auch ein kleines Testprogramm mit enthalten.

Im Quelltext und in den PAPs wird oft von IN- bzw. OUT-Pointern gesprochen. Dies ist nicht ganz korrekt, da es sich bei IN und OUT lediglich um Offsets innerhalb einer FIFO handelt. Die Arbeit mit Offsets ist ein wenig einfacher. Der Begriff Pointer (Zeiger) dient lediglich der Veranschaulichung.

Organisation des Speicherbereiches einer Liste von drei FIFOs mit jeweils 6 Elementen:

IN0	OUT0	IN1	OUT1	IN2	OUT2	D0.0	D0.1	D0.2	D0.3	D0.4	D0.5
D1.0	D1.1	D1.2	D1.3	D1.4	D1.5	D2.0	D2.1	D2.2	D2.3	D2.4	D2.5

Wie man erkennen kann, befinden sich die Zeiger vor den eigentlichen FIFOs. Dies muss bei den FIFO-Operationen mit beachtet werden.

Zusammenfassend werden hier kurz alle relevanten Funktionen für die Anwender aufgelistet:

_initFIFO

Initialisiert die FIFO.

Parameter:

Keine

Rückgabewerte:

Keine

_deleteFIFO

Setzt die entsprechende FIFO zurück.

Parameter:

X-Register: FIFO-Nr

Rückgabewerte:

X-Register: Erfolg

_readFIFO

Parameter:

X-Register: Nummer der FIFO

Rückgabewerte:

Akku: Datenbyte

X-Register: Erfolg der Operation

_readFirstFIFO

Unterschied zu _readFIFO: Der gelesene Wert bleibt in der FIFO erhalten

Parameter:

X-Register: Nummer der FIFO

Rückgabewerte:

Akku: Datenbyte

X-Register: Erfolg der Operation

_writeFIFO

Parameter:

X-Register: Nummer der FIFO

Akku: Datenbyte

Rückgabewert:

X-Register: Erfolg der Operation

Quelltext

Zu Beginn des Quelltextes sind einige Konstanten festgelegt:

```
; Pointer für interne Zwecke benötigt
_p = $80

; Anzahl der FIFOs, max. 127
_fifoNum = $8

; Größe pro FIFO (in Bytes)
_fifoSize = $10

; Start-Adresse der kompletten Datenstruktur
_fifoAddr = $4500
```

Bei Bedarf können diese Werte angepasst werden. Es wird ein Zeiger `_p` benötigt, dessen Funktion aber für den Anwender nicht relevant ist, da er lediglich für interne Zwecke benötigt wird.

Zu Beginn des Programmes muss die FIFO-Liste initialisiert werden. Dies geschieht mit dem Funktionsaufruf `_initFIFO`.

```

;=====
; Parameter:
;   Keine
; Rückgabewerte:
;   Keine
;=====

_initFIFO:
    SEI    ;IRQs verbieten
    PHA    ;sichere Akku auf den Stack

    ;sichere das X-Register auf den Stack
    TXA
    PHA

    ;Setze alle In-/Out-Zeiger zurück (Null)
    ;Ermittle zunächst Anzahl der Schleifendurchläufe
    LDA #_fifoNum
    ASL
    TAX
    LDA #0

    ;Setze alle In-/Out Pointer in der Schleife zurück
_initFIFOsetP:
    STA _fifoAddr-1, X
    DEX
    BNE _initFIFOsetP

    ;Alten Wert des X-Registers wieder laden
    PLA
    TAX

    ;Alten Akku-Wert wieder initialisieren
    PLA

    CLI    ;Erlaube IRQs wieder
    RTS
;-----

```

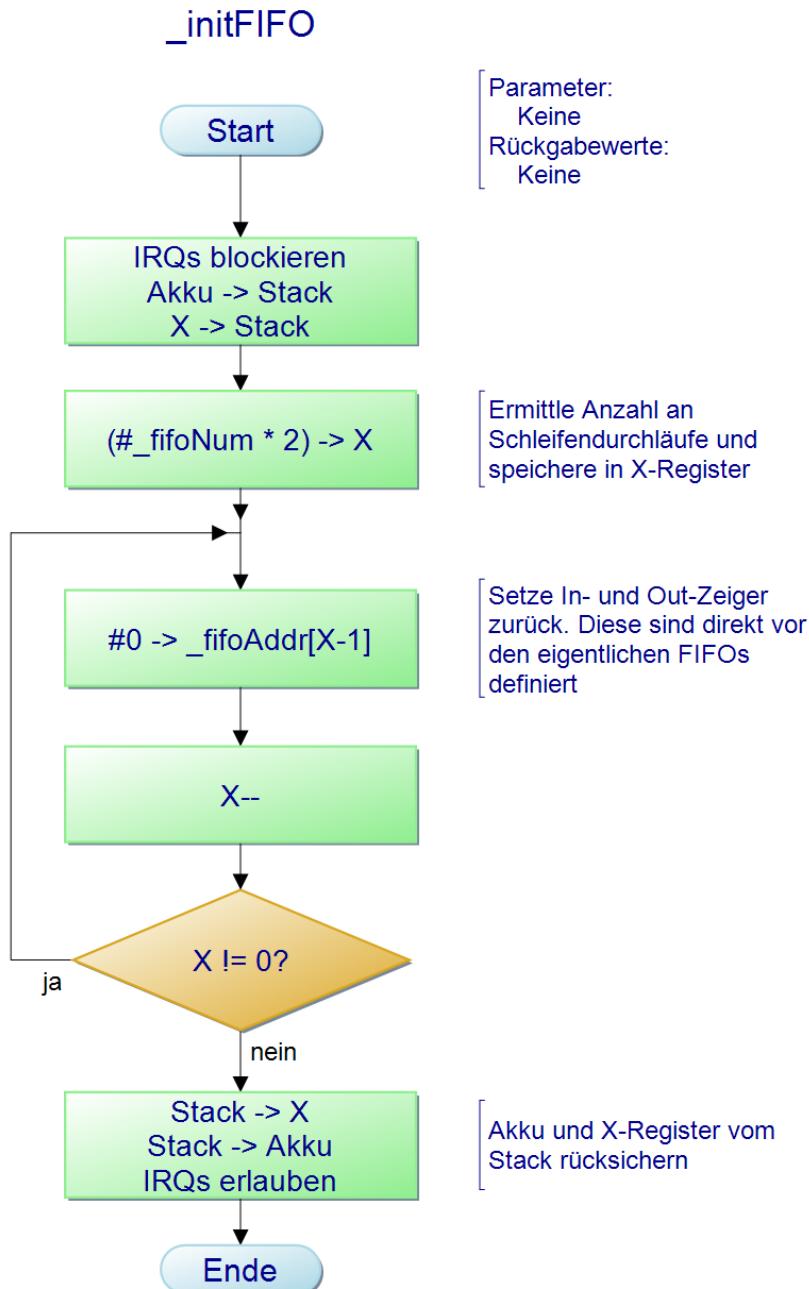


Abbildung 36 PAP InitFIFO MOSES

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

Soll eine FIFO zurückgesetzt werden, so wird die Subroutine _deleteFIFO verwendet. Als Parameter wird die FIFO-Nr. angegeben.

```

;=====
; Parameter:
;   X-Register -> FIFO_Nr
; Rückgabewerte:
;   X-Register -> Erfolg
;=====

_deleteFIFO:
    PHA ;sichere Akku

    ;Prüfe Nummer: Diese darf nicht >= _fifoNum sein
    CPX #_fifoNum
    BCS _deleteFIFOWrongNum

    TXA
    ASL ;Bestimme Offset für In-Pointer
    TAX

    LDA #0
    STA _fifoAddr, X
    STA _fifoAddr+1, X

    TAX ;Rückgabewert: 0
    BEQ _deleteFIFOend

_deleteFIFOWrongNum:
    LDX #1 ;Fehler, FIFO_Nr nicht vorhanden

_deleteFIFOend:
    PHA ;Rücksichern des Akkus
    RTS
;-----

```

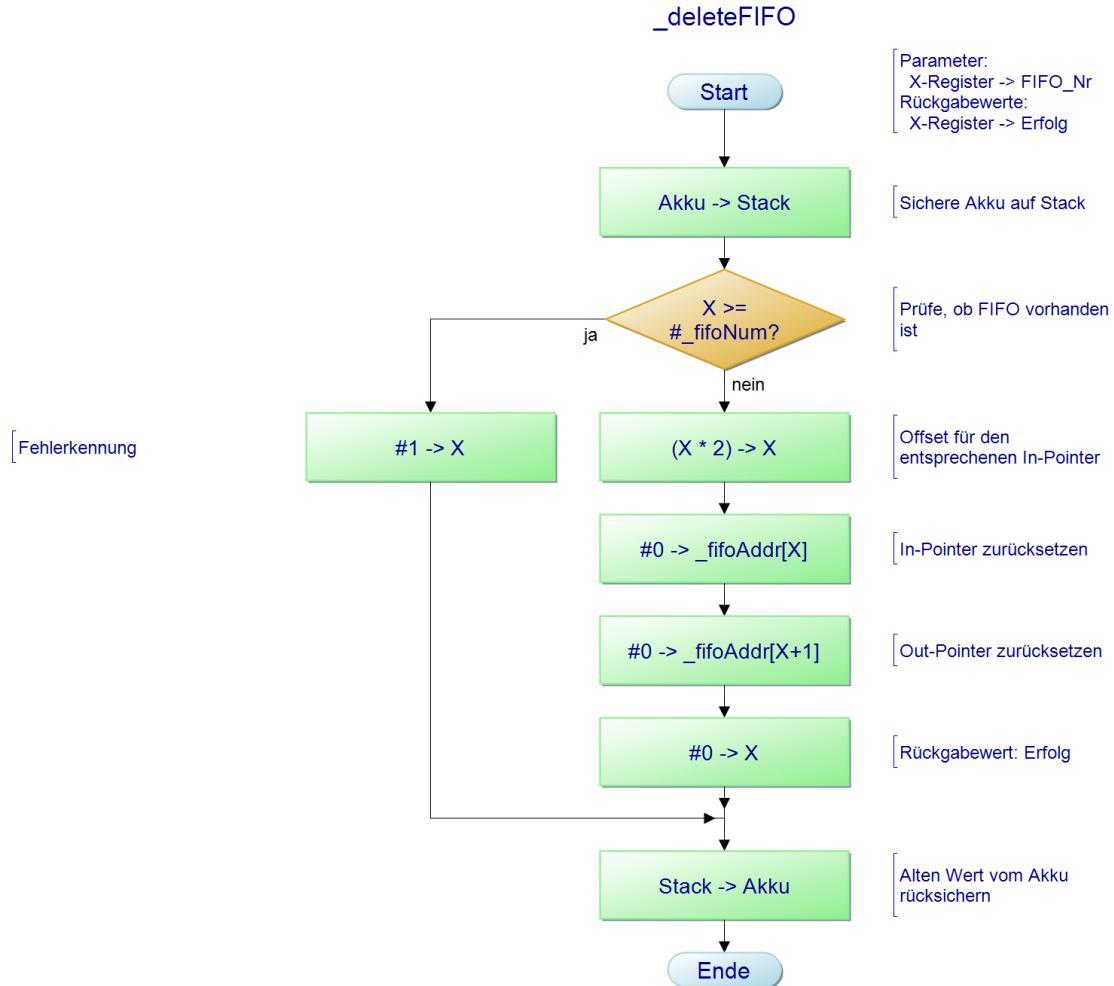


Abbildung 37 PAP DeleteFIFO MOSES

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

Aus einer FIFO wird mit _readFIFO gelesen. Auch hier wird die Nummer der FIFO übergeben.

```

;=====
; Parameter:
;   X-Register: Nummer der FIFO
; Rückgabewerte:
;   Akku: Datenbyte
;   X-Register: Erfolg der Operation
; TODO: Verbiete IRQs in kritischen Bereichen
;=====

_readFIFO:
    TYA
    PHA ;sichere Y-Register auf den Stack

    ;Prüfe Nummer: Diese darf nicht >= _fifoNum sein
    CPX #_fifoNum
    BCS _readFIFOWrongNum

    TXA
    ASL ;Bestimme Offset für In-Pointer
    ;FIFO leer, Wenn IN = OUT
    TAY
    LDA _fifoAddr, Y
    ;INY
    CMP _fifoAddr + 1, Y
    BEQ _readFIFOempty ;Verzweige, wenn FIFO leer

    LDA _fifoAddr + 1, Y ;Lade OUT-Pointer
    TAY

    ;Berechne:
    ; Basis-Adresse      ->Beginn des Datenbereiches
    ; + FIFO_Nr * _fifoSize  ->Pointer auf Start der FIFO
    ; + _fifoNum * 2->IN-/OUT P. befinden sich vor FIFOs

    TXA ;Nr. der FIFO
    JSR _getFIFOPointer

    LDA (_p), Y ;Lese Wert aus der FIFO
    PHA ;zunächst auf den Stack
    LDA #0
    STA (_p), Y ;Lösche übersichtshalber den Wert (DEBUG)

    INY ;Inkrementiere den OUT-Pointer
    CPY #_fifoSize ;Prüfe, ob Ende der FIFO erreicht
    BNE _readFIFOSuccess
    LDY #0 ;Zeiger wird wieder auf den Start gesetzt

_readFIFOSuccess:
    TXA ;Nr. der FIFO steht noch im X-Register

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

ASL    ;Offset für den In-Pointer
TAX
INX    ;Offset für den Out-Pointer

TYA    ;OUT-Pointer nach Akku
STA _fifoAddr, X

PLA    ;Datenbyte vom Stack

LDX #0      ;Rückgabewert
BEQ _readFIFOend

_readFIFOWrongNum:
LDX #1
BNE _readFIFOend

_readFIFOempty:
LDX #2
_readFIFOend:
STA _p      ;Nutzte _p als temporäre Variable

PLA    ;Hole Y-Register vom Stack
TAY

LDA _p
RTS
;-----

```

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

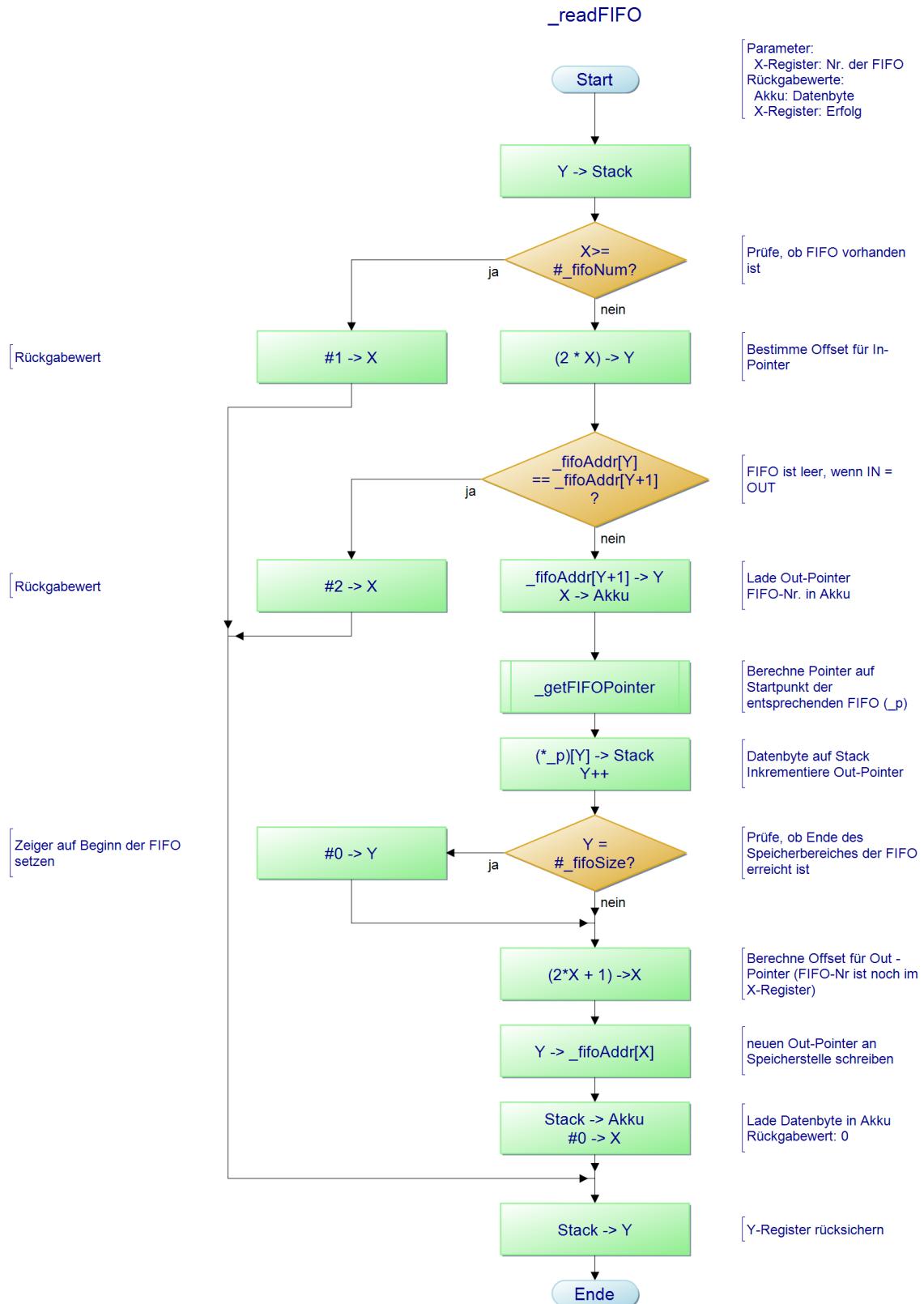


Abbildung 38 PAP ReadFIFO MOSES

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

Mit dieser weiteren Methode kann auch aus einer FIFO der Liste gelesen werden, mit dem Unterschied, dass der gelesene Wert nicht aus der FIFO gelöscht wird.

```

;=====
;Unterschied zu _readFIFO:
;    Der gelesene Wert bleibt in der FIFO erhalten
;Parameter:
;    X-Register: Nummer der FIFO
;Rückgabewerte:
;    Akku: Datenbyte
;    X-Register: Erfolg der Operation
;TODO: Verbiete IRQs in kritischen Bereichen
=====

_readFirstFIFO:
    ;Prüfe Nummer: Diese darf nicht >= _fifoNum sein
    CPX # _fifoNum
    BCS _readFirstFIFOWrongNum

    TYA
    PHA ;sichere Y-Register auf den Stack

    TXA
    ASL ;Bestimme Offset für In-Pointer

    ;FIFO leer, Wenn IN = OUT
    TAY
    LDA _fifoAddr, Y
    ;INY
    CMP _fifoAddr + 1, Y
    BEQ _readFirstFIFOempty ;Verzweige, wenn FIFO leer

    LDA _fifoAddr + 1, Y ;Lade OUT-Pointer
    TAY

    ;Berechne:
    ;    Basis-Adresse      ->Beginn des Datenbereiches
    ;    + FIFO_Nr * _fifoSize ->Pointer auf Start der FIFO
    ;    + _fifoNum * 2->IN-/OUT P. befinden sich vor FIFOs
    TXA ;Nr. der FIFO
    JSR _getFIFOPointer

    LDA (_p), Y ;Lese Wert aus der FIFO
    TAX ;zunächst nach X-Register

    PLA
    TAY ;Rücksichern des Y-Registers

    TXA ;Datenbyte

    LDX #0 ;Rückagebwert
  
```

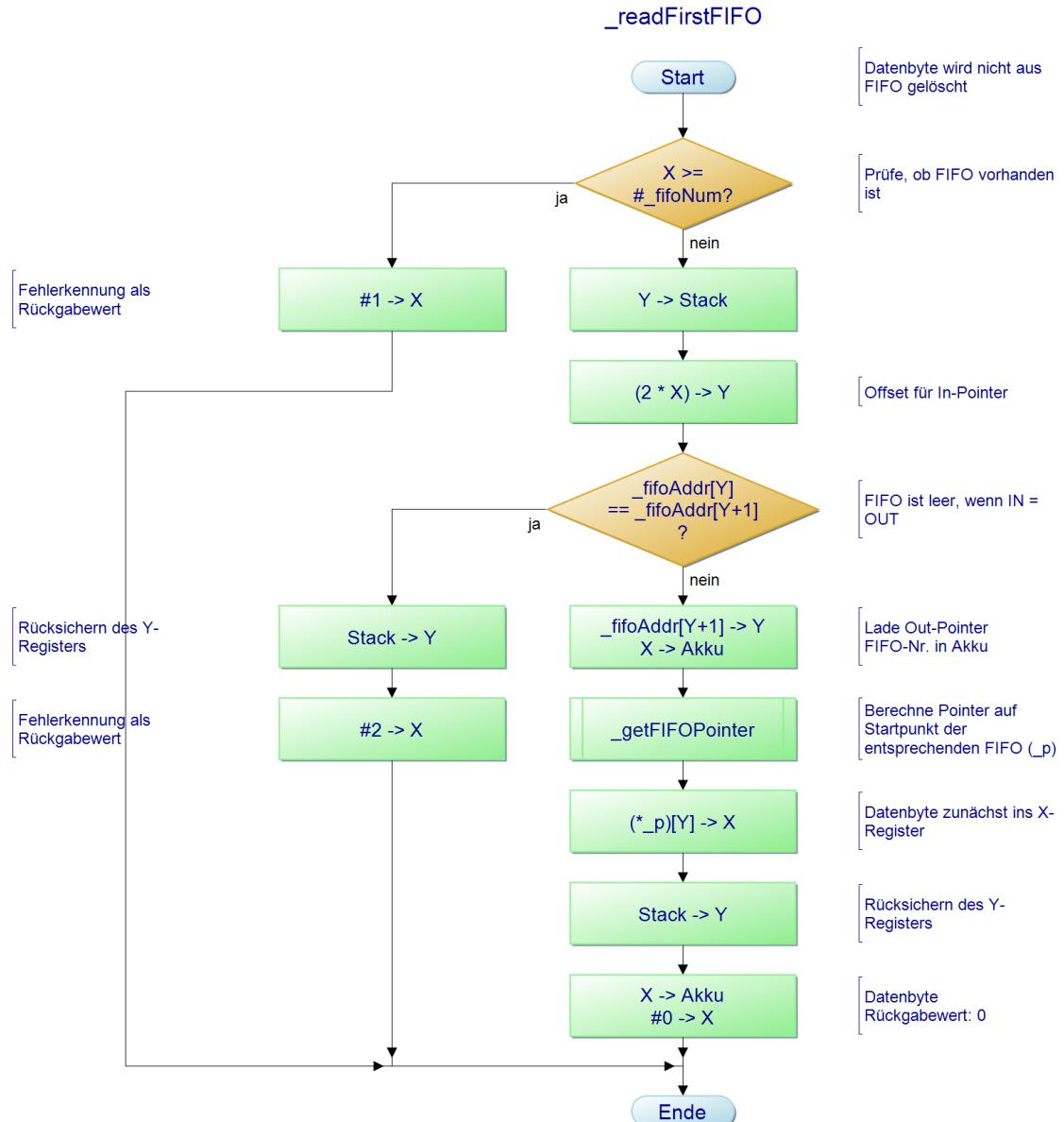
Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```
BEQ _readFirstFIFOend

_readFirstFIFOWrongNum:
    LDX #1
    BNE _readFirstFIFOend

_readFirstFIFOempty:
    PLA
    TAY          ;Rücksichern des Y-Registers
    LDX #2

_readFirstFIFOend:
    RTS
;-----
```



Schreiben in eine FIFO wird mithilfe der Routine `_writeFIFO` realisiert:

```

;=====
; Parameter:
;   X-Register: Nummer der FIFO
;   Akku: Datenbyte
;Rückgabewert:
;   X-Register: Erfolg der Operation
;TODO: Verbiete IRQs in kritischen Bereichen
;=====

_writeFIFO:
    PHA ;sichere Datenbyte zunächst auf den Stack

    TYA
    PHA ;sichere Y-Register auf den Stack

    TXA
    PHA ;sichere X-Register auf den Stack

    ;Prüfe Nummer: Diese darf nicht >= _fifoNum sein
    ;SEC
    ;TXA
    CMP #_fifoNum
    BCS _writeFIFOWrongNum

    ;Prüfe ob FIFO voll: Kann auch ohne Berechnung der
    ;entsprechenden FIFO-BasisAdresse erfolgen
    ;Voll = IN+1==OUT v (IN+1==SIZE & OUT==0)

    ;TXA ;Offset für IN-Pointer berechnen
    ASL

    ;Prüfe, ob IN+1==OUT
    TAY
    LDA _fifoAddr, Y ;Lade IN-Pointer
    SEC
    ADC #0
    ;INY
    CMP _fifoAddr + 1, Y ;Vergleiche mit OUT-Pointer
    BEQ _writeFIFOFull

    ;Prüfe, ob IN+1==SIZE & OUT==0
    ;IN+1 steht noch im Akku
    CMP #_fifoSize
    BNE _writeFIFO1 ;Verzweige, wenn FIFO nicht voll

    ;Prüfe, ob OUT==0
    LDA _fifoAddr + 1, Y
    BEQ _writeFIFOFull

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

_writeFIFO1:
;FIFO ist nicht voll
TXA ;Nr. der FIFO befindet sich noch in X

;Berechne:
; Basis-Adresse      ->Beginn des Datenbereiches
; + FIFO_Nr * _fifoSize ->Pointer auf Start der FIFO
; + _fifoNum * 2->IN-/OUT P. befinden sich vor FIFOs
JSR _getFIFOPointer

; + IN-Offset bestimmen      ->Offset innerhalb der FIFO
;DEY ;Y zeigt noch auf Out-Pointer

TYA ;Sichere Offset für IN-Pointer
PHA

LDA _fifoAddr, Y
TAY ;Offset innerhalb der FIFO -> IN-Pointer

;Schreibe Datenbyte in die FIFO
TSX
LDA $0104, X ;Hole Datenbyte vom Stack
STA (_p), Y ;Schreibe in die FIFO

;Inkrementiere IN-Pointer
INY
CPY #_fifoSize ;Prüfe, ob Ende der FIFO erreicht
BNE _writeFIFOSuccess
LDY #0 ;Zeiger wird wieder auf den Start gesetzt

_writeFIFOSuccess:
PLA ;Offset für IN-Pointer
TAX
TYA
STA _fifoAddr, X;Speichere neuen IN-Pointer

LDX #0
BEQ _writeFIFOend

_writeFIFOWrongNum:
LDX #1
BNE _writeFIFOend

_writeFIFOFull:
LDX #2
;BNE _writeFIFOend

_writeFIFOend:
PLA ;Hole FIFO_Nr vom Stack

```

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```
PLA    ; Hole Y-Register vom Stack
TAY
PLA    ; Hole Datenbyte vom Stack
RTS
; -----
```

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

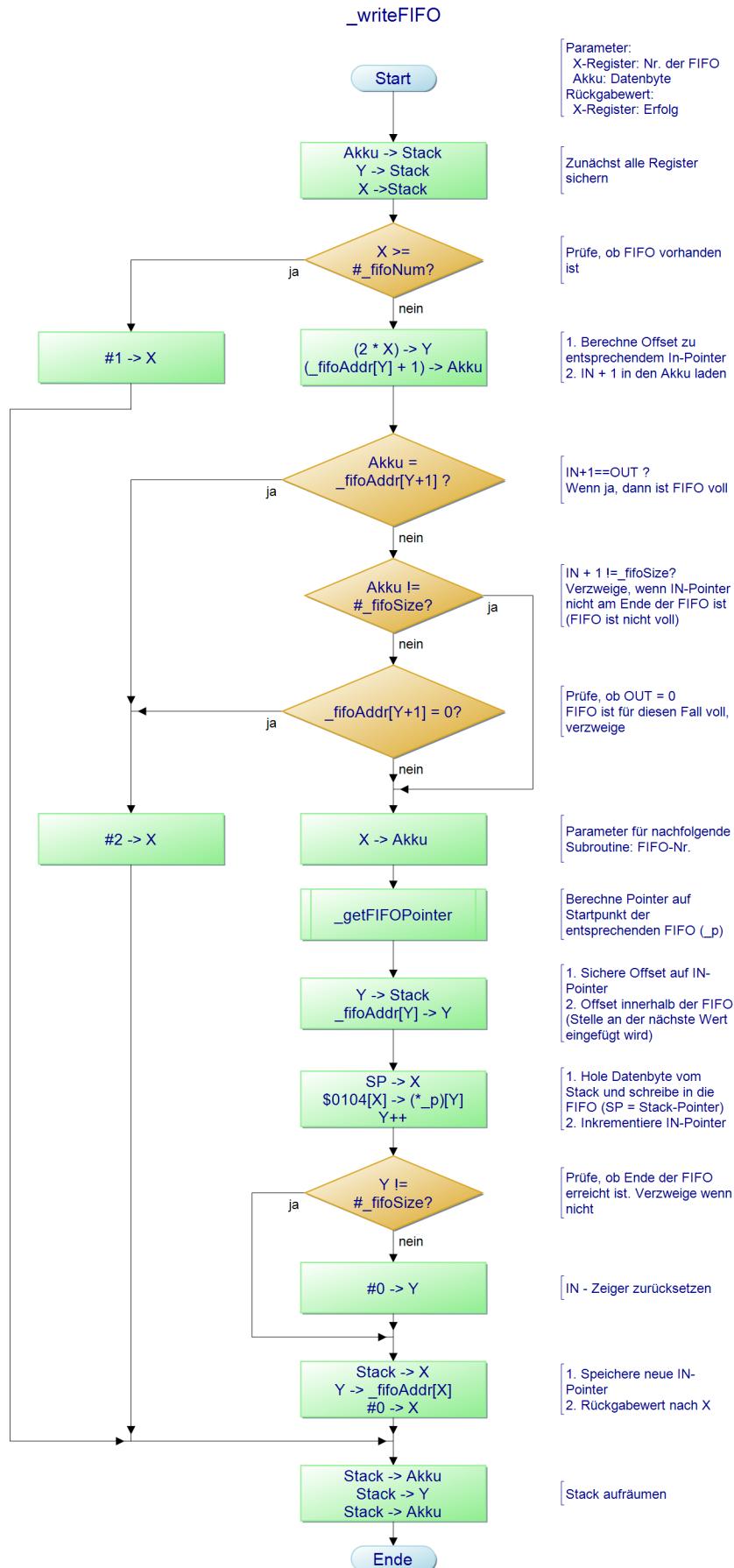


Abbildung 39 PAP writeFIFO

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

Die beiden letzten Routinen sind nicht relevant für den Anwender. Vollständigkeitshalber werden sie hier dennoch aufgeführt.

```

;=====
;Berechnet einen Zeiger auf den Startpunkt einer FIFO!
;Akku wird nicht verändert!
;
;Parameter:
;    Akku -> Nr. der FIFO
;
;Ergebnis:
;    Der berechnete Pointer wird in _p abgelegt
;=====

_getFIFOPointer:
    PHA ;sichere FIFO_Nr auf den Stack

    ;Berechne Grundadresse der angesprochenen FIFO
    ;Beide Operanden der Multiplikation werden direkt
    ;über den Pointer übergeben

    STA _p
    LDA #_fifoSize ;Größe der FIFO
    STA _p + 1
    JSR _fifoMult ;Berechne: Größe_der_FIFO * FIFO_NR

    ;Addiere Anzahl der belegten Bytes aller Pointer auf die
    ;Adresse, da die sich vor allen FIFOs befinden
    LDA #_fifoNum ;Anzahl an belegten Bytes für die Pointer
    ASL

    CLC
    ADC _p
    STA _p

    LDA _p+1
    ADC #0
    STA _p+1

    ;Addiere Basis-Adresse
    CLC
    LDA #<_fifoData
    ADC _p
    STA _p

    LDA #>_fifoData
    ADC _p + 1
    STA _p + 1

    PLA ;Hole FIFO_Nr wieder vom Stack
  
```

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```
; Im Pointer _p befindet sich jetzt der Zeiger auf das
; ausgewählte Element der entsprechenden FIFO
RTS
; -----
```

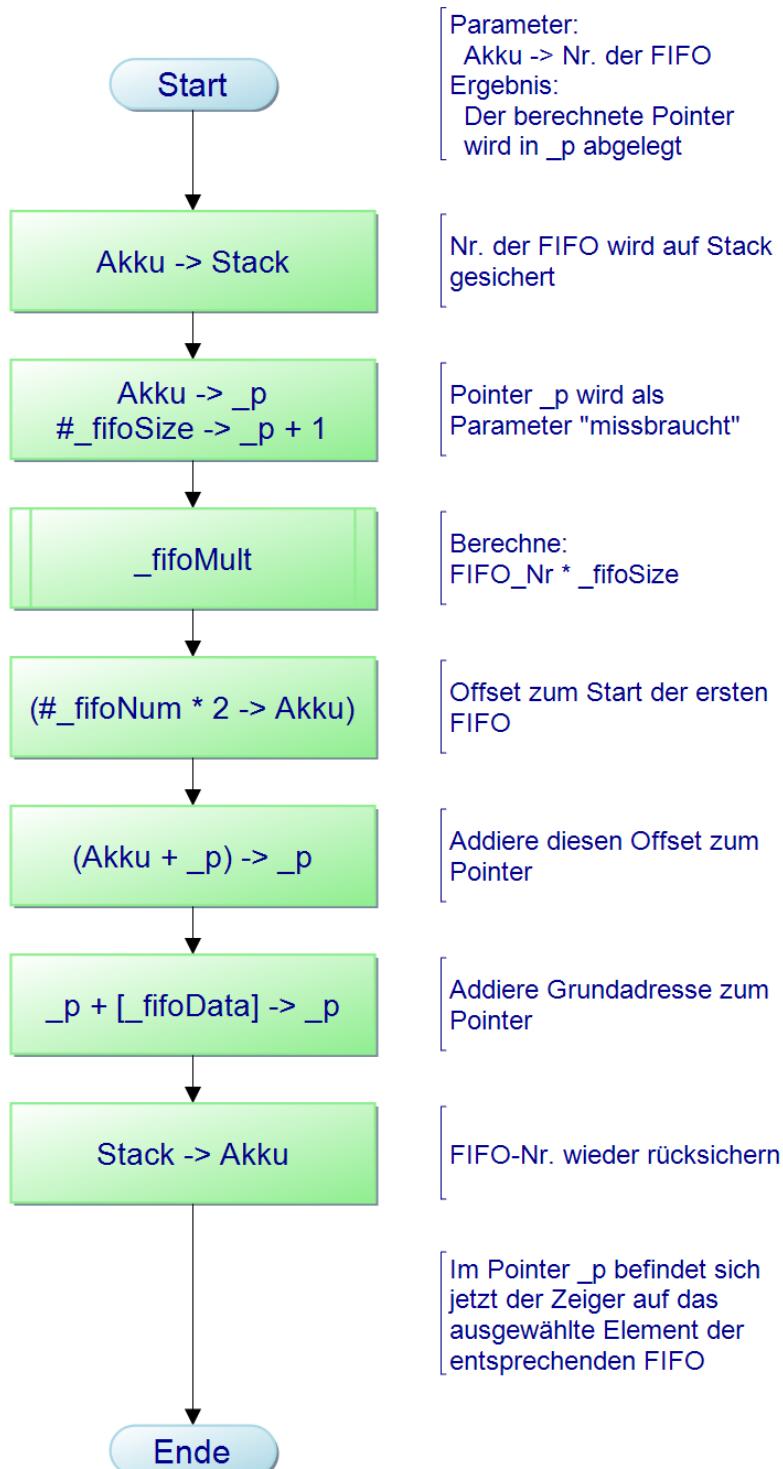
_getFIFOPointer

Abbildung 40 PAP getFIFOPointer

```

;=====
;Berechnet die Grundadresse der ausgewählten FIFO
;Parameter werden im Pointer abgelegt:
;    _p      : FIFO_Nr
;    _p + 1    :_fifoSize
;Ergebnis in kompletten Pointer _p:
;    FIFO_Nr * _fifoSize
=====

_fifoMult:
    TXA    ;sichere X-Register
    PHA

    LDA #0
    LDX #8
    LSR _p          ;0 -> MSB
_fifoMult1:
    BCC _fifoMult2
    CLC
    ADC _p + 1
_fifoMult2:
    ROR      ;darf nicht LSR sein
    ROR _p
    DEX
    BNE _fifoMult1

    STA _p + 1

    PLA
    TAX

    RTS
;-----

```

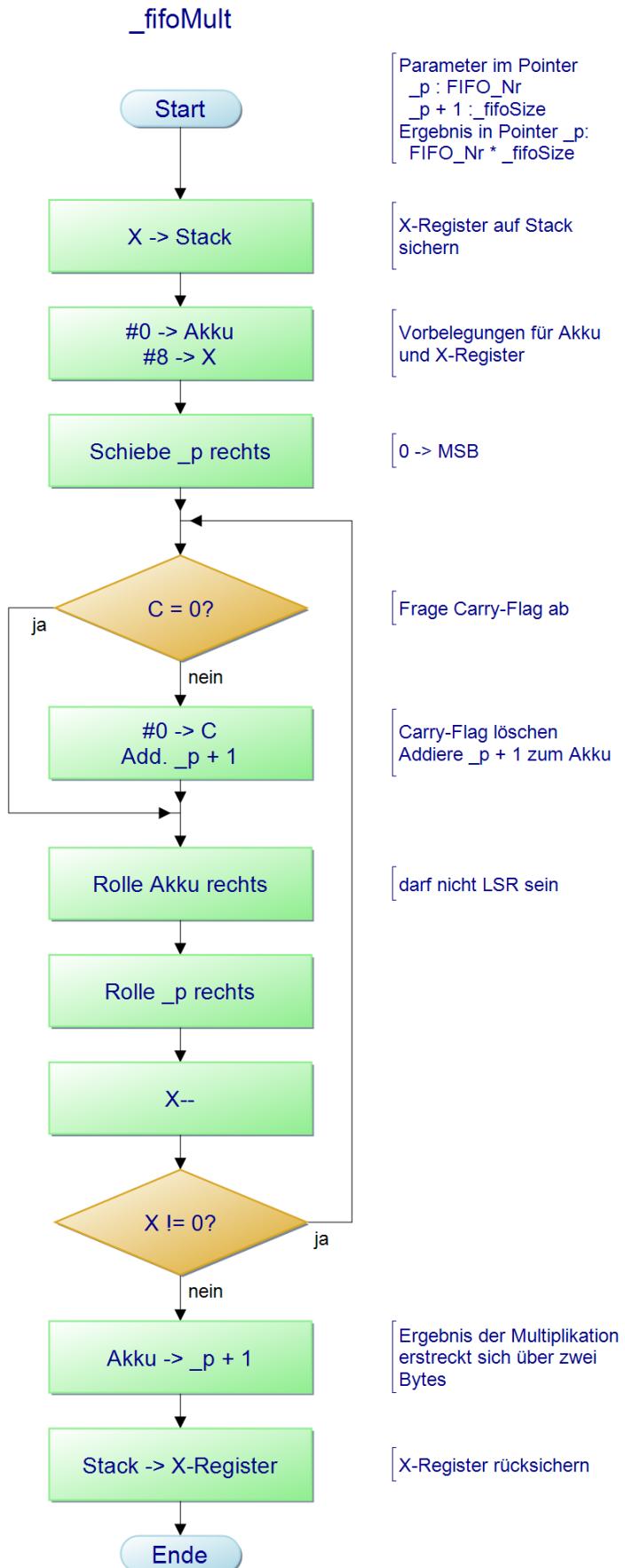


Abbildung 41 PAP fifoMult

Auswertung von bool'schen Funktionen mithilfe der SPS

Die Implementierung von boolschen Funktionen ist an der SPS deutlich einfacher als beim MOSES.

Hier knüpft dieses Anwendungsbeispiel an.

An einer Portbox des MOSES können bis zu acht Variablen (8 Bits) eingelesen werden, diese werden an die SPS übermittelt und die die entsprechende boolsche Funktion ausgewertet. Das Ergebnis dieser Operation wird anschließend wieder zurück an den MOSES übertragen. Die Funktion kann bis zu acht Variablen enthalten {f(A,B,C,D,E,F,G,H)}.

In diesem Beispiel prüft die Funktion, ob eine 8-Bit-Zahl eine Primzahl ist. In der Regel würde dies algorithmisch gelöst werden. Hier wird dies jedoch mithilfe der boolschen Algebra realisiert, um eine etwas komplexere Funktion zu erhalten, die jedoch ohne Probleme auf der SPS implementiert werden kann.

Das MOSES-Programm wurde vollständig fertig gestellt, aus Zeitgründen konnten jedoch nur Teile für die SPS realisiert werden.

Das Programm verwendet den Port A als Eingang, hier sollten also die Schalter zur Einstellung der Variablenkombination angeschlossen werden. Das Ergebnis der Funktion wird in diesem Beispiel auf dem Port B ausgegeben.

```
.LIB "C:\SHELL\arbeit\KONST.lib"
.Lib "C:\SHELL\arbeit\PIAT.lib"

.ORG $4000
LDA $02
JSR SPSIni

;Alle Pins von Port B als Ausgang
LDA CRB
AND #11111011 ;Zugriff auf DDRB
STA CRB
LDX #$FF ;Alle Pins als Ausgänge
STX DDRB
ORA #00000100 ;Jetzt wieder Zugriff auf DRB
STA CRB

;Port A als normalen Eingang verwenden
;Port A sollte verwendet werden, damit nicht zufälliger-
;weise IRQs an PortB währenden Senden/Empfang quittiert
;werden
LDA #00000100
STA CRA
;DDRA sollte bereits korrekt beschrieben sein

waitDig1:
  LDA #0
  JSR DigAus
  JSR DigEin
  STA tmp ;Schalterstellung zwischenspeichern
  AND #$04
  BEQ waitDig_con
```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

JMP progEnd ;Mit Schalter 3 abbrechen

waitDig_con:
  LDA tmp ;Schalterstellung wieder laden
  AND #$08
  ;solange warten wie Schalter 4 nicht betätigt
  BEQ waitDig1

  ;Lese Schalterstellung an PortA ein
  LDA DRA
  STA tetr2 ;zunächst zwischenspeichern
  ;LSBs wegschreiben
  AND #$0F
  STA tetr1

  ;Ermittle MSBs
  LDA tetr2
  AND #$F0
  LSR
  LSR
  LSR
  LSR
  STA tetr2

  ;Auszuwertende Funktion ermitteln
  ;hinteren beiden Schalter
  JSR DigEin
  AND #$03
  ASL
  ASL
  ASL
  ASL
  STA tmp ;zwischenspeichern

  ;Zum ersten Übertragungsbyte hinzufügen
  ORA tetr1
  STA tetr1

  ;Zum zweiten Übertragungsbyte hinzufügen
  LDA tmp
  ORA tetr2
  STA tetr2
  ORA #$40 ;Vordere Tetrade

  LDY #0
sndLoop:
  LDA tetr1, Y
  JSR SPSAusR
  CPX #0

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

BNE progEnd ; Im Fehlerfall Programm beenden

INY
CPY #2
BNE sndLoop

; Warte auf Antwort der SPS
waitRecv:
JSR _countFIFO
CMP #2
BNE waitRecv

; Zwei Werte in der FIFO lesen
JSR readFIFO
STA tetr1
JSR readFIFO
STA tetr2

; Prüfe, ob die Funktionsnummern übereinstimmen
LDA tetr1
AND #%00110000
STA tmp

LDA tetr2
AND #%00110000
CMP tmp
; Programm Abbrechen, wenn Nummern nicht übereinstimmen
BNE progEnd

; Verarbeite Tetrade 1
LDA tetr1
AND#$40 ; Prüfe, ob vordere Tetrade
BNE tetr1_front

;tetr1 ist hintere Tetrade
LDA tetr1
AND #$0F
JMP test_tetr2

tetr1_front:
;tetr1 ist vordere Tetrade
LDA tetr1
ASL
ASL
ASL
ASL

test_tetr2:
STA tmp ; Tetrade 1 zwischenspeichern
  
```

```

;Verarbeite Tetrade 2
LDA tetr2
AND #$40 ;Prüfe, ob vordere Tetrade
BNE tetr2_front

;tetr2 ist hintere Tetrade
LDA tetr2
AND #$0F
JMP put_together

tetr2_front:
;tetr2 ist vordere Tetrade
LDA tetr2
ASL
ASL
ASL
ASL

put_together:
;Setze beide Tedraden zusammen
ORA tmp
STA DRB ;Ergebnis auf Port B schreiben

;Prüfe, ob Programm weiter ausgeführt wird
JSR DigEin
AND #$04
BEQ JMPwaitDig1

progEnd:
;Programmende
LDA #$0F
JSR DigAus
RTS

JMPwaitDig1:
JMP waitDig1

.ORG $4500
tetr1: .BYTE $00
tetr2: .BYTE $00
tmp: .BYTE

```

Die Funktion wurde zunächst mithilfe einer Funktionstabelle aufgestellt und anschließend mithilfe des Quine-McClusky Verfahrens minimiert und letztendlich durch Faktorisierung in die untere Form gebracht.

Aus Übersichtlichkeitsgründen wurde die Funktion in drei Abschnitte unterteilt und in FUP auf der SPS realisiert.

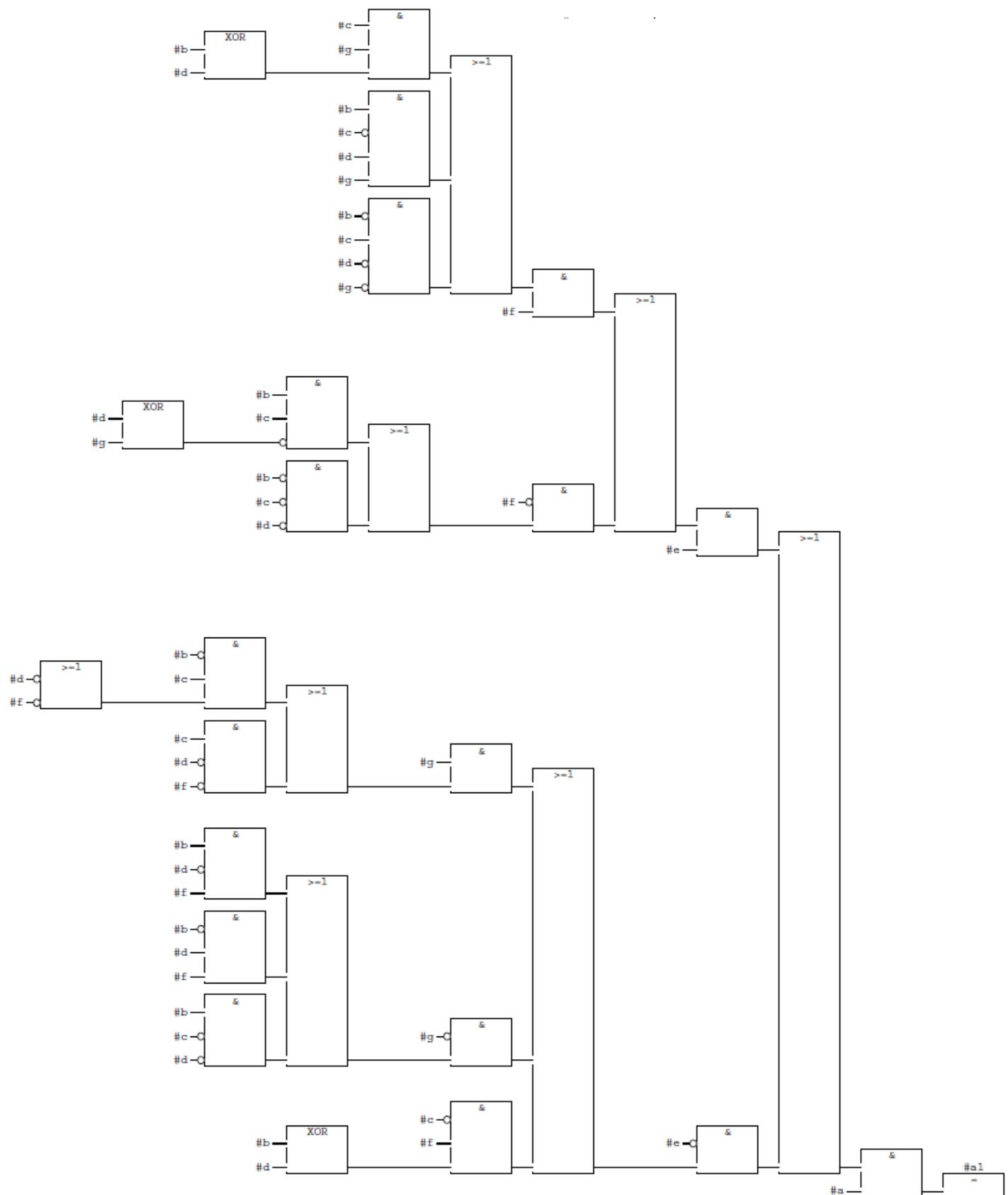


Abbildung 42 Prim-Funktion (1)

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

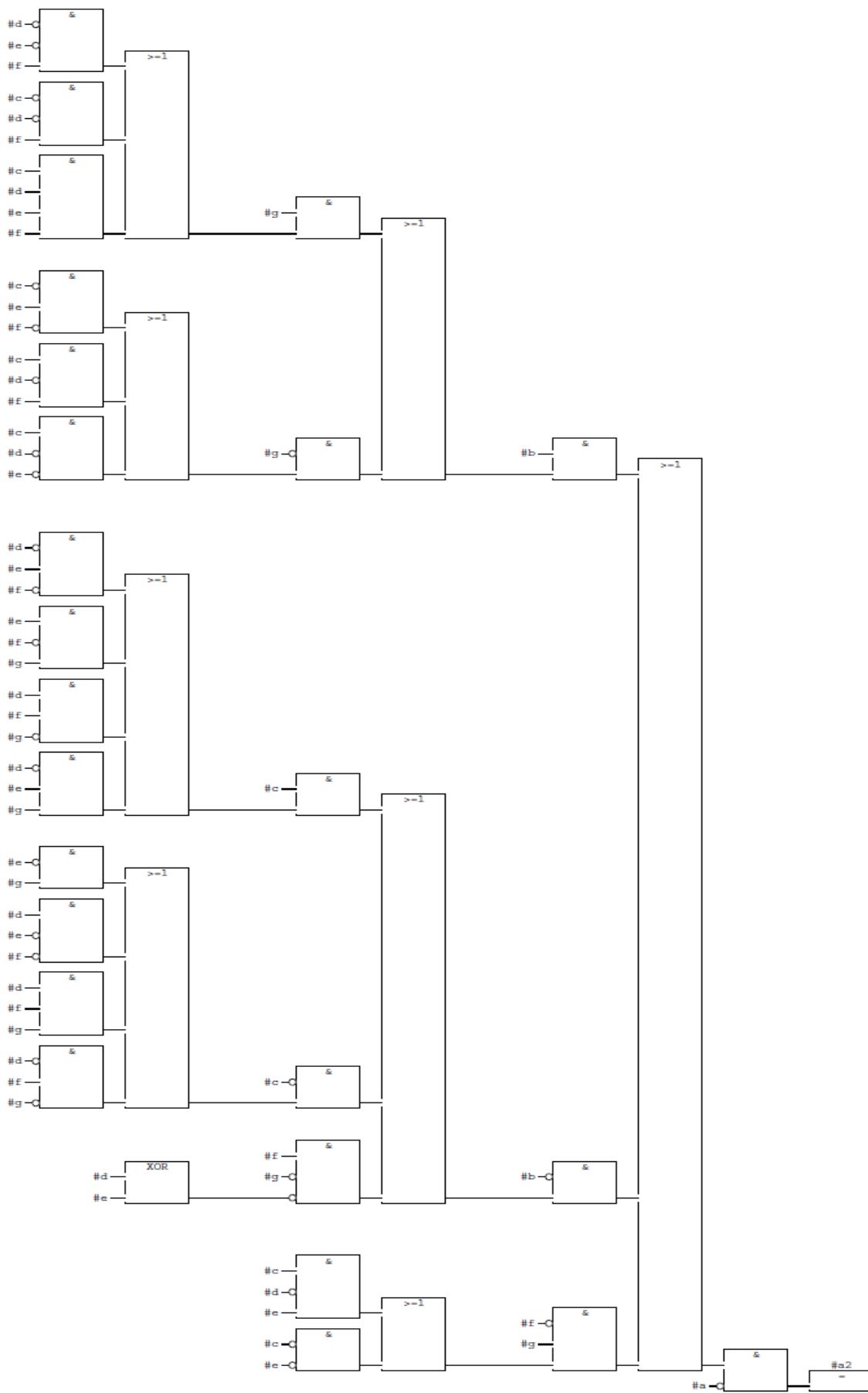


Abbildung 43 Prim-Funktion (2)

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

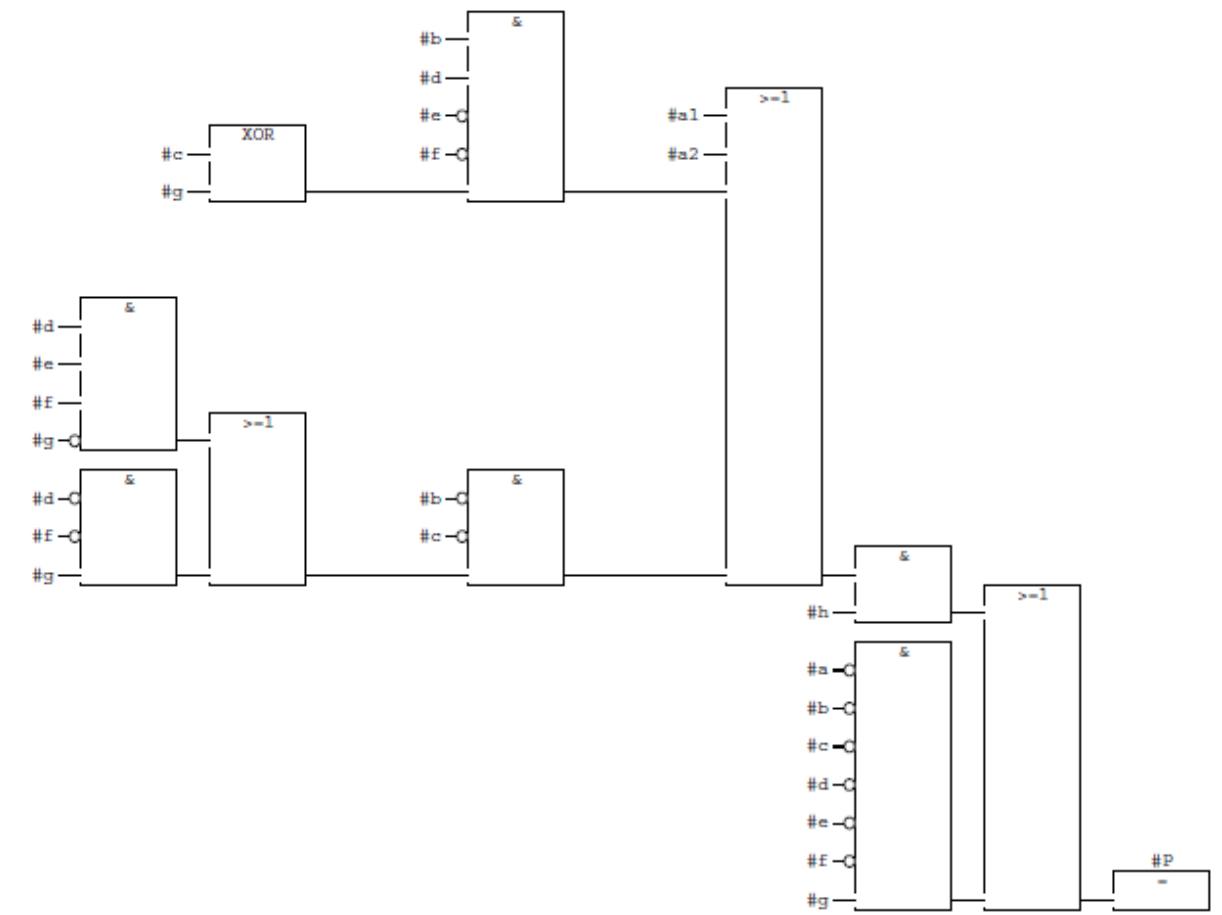


Abbildung 44 Prim-Funktion (3)

MOSES - Programm

Quelltext

```

;=====
;Start-Adresse der Subroutinen und IRQ-Behandlungs-Routine
progAddr = $3900
;=====

;=====
;Daten der FIFO
_fifoSize = 255      ;Max. 255
_fifoAddr = progAddr + $500    ;Startadr der FIFO und
Verwaltungsinformationen
;=====

;=====
;Timer-Variablen
;-----
_TimerLL = $88
_TimerUL = $13

;Halbe Zeit, für erstes Intervall beim Empfang
;Es ist Sinnvoller ohne Toggle-Flag zu arbeiten, da ansonsten
;beim Senden immer und beim Empfang fast immer unnötige Timer-IRQs
;erzeugt werden...
_TimerLLHalf = $C4
_TimerULHalf = $09
;=====

;=====
;Masken für das CRB-Register
;-----
_serCRB1Edge      = %00000010
_serCB2Set        = %00001000
_serCB2Del        = %11110111
;=====


```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

;=====
; Masken für die einzelnen Flags von _serFlags
;-----

_serSetCB1 =      %00000001 ;Bit 0 -> Status Flag der Leitung CB1
_serDelCB1 =      %11111110
_serRcvEn =       %00000010 ;Bit 1 -> Empfangsmodus Freigabe
_serRcvDis =       %11111101 ;Bit 1 -> Empfangsmodus deaktivieren
_serRcvAct =       %00000100 ;Bit 2 -> Empfangsmodus aktiv
_serRcvNAct =      %11111011 ;Bit 2 -> Empfangsmodus deaktivieren
_serSndAct =       %00001000 ;Bit 3 -> Sendemodus aktiv
_serSndNAct =      %11110111 ;Bit 3 -> Sendemodus deaktivieren
_serStartbit =     %00010000 ;Bit 4 -> StartBit
_serStopbit =      %00100000 ;Bit 5 -> StopBit
_serParity =       %01000000 ;Bit 6 -> Parität
_serErrFlg =        %10000000 ;Bit 7 -> Fehlerkennung (1 -> Fehler)

;=====

; Masken für _serSpecFlags
_serSpecLine = %00000001 ;Bit 0 -> Verwende Leitung 0 des Port B
;=====

_serPatternLen = 11 ;Länge der Folge

_serPauseT = 255 ;Anzahl an Schleifendurchläufe für Tw

;=====

;User-IRQ Vektor für den IRQ A, der vom Anwender umgebogen wird
UIRQAVek = $30

;Der Timer-Interrupt kann vom Anwender genutzt werden, wenn kein
;Sendevorgang und kein Empfangsvorgang aktiv ist. Nach einem
;Sende- bzw Empfangsvorganges müssen CMCR, LL und UL vom Anwender
;neu beschrieben werden. Die Nutzung sollte daher nur dann
;stattfinden, wenn gewährleistet ist, dass kein Sende-/
;Empfangsvorgang gestartet werden kann. Wie UIRQAVek muss auch
;dieser Vektor zuvor vom Anwender umgebogen werden!
UIRQTVek = $32
=====
```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

;=====
    .ORG progAddr
;=====
;kritische Bereiche??
;nur Akku wird verändert
;Parameter:
;    0      -> Löschen aller Einstellungen der ser. Schnittstelle
;    1      -> Senden erlauben / jedoch den Empfang verbieten
;    sonst -> Senden sowie Empfang erlauben
;    MSB = 1 -> Verwendung von Leitung 0 des PORT B als Eingang
;=====

SPSIni:
    ;Prüfe, ob MSB gesetzt -> Leitung 0 des PORT B verwenden
    PHA          ;BIT-Befehl nicht möglich, dh. Sichern
    LDA #0
    ;Setze Timer zurück
    STA CMCR
    LDA CRB
    AND #%11000100
    STA CRB
    PLA
    PHA
    AND #$80
    BEQ _SPSIni1

    ;MSB war gesetzt, dh. Pin 0 des PORT B als Eingang verwenden
    LDA CRB
    AND #%11111011      ;Zugriff auf DDRB
    STA CRB

    LDA DDRB           ;Lade Datenrichtungsregister
    AND #%11111110      ;Pin 0 als Eingang schalten
    STA DDRB

    LDA CRB
    ORA #%00000100      ;Jetzt wieder Zugriff auf DRB
    STA CRB

    LDA _serSpecFlags
    ORA #_serSpecLine   ;Verwendung von Pin 0 festhalten
    STA _serSpecFlags

_SPSIni1:
    PLA          ;Parameter zurücksichern

    ;Lösche MSB
    AND #$7F        ;Frage Akku ab (N,Z-Flags setzen)
    BEQ _SPSIniReset ;Bei Parameter == 0 ,alles Zurücksetzen
    PHA          ;Parameter auf den Stack sichern
  
```

```

SEI
;Verbiegen des UserIRQ-Vektor
LDA #<_USR_IRQ
STA UIRQVek
LDA #>_USR_IRQ
STA UIRQVek+1

;CB2 als Ausgang, CB2 := CRB.3
;CB1 als Eingang, fallende Flanke
LDA #%00111101
STA CRB
LDA DRA
LDA DRB
LDA SL
CLI

;Freigabe für Empfang zunächst setzen
LDA _serFlags
ORA #_serRcvEn
STA _serFlags

PLA           ;hole Parameter vom Stack
CMP #1
BNE _SPSIniEnd ;Wenn Parameter != 1, verlasse Routine

;blockiere Empfang wenn Parameter == 1
;EN-Flag löschen
LDA _serFlags
AND #_serRcvDis
STA _serFlags
BNE _SPSIniEnd

_SPSIniReset:
;Setze alle Einstellungen zurück
;User-Interrupt Vektor zurücksetzen???
SEI
LDA #<IRQEnde
STA UIRQVek
LDA #>IRQEnde
STA UIRQVek + 1
LDA #0
;Setze Timer zurück
STA CMCR
LDA DRA
LDA DRB
LDA SL
CLI

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

LDA #0
; Spezielle Flags zurücksetzen
STA _serSpecFlags

; CountBits zurücksetzen
STA _serCountBits

; Puffer zurücksetzen
STA _serBuf
STA _serBuf + 1

; CB2 als Eingang, keine Freigabe
; CB1 als Eingang, keine Freigabe
; Zugriff auf DRB festlegen
ORA #0000001000
STA CRB

; Flags zurücksetzen
LDA #1
STA _serFlags

_SPSIniEnd:
RTS
;-----
  
```

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

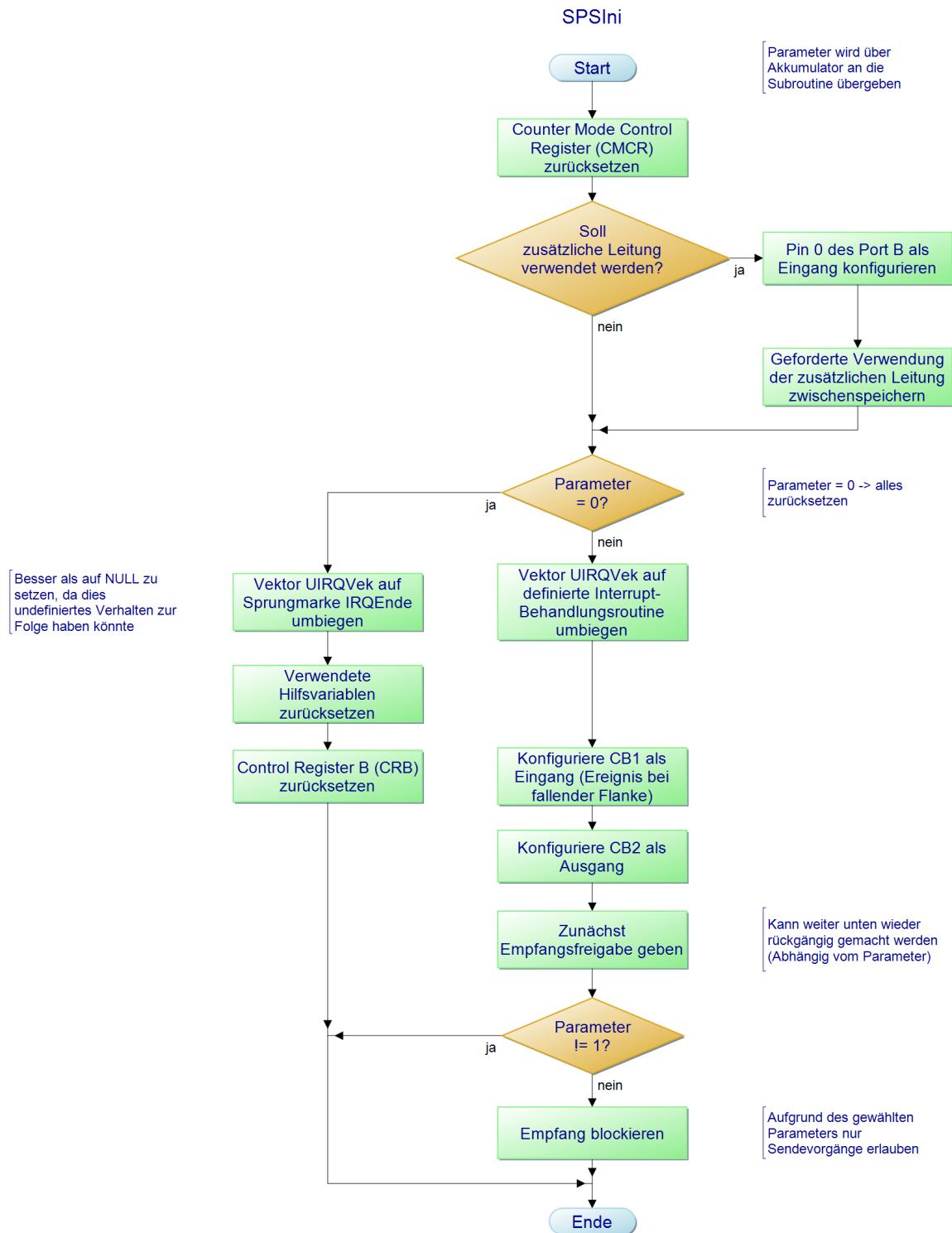


Abbildung 45 PAP SPSIni

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

;=====
;Diese Subroutine prüft, ob die SPS ein Datenbyte übertragen möchte
;und empfängt dieses ggfs. Der Durchlauf der Subroutine kann bis zu
;500ms dauern. Daher sollte diese Routine nur bei Bedarf verwendet
;werden.
;=====

receiveCheck:
    PHA

    TYA
    PHA

    TXA
    PHA

    LDA #2
    JSR SPSIni

    LDY #$FF
receiveChecklop1:
    NOP
    LDX #$FF
receiveChecklop2:
    NOP
    NOP
    NOP
    DEX
    BNE receiveChecklop2

    LDA _serFlags           ;Prüfen, ob Empfang inzwischen aktiv
geworden ist
    AND #_serRcvAct
    BNE receiveChecklop3   ;Wenn ja, verlasse Warteschleife
schon jetzt

    DEY
    BNE receiveChecklop1

receiveChecklop3:
    LDA _serFlags
    AND #_serRcvAct
    BNE receiveChecklop3

;.....Delay after receive.....
;Da mit SPSIni die Interrupts kurzzeitig gesperrt werden würde ohne
;das Delay der Status der CB1 Leitung via IRQ nicht nachgeführt!
    LDY #$7F
Wait02:
    LDX #$FF
  
```

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

Wait01:

```
NOP  
NOP  
NOP  
DEX  
BNE Wait01  
DEY  
BNE Wait02
```

```
; .....
```

```
LDA #1 ;Nur senden freigeben, CB2 bleibt auf 1!  
JSR SPSIni
```

```
PLA  
TAX ;X-Register rücksichern
```

```
PLA  
TAY ;Y-Register rücksichern
```

```
PLA ;Akku rücksichern
```

```
RTS
```

```
-----
```

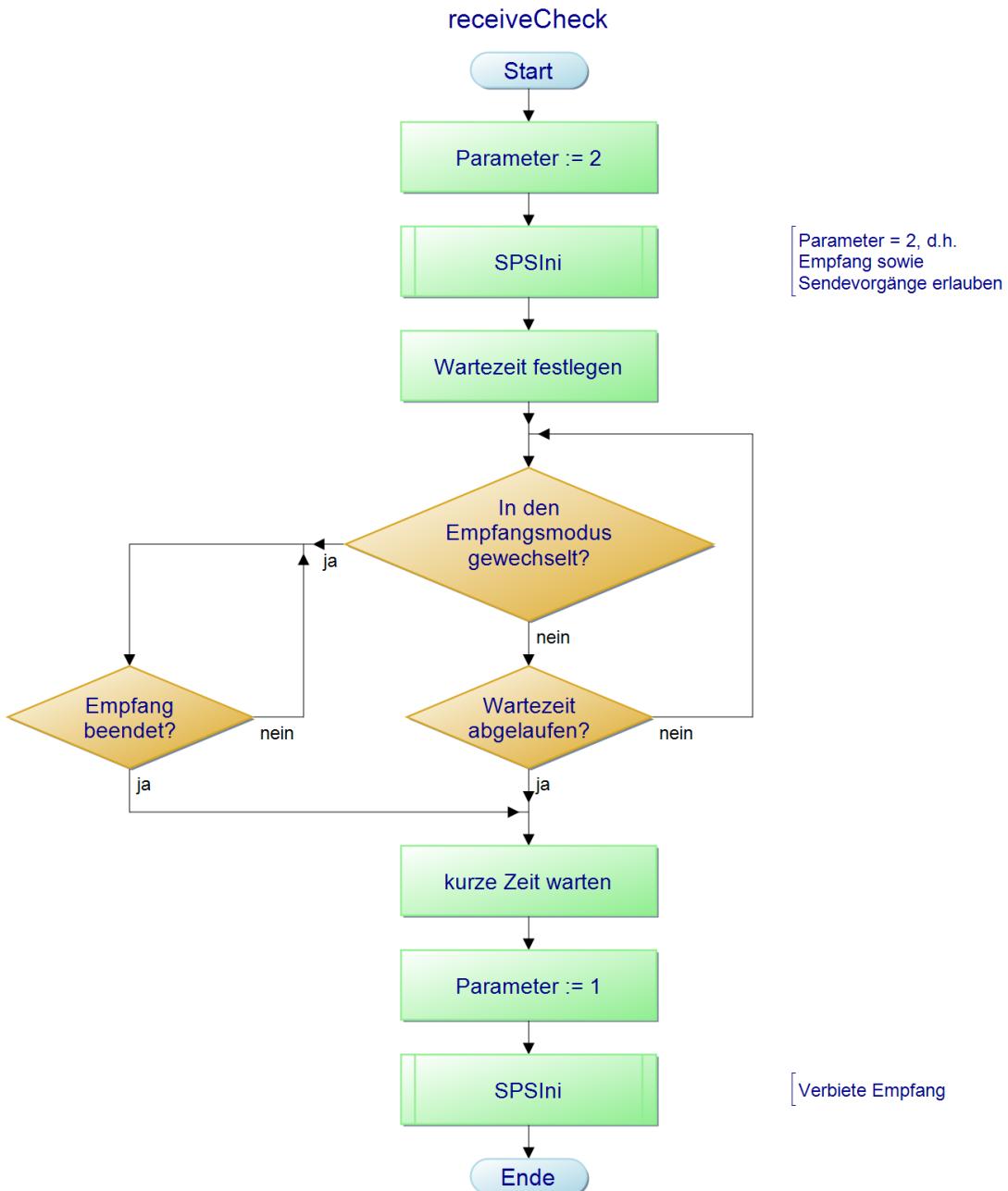


Abbildung 46 PAP receiveCheck

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

;=====
;Senden eines Bytes
;Parameter:
;    im Akku -> Datenbyte
;    im X - Register -> Warte auf KP(0), Nicht warten(sonst)
;Rückgabewert im X-Register:
;    0 -> kein Fehler
;    1 -> Fehler vom Empfänger erkannt
;    2 -> Noch selbst im Empfangsmodus
;    3 -> Fehler vom Empfänger erkannt bzw. vorzeitige Beendigung
;          des Sendevorganges (Empfänger wechselt nicht in den
;          Empfangsmodus
;=====

SPSAus:
    ;sichere Akku auf den Stack, der Wert soll später noch
    ;verfügbar sein
    PHA

;Warte darauf, dass der Empfangsvorgang abgeschlossen ist. Dies
;muss beachtet werden, da der Empfang im Hintergrund läuft...
_SPSAus1:
    NOP
    LDA _serFlags
    AND #_serRcvAct
    ;Weiter, wenn nicht im Empfangsmodus
    BEQ _SPSAus2
    TXA
    ;Warte, wenn X-Register = 0
    BEQ _SPSAus1

    ;Breche mit Fehlermeldung 2 ab
    LDX #2
    JMP _SPSAusEnde

;Warte darauf, dass der Kommunikationspartner seine
;Sende-Leitung wieder auf 1 setzt
_SPSAus2:
    NOP

;Wenn Pin 0 von PORT B verwendet wird, dann lese den aktuellen
;Zustand der Leitung ein
    LDA _serSpecFlags
    AND #_serSpecLine
    BEQ _SPSAUS2L

;Pin 0 wird, verwendet, lese Zustand ein
;Lösche zunächst das Zustands-Flag
    LDA _serFlags
    AND #_serDelCB1

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

STA _serFlags

;Lese Zustand von Pin 0
LDA DRB
AND #_serSetCB1
ORA _serFlags
STA _serFlags

_SPSAUS2L:
;Pin 0 wird nicht als Eingang verwendet

LDA _serFlags
LSR
;Weiter, wenn KP bereit ist
BCS _SPSAus2_1
TXA
;Warte, wenn X-Register = 0
BEQ _SPSAus2

;Breche mit Fehlermeldung 3 ab
LDX #3
JMP _SPSAusEnde

_SPSAus2_1:
;KRITISCHER BEREICH
;BEGRÜNDUNG: Eine fallende Flanke würde in den Empf-
;Modus wechseln
SEI

;Löse sofort einen IRQ beim Kommunikationspartner aus
;Schreibe also das Startbit auf die Leitung
LDA CRB
AND #_serCB2Del
STA CRB

;Starte Timer
;Vorteiler an, IRQ-T aktivieren, Betriebsart 2
LDA #%10010010
STA CMCR
LDA #_TimerLL
STA LL
LDA #_TimerUL
STA ULEC

;Anzahl der zu übertragenen Bits setzen
;Startbit ist bereits gesetzt
;Nach Stopbit erfolgen noch zwei Abfragen:
;Abfrage auf 0, Abfrage Erfolg
;Zum Schluss muss noch auf Empf gewartet werden

```

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

;Daher insgesamt 13 Timer IRQs
LDA #_serPatternLen + 2
STA _serCountBits

;Flag für Sendemodus setzen
LDA _serFlags
ORA #_serSndAct

;Startbit, Stopbit, Parität, Fehlerkennung des Flagbytes setzen
;(nicht unbedingt notwendig)
AND #%00000111
ORA #_serStopbit
STA _serFlags

;Ermittle Parität
TSX
LDA $0101, X
JSR _parity

;Schiebe Paritätsbit nach Bit 6 des Flag-Bytes
TXA
LSR
ROR
LSR
ORA _serFlags
STA _serFlags

;Lösche Puffer und dann 11-Bit Frame erzeugen
LDA #0
STA _serBuf
STA _serBuf + 1

;Stopbit nach Frame
SEC
ROR _serBuf

;Parität nach Frame
TXA ;Parität ist noch im X-Register
LSR
ROR _serBuf

;Datenbits nach Frame
TSX
LDA $0101, X
LDX #8
_SPSAus3:
ASL
ROR _serBuf
ROR _serBuf + 1

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

DEX
BNE _SPSAus3

;Startbit wurde bereits geschrieben und muss nicht
;in den Frame

;ENDE KRITISCHER BEREICH
CLI

;Warte solange bis (Miss-)Erfolg des Sendevorganges vorliegt
;Also 13 Timer-Durchläufe lang warten
;Abbrechen, wenn Fehler während der Übertragung auftritt
_SPSAus4:

;KRITISCHER BREICH:
;Das Laden und Vergleichen von Count-Bits muss als atomare
;Aktion betrachtet werden.
;Szenario: Nach dem Laden von _serCountBits wird ein IRQ
;ausgelöst und _serCountBits darin um eins dekrementiert.
;Der Anschließende Vergleich würde dementsprechend falsch
;ausgewertet.
;Extremfall (wurde von uns beobachtet): Wird ein NACK gelesen
;werden, dann darf diese Schleife nicht verlassen werden, dies
;geschieht dennoch, da im X-Register der falsche Wert von
;_serCountBits steht (eins zu Viel)
SEI
LDX _serCountBits
CPX #2

;Während Übertragung von Daten prüfen ob CB1 = 0
;Also bis nach dem Stopbit diese Leitung prüfen
BMI _SPSAus5
CLI
;ENDE KRITSCHER BEREICH

;Prüfe ob CB1 = 0 noch gilt
LDA _serFlags
AND #_serErrFlg
;Breche ab, wenn Fehler aufgetreten ist
BNE _SPSAus6
_SPSAus5:
CLI
;ENDE KRITSCHER BEREICH
;Prüfe Anzahl der Timer-Durchläufe
NOP
NOP
NOP
NOP
LDX _serCountBits

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

BNE _SPSAus4

_SPSAus6:
; Setze Sende-Leitung auf Idle
LDA CRB
ORA #_serCB2Set
STA CRB

; Sendevorgang abgeschlossen, Flag-Byte anpassen
LDA _serFlags
AND #_serSndNAct
STA _serFlags

; Stoppe den Timer
LDA #0
STA CMCR

; Sichere Y-Register
TYA
PHA

LDY #200
_SPSAus7:
; Warte die Zeit Y*Tw, (möglichst klein halten)
; Ermöglicht den ehemaligen Empfänger eine Übertragung

; Innere Schleife
_SPSAus8:
NOP
NOP
NOP
DEX
BNE _SPSAus8

; Äußere Schleife
DEY
BNE _SPSAus7

; Lade Y-Register zurück
PLA
TAY

; Lade Fehler-Bit ins X-Register (Bit 0)
LDA _serFlags
AND #_serErrFlg
ASL
ROL
TAX
  
```

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

_SPSAusEnde:

PLA ; ursprüngliches Datenbyte zurücksichern

RTS

; -----

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

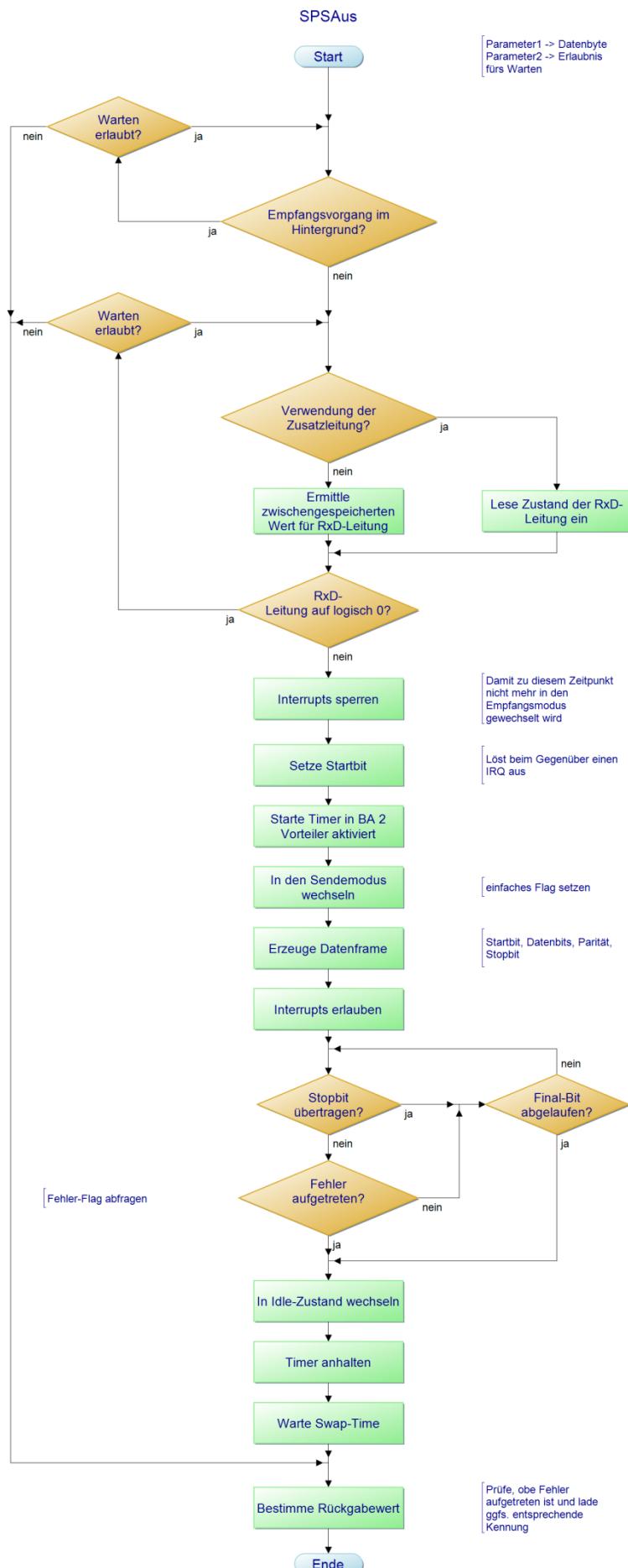


Abbildung 47 PAP_SPSAus

```

; =====

;User-Interrupt Routine
;-----
;Verwendung von Timer-Interrupts und IRQB dürfen vom Anwender NICHT!
;benutzt werden, da ansonsten die Kommunikation mit der SPS nicht
;mehr möglich ist!
;=====

_USR_IRQ:
    ;Frage die Interruptquelle ab
    ;PRIORITÄT B > A IMPLEMENTIEREN
    BIT SR
    BMI _IRQ_T
    BVS _IRQ_A
    JMP _IRQ_B

;=====
;IRQ_A kann vom Anwender benutzt werden! Dazu muss jedoch der
;Zeiger UIRQAVek umgebogen werden! Der Anwender muss außerdem in
;seiner Subroutine als letzte Anweisung "JMP IRQAEnde" verwenden
;=====
_IRQ_A:
    LDA DRA
    LDA UIRQAVek+1
    ;Wenn UIRQAVek = 0, also noch nicht umgebogen, dann bei einem
    ORA UIRQAVek      ;IRQA diesen quittieren!
    BEQ IRQAEnde
    JMP (UIRQAVek)

IRQAEnde:
    JMP IRQEnde

;=====

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

; =====
_IRQ_T:

LDA SL      ; Lösche UF-Bit

; Prüfe ob Sendevorgang aktiv ist
LDA _serFlags
TAX
AND #_serSndAct
BNE _IRQ_T1

; Prüfe, ob Empfangsvorgang aktiv ist
TXA
AND #_serRcvAct
BNE _IRQ_T2JMP

; LDA #0
; STA CMCR
; JMP _IRQ_TEnde
; Sende- und Empfangsmodus sind nicht aktiv
; Springe in User-Interrupt-Routine
LDA UIRQTVek+1
; Wenn UIRQAVek = 0, also noch nicht umgebogen, dann bei einem
ORA UIRQTVek      ; IRQA diesen quittieren!
BEQ IRQTEnde

JMP (UIRQTVek)

IRQTEnde:
JMP IRQEnde

_IRQ_T2JMP:
; Empfangsvorgang ist aktiv, Springe also dort hin
JMP _IRQ_T2

```

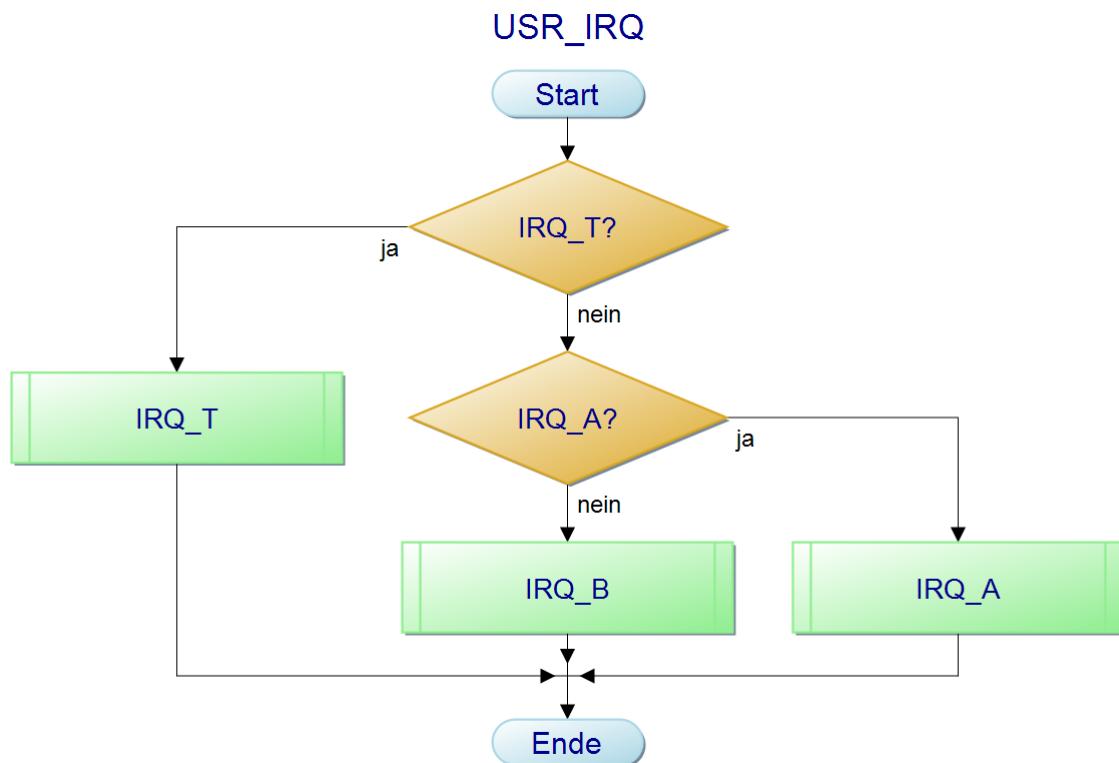


Abbildung 48 PAP USR_IRQ

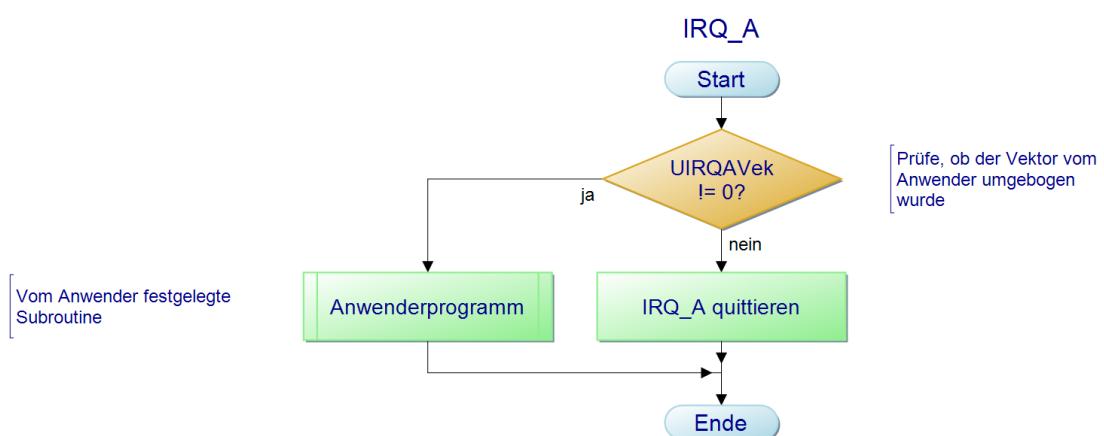


Abbildung 49 PAP IRQ_A

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

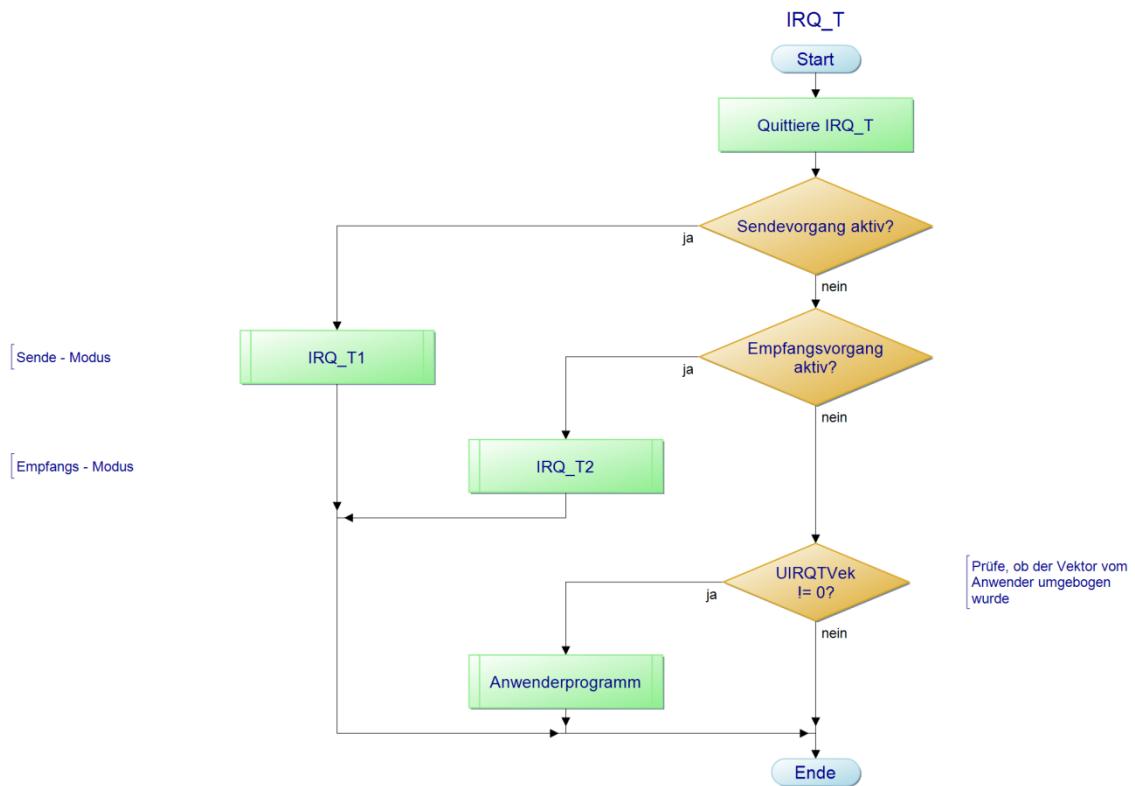


Abbildung 50 PAP IRQ_T

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

;=====
; Sende-Modus
;=====

 IRQ_T1:
 ;Sendemodus ist aktiv

 ;Wenn Pin 0 von PORT B verwendet wird, dann lese den aktuellen
 ;Zustand der Leitung ein
 LDA _serSpecFlags
 AND #_serSpecLine
 BEQ _IRQ_TL1

 ;Pin 0 wird, verwendet, lese Zustand ein
 ;Lösche zunächst das Zustands-Flag
 LDA _serFlags
 AND #_serDelCB1
 STA _serFlags

 ;Lese Zustand von Pin 0
 LDA DRB
 AND #_serSetCB1
 ORA _serFlags
 STA _serFlags

 IRQ_TL1:
 ;Dekrementiere Anzahl der Bits
 LDX _serCountBits
 DEX
 STX _serCountBits

 ;Abfrage Leitung CB1 auf 0, bei IRQ 11
 ;Außerdem Leitung CB2 auf 0 setzen
 CPX #2
 BEQ _IRQ_T14

 ;Abfrage Erfolg, bei T_IRQ 12
 CPX #1
 BEQ _IRQ_T15

 ;Sofort Beenden, bei T_IRQ 13
 TXA
 BEQ _IRQ_T16

 ;Leitung CB1 muss auf Null sein, sonst Fehler
 LDA _serFlags
 LSR
 BCC _IRQ_T12

 ;Fehler, Empfänger ist nicht mehr Empf.-Bereit

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

LDA _serFlags
ORA #_serErrFlg
STA _serFlags
JMP _IRQ_TEnde

_IRQ_T12:
;Ermittle nächstes Datenbit
ASL _serBuf + 1
ROL _serBuf
;Bit ist im Carry-Flag

;Zunächst CB2 auf 0 setzen
LDA CRB
AND #_serCB2Del
BCC _IRQ_T13
;Setze Leitung, wenn Datenbit = 1
ORA #_serCB2Set

_IRQ_T13:
;Schreibe Datenbit auf die Leitung
STA CRB
JMP _IRQ_TEnde

_IRQ_T14:
;Prüfe, ob Leitung CB1 auf Null ist

;Überprüfe CB1
LDA _serFlags
LSR
BCC _IRQ_T14_1 ;Verzeige wenn CB1 = 0

;Fehler: Setze FehlerFlag
LDA _serFlags
ORA #_serErrFlg
STA _serFlags
JMP _IRQ_TEnde

_IRQ_T14_1:
;Setze Leitung CB2 auf Null, alles erfolgreich
LDA CRB
AND #_serCB2Del
STA CRB
JMP _IRQ_TEnde

_IRQ_T15:
;Überprüfe den Erfolg des Sendevorganges
LDA _serFlags
LSR
;Verzweige, wenn Übertragung erfolgreich
BCS _IRQ_T16
  
```

```
; Fehler: Setze FehlerFlag
LDA _serFlags
ORA #_serErrFlg
STA _serFlags
JMP _IRQ_TEnde

_IRQ_T16:
;Bei letztem Timer IRQ sofort Beenden,
;der Sendevorgang ist abgeschlossen
JMP _IRQ_TEnde
;-----
```

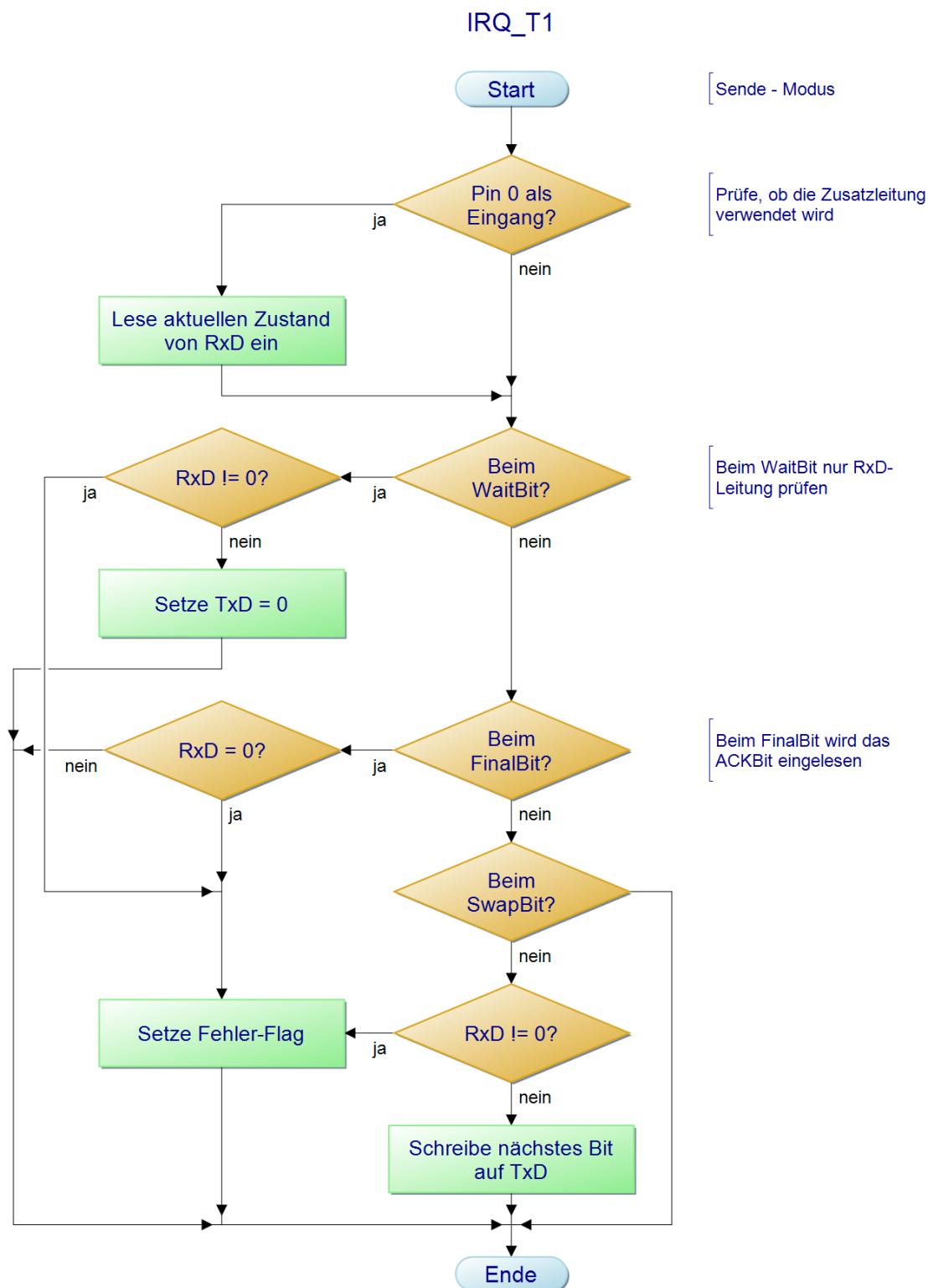


Abbildung 51 PAP IRQ_T1

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

;=====
;Empfangsmodus
;=====

 IRQ_T2:
 ;Empfangsmodus ist aktiv

 ;Wenn Pin 0 von PORT B verwendet wird, dann lese den aktuellen
 ;Zustand der Leitung ein
 LDA _serSpecFlags
 AND #_serSpecLine
 BEQ _IRQ_TL2

 ;Pin 0 wird, verwendet, lese Zustand ein
 ;Lösche zunächst das Zustands-Flag
 LDA _serFlags
 AND #_serDelCB1
 STA _serFlags

 ;Lese Zustand von Pin 0
 LDA DRB
 AND #_serSetCB1
 ORA _serFlags
 STA _serFlags

 IRQ_TL2:
 ;Prüfe Abtastzeitpunkt
 LDX _serCountBits
 BNE _IRQ_T3

 ;Timer bei erstem Abtastzeitpunkt neu initialisieren
 ;Starte Timer neu mit normaler Abtastzeit
 LDA #_TimerLL
 STA LL
 LDA #_TimerUL
 STA ULEC

 IRQ_T3:
 ;Inkrementiere Anzahl an gelesenen Bits
 ;bzw. Anzahl der Timer-IRQs
 INX
 STX _serCountBits

 ;Nachdem die Parität berechnet wurde, muss Erfolgskode
 ;auf die Leitung CB2 gelegt werden
 CPX #_serPatternLen + 1
 BEQ _IRQ_T4

 ;Ganz zum Schluss muss wieder in Idle gewechselt werden

```

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

CPX #_serPatternLen + 2
BEQ _IRQ_T6

;Lese Bit von der Leitung CB1
LDA _serFlags
LSR ;Schiebe Bit ins C-Flag

ROR _serBuf ;Schiebe C-Flag nach buf
ROR _serBuf + 1

;Wenn Anzahl = 11 erreicht, dann die Parität berechnen
CPX #_serPatternLen
BEQ _IRQ_T7

;Ansonsten Beenden
JMP _IRQ_TEnde ;BNE funktioniert hier nicht (Reichweite)

_IRQ_T4:
;Leitung TxD des Senders sollte auf Null sein, sonst Fehler
LDA _serFlags
LSR
BCC _IRQ_T4L

;Fehler, Leitung TxD des Sender ist gleich 1
LDA _serFlags
ORA #_serErrFlg
STA _serFlags

_IRQ_T4L:
;Lege Ergebnis auf CB2
LDA _serFlags
AND #_serErrFlg

;Prüfe auf Fehler
BNE _IRQ_T5
;kein Fehler, Leitung CB2 geht auf 1
LDA CRB
ORA #_serCB2Set
STA CRB

_IRQ_T5:
;Fehler, Leitung CB2 bleibt auf 0
JMP _IRQ_TEnde

_IRQ_T6:
;in Zustand Idle wechseln
LDA CRB
ORA #_serCB2Set
STA CRB

```

```

;Stoppe Timer
LDA #0
STA CMCR

;Flag für Empfangsmodus deaktivieren
LDA _serFlags
AND #_serRcvNAct
STA _serFlags
JMP _IRQ_TEnde

_IRQ_T7:
;Alle 11 Bits wurden bereits eingelesen
;Es bleibt ca. 1 Timer-Zyklus, um die Parität zu
;berechnen, sowie Start- bzw. Stopbit zu prüfen
;Außerdem muss an dieser Stelle der Wert in die
;FIFO eingetragen werden

;Lösche StartBit, StopBit, Parität und Fehlerkenn.
LDA _serFlags
AND #%00001111

ASL _serBuf + 1
ROL _serBuf
;Stopbit befindet sich jetzt im C-Flag

;Prüfe Wert des Stopbits
BCC _IRQ_T8
ORA #_serStopbit;Stopbit = 1

_IRQ_T8:
;Stopbit = 0
ASL _serBuf + 1
ROL _serBuf
;Parity befindet sich jetzt im C-Flag
;und das Datenbyte in "buf"

;Prüfe Wert von Parität
BCC _IRQ_T9
ORA #_serParity ;Parität = 1

_IRQ_T9:
;Parität = 0
ASL _serBuf + 1
;Starbit befindet sich jetzt im C-Flag

;Prüfe Wert des Startbits
BCC _IRQ_T10
ORA #_serStartbit      ;Startbit = 1

_IRQ_T10:
;Startbit = 0

```

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

STA _serFlags ;Schreibe Flags weg

;Berechne Parität
LDA _serBuf      ;Lade Datenbyte
JSR _parity      ;Ergebnis ist im X-Register
TXA

;Rolle ParitätsBit nach Bit 6
LSR
ROR
LSR

EOR _serFlags    ;Prüfe, ob Paritäten übereinstimmen
AND #01110000   ;Hinteren 4 Bits nicht relevant für Fehler
EOR #_serStopbit;Prüfe, ob Stopbit = 1

;Akku = 0 ->korrekt, 1-> Fehler
BEQ _IRQ_T11
LDA _serFlags
ORA #_serErrFlg
STA _serFlags
JMP _IRQ_TEnde

_IRQ_T11:
;Kein Fehler, _serFlg.serErrFlg bleibt auf 0

;Schreibe das Byte in die FIFO
LDA _serBuf
JSR _writeFIFO
SEI           ;_writeFIFO löscht das Flag

;Prüfe ob Eintragung in FIFO erfolgreich war
TXA
BEQ _IRQ_TEnde ;Verzweige, wenn kein Fehler

;Setze das Fehler-Flag im Status-Byte
LDA _serFlags
ORA #10000000
STA _serFlags
JMP _IRQ_TEnde

_IRQ_TEnde:
JMP IRQEnde
;-----

```

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

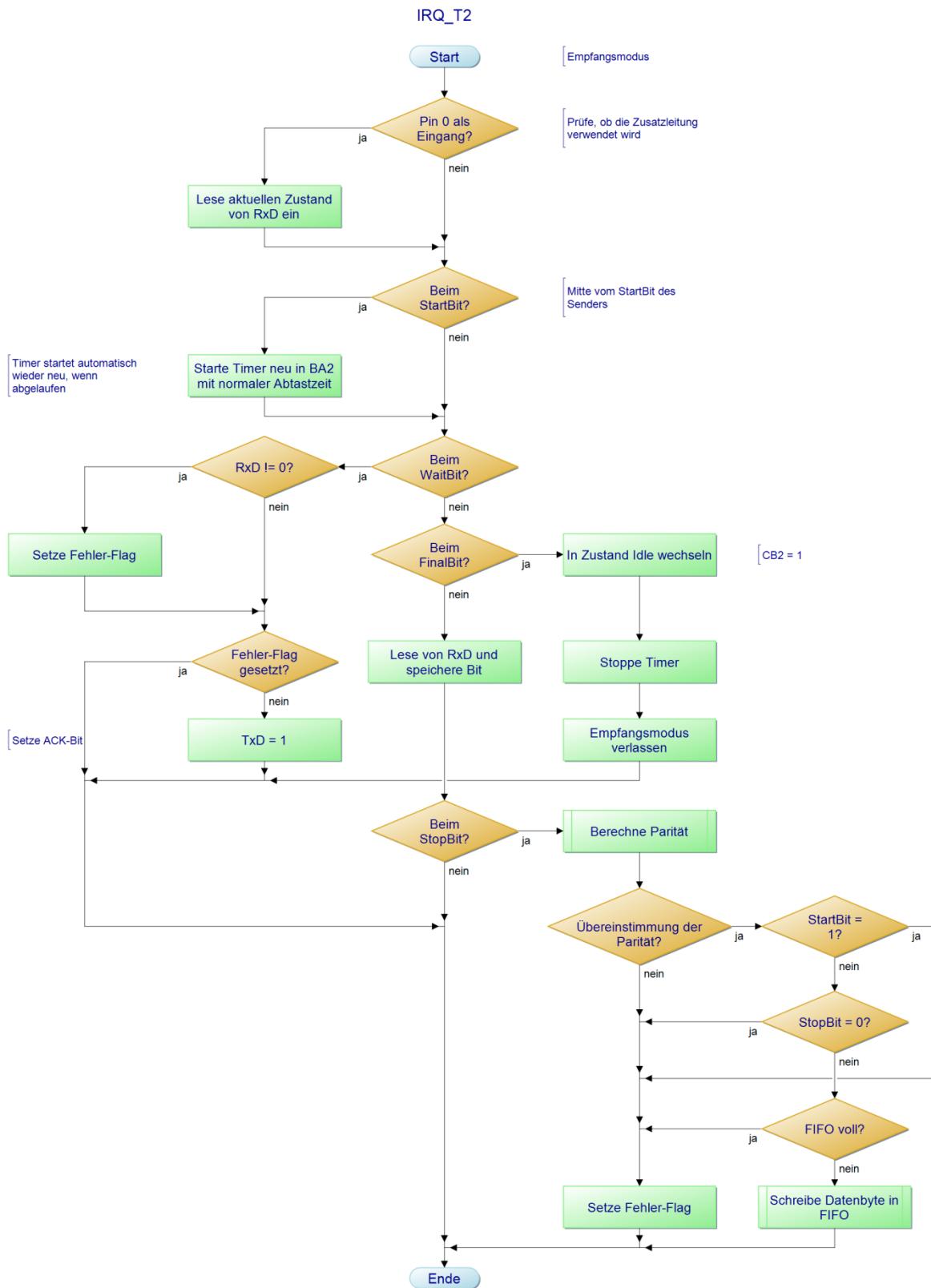


Abbildung 52 PAP IRQ_T2

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

;=====
;Die Leitung CB1 ist nur Ereignisgesteuert, es können also nur
;Flanken gelesen werden. Der Status der Leitung muss daher nach
;Flanke zwischengespeichert werden!
=====

 IRQ_B:
 ;IRQ-B quittieren
 LDA DRB

 ;Lösche Status-Bit der Leitung zunächst
 LDA _serFlags
 AND #_serDelCB1
 TAX

 ;Wenn Pin 0 von PORT B verwendet wird, dann muss keine
 ;Umparametrierung der wirksamen Flanke erfolgen
 LDA _serSpecFlags
 AND #_serSpecLine
 BNE _IRQ_B1

 ;Wahl der wirksamen Flanke invertieren
 LDA CRB
 EOR #_serCRB1Edge
 STA CRB
 AND #_serCRB1Edge
 BNE _IRQ_B1      ;Verzweige, wenn Status der Leitung CB1 == 0

 ;Status der Leitung == 1
 TXA
 ORA #_serSetCB1
 TAX

 IRQ_B1:
 ;CB1-Leitung ist 0
 STX _serFlags

 ;IRQ_B2:
 ;Prüfe, ob und wie Flanke interpretiert werden muss
 ;_serFlags ist noch im X-Register

 ;Bedingung für Wechsel in den Empfangsmodus:
 ;(!SetCB1 & EmpfEn & !EmpfAct & !SndAct)

 ;Prüfe, ob fallende Flanke
 TXA
 LSR
 BCS _IRQ_B3

 ;IRQ_B2:

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

;Fallende Flanke, ggfs. weitere Schritte einleiten (in
;den Empfangsmodus wechseln)

;Prüfe, ob Empfang erlaubt ist
LSR
BCC _IRQ_B4

;Prüfe, ob Empfang aktiv ist
LSR
BCS _IRQ_B5

;Prüfe ob Sendevorgang aktiv ist
LSR
BCS _IRQ_B6

TXA
;Initialisiere Empfangsmodus
ORA #_serRcvAct
STA _serFlags

;Anzahl der empfangen Bits zurücksetzen
LDA #0
STA _serCountBits

;Lösche Eingabe- / Ausgabepuffer
;STX _serBuf

;IRQ-T aktivieren, Vorteiler an, Betriebsart 2
LDA #%10010010
STA CMCR

;Initialisiere Timer mit halber Zeitkonstante
;und starte ihn
LDA #_TimerLLHalf
STA LL
LDA #_TimerULHalf
STA ULEC

;Setze Leitung CB2 auf NULL (Bereitschaft anzeigen)
LDA CRB
AND #_serCB2Del
STA CRB

_IRQ_B3:
;Empfangsmodus kann nicht gestartet werden, da nicht
;fallende Flanke

_IRQ_B4:
;Empfang ist nicht erlaubt

_IRQ_B5:
;Empfangsmodus ist aktiv
  
```

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```
_IRQ_B6:  
    ;Sendemodus ist aktiv  
_IRQ_BEnde:  
    JMP IRQEnde  
;-----
```

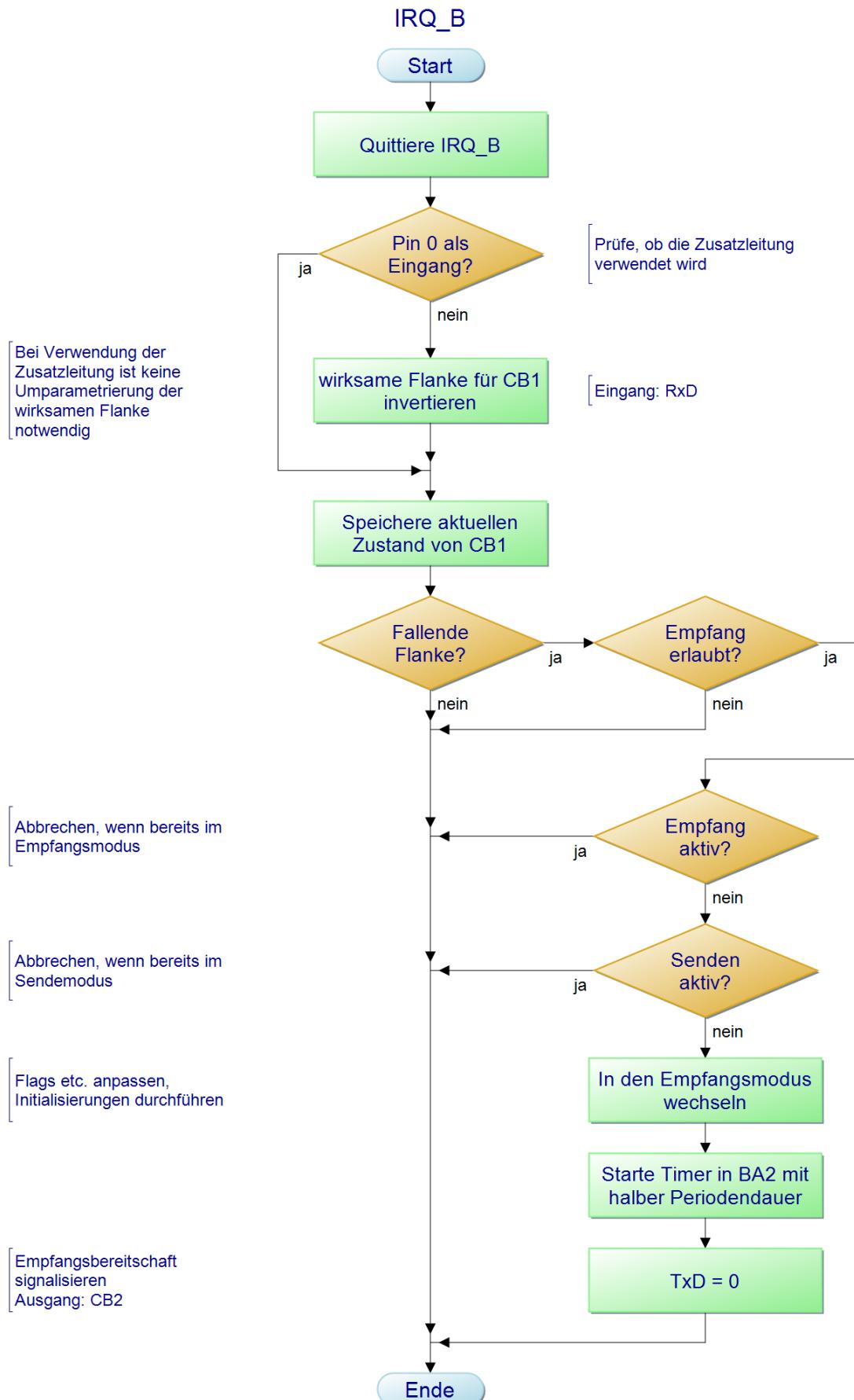


Abbildung 53 PAP IRQ_B

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

;=====
; Bildet die Parität des im Akku übergebenen Bytes. Das Ergebnis wird
; im X-Register zurückgegeben. C-Äquivalent:
;   while(x) { //Anzahl der gesetzten Bits bestimmen
;     count++;
;     x &= (x-1);
;   }
;   count &= 1;      //Parität bestimmen
;=====

_parity:
  PHA          ;Sicherung, wird später zurückgeladen
  PHA          ;gleicher Wert, wird jedoch verändert
  TSX
  LDA #0
  PHA          ;Zähler für die Anzahl der gesetzten Bits
  LDA $0101, X
  BEQ _parity2
_parity1:
  INC $0100, X
  DEC $0101, X
  AND $0101, X
  STA $0101, X
  BNE _parity1
_parity2:
  PLA          ;Anzahl der gesetzten Bits
  AND #1      ;Berechne Parität
  TAX          ;Rückgabewert
  PLA          ;Müll
  PLA          ;gesicherter Wert
  RTS
;-----

```

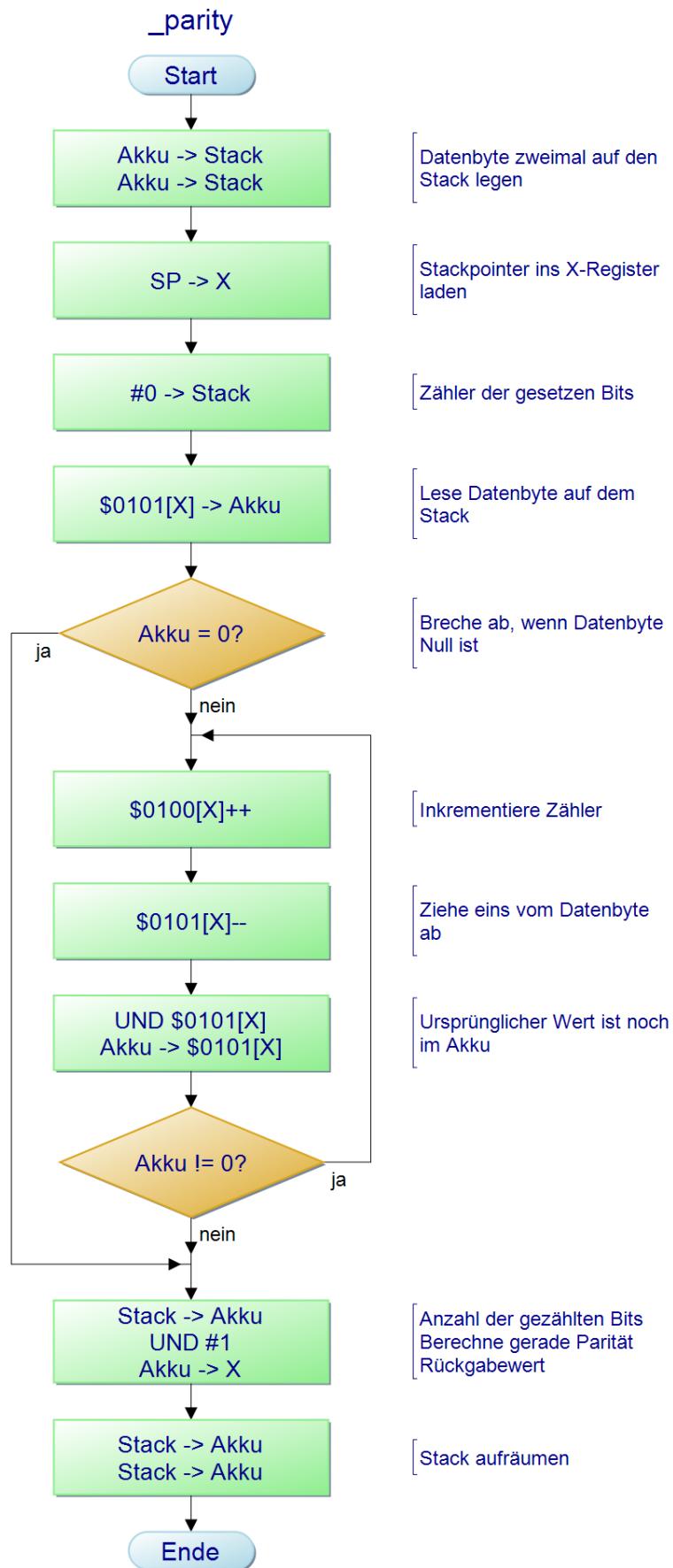


Abbildung 54 PAP parity

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

; =====
; Initialisiert die FIFO.
; =====
_initFIFO:
    ;Verbiete Interrupts solange in dieser Funktion, um Fehler
    ;zu vermeiden
    SEI

    ;Initialisiere IN- und OUT-Offset
    LDA #0
    STA _fifoInOffs
    STA _fifoOutOffs

    ;FIFO ist leer
    LDA #2
    STA _fifoFlags
    CLI
    RTS
; -----

```

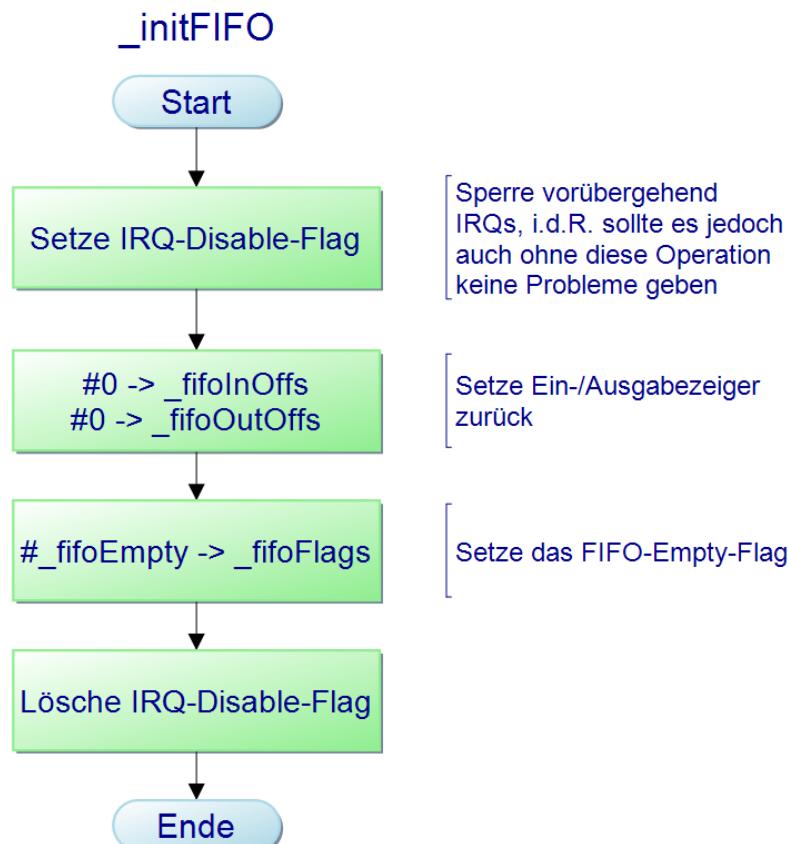


Abbildung 55 PAP initFIFO

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

;=====
;Schreibt den im Akku übergeben Wert in die FIFO. Sollte
;die FIFO voll sein, so wird eine Fehlercode im X-Register
;zurückgegeben.
;Rückgabewerte im X-Register:
;    0 -> Fehlerfreie Ausführung
;    1 -> FIFO voll, Eintragung nicht möglich
;=====

_writeFIFO:
    ;Verbiete IRQs, da im schlimmsten Fall beim Empfangsmodus
    ;gleichzeitig in die FIFO geschrieben wird und dies zu
    ;Inkonsistenzen führen kann
    SEI

    ;Prüfe ob FIFO voll und breche ggfs. ab
    LDX _fifoFlags
    CPX #1
    BEQ _writeFIFO2

    ;Schreibe Wert in FIFO
    LDX _fifoInOffs
    STA _fifo, X

    ;Wert konnte erfolgreich geschrieben werden.
    ;Rückgabewert X-Register = 0
    LDX #0

    ;FIFO ist jetzt nicht mehr leer
    STX _fifoFlags

    ;Inkrementiere IN-Position
    INC _fifoInOffs

    ;Prüfe, ob Ende des Speicherbereiches erreicht
    LDA _fifoInOffs
    CMP #_fifoSize
    BNE _writeFIFO1

    ;Ende ist erreicht, daher In-Offset auf Startwert 0 setzen
    STX _fifoInOffs
    TXA

_writeFIFO1:
    ;Prüfe ob FIFO voll, dh. IN = OUT
    CMP _fifoOutOffs
    BNE _writeFIFO2

    ;FIFO voll, daher Flag setzen
    LDA _fifoFlags
  
```

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```
ORA #00000001
STA _fifoFlags
;INC _fifoFlags ;sollte auch gehen

_writeFIFO2:
;Interrupts wieder erlauben
CLI
RTS
;-----
```

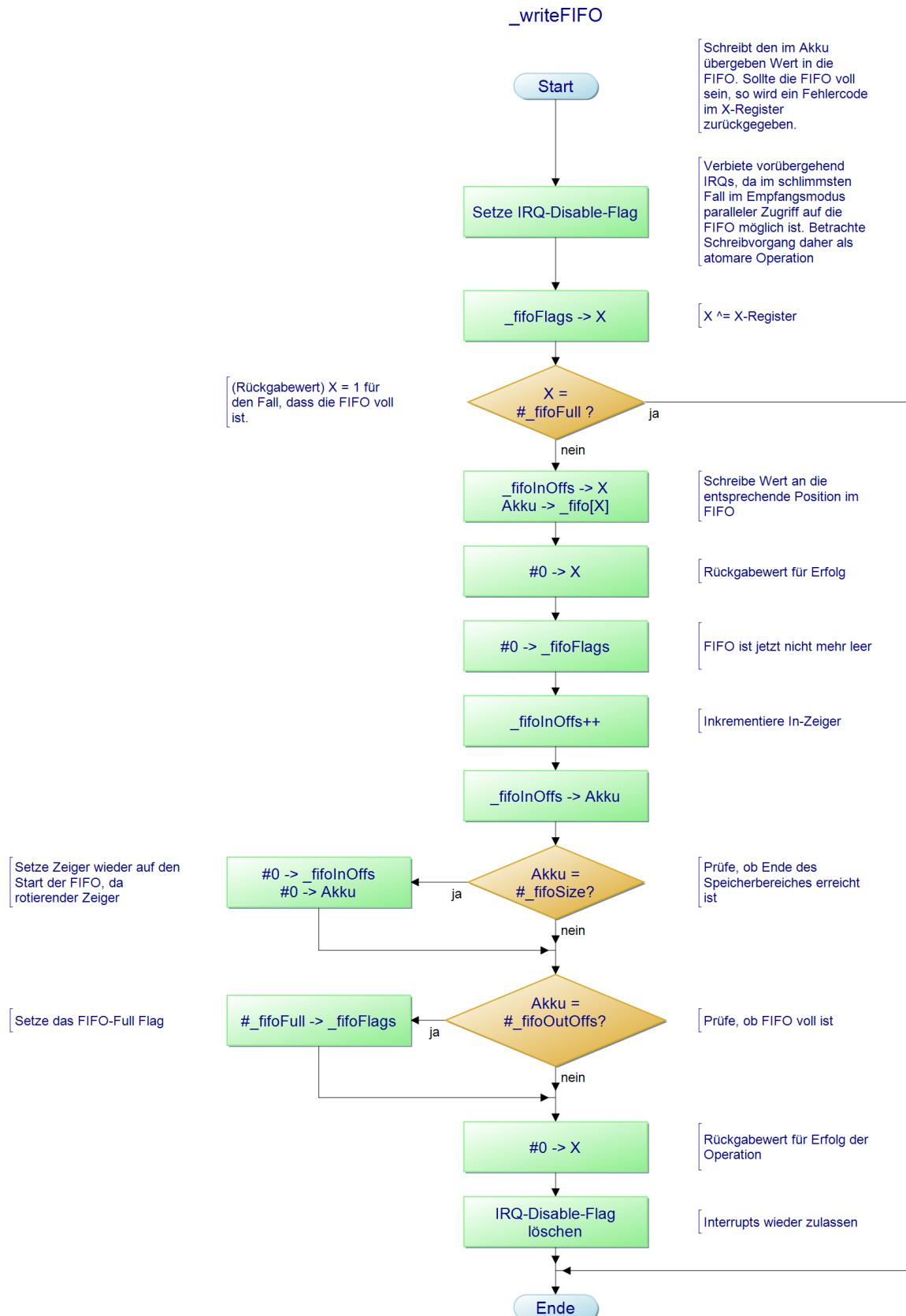


Abbildung 56 PAp writeFIFO

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

;=====
;Ließt einen Wert aus der FIFO und schreibt den Wert in den
;Akku. Fehlerkennungen werden in das X-Register geschrieben
;Rückgabewerte im X-Register:
;    0 -> Konnte Wert erfolgreich aus der FIFO lesen
;    1 -> FIFO ist leer, konnte keinen Wert entnehmen
=====

SPSEin:
_readFIFO:
    ;Verbiete IRQs, da im schlimmsten Fall beim Empfangsmodus
    ;gleichzeitig in die FIFO geschrieben wird und dies zu
    ;Inkonsistenzen führen kann
    SEI

    ;Breche ab, wenn Fifo leer ist
    LDX _fifoFlags
    CPX #2
    BEQ _readFIFO3

    ;Lese Wert aus Fifo und lege vorübergehend auf Stack
    LDX _fifoOutOffs
    LDA _fifo, X
    PHA

    ;Inkrementiere Out-Position
    INX
    STX _fifoOutOffs

    ;FIFO ist nicht mehr Voll
    LDA #0
    STA _fifoFlags

    ;Prüfe, ob Ende des Speicherbereiches erreicht
    CPX #_fifoSize
    BNE _readFIFO1

    ;Ende ist erreicht, daher Out-Offset auf Startwert 0 setzen
    TAX
    STA _fifoOutOffs

_readFIFO1:
    ;Prüfe, ob FIFO leer ist, dh. IN = OUT
    CPX _fifoInOffs
    BNE _readFIFO2

    ;FIFO ist leer, dh. Flag setzen
    LDA _fifoFlags
    ORA #%00000010
    STA _fifoFlags
  
```

```
_readFIFO2:  
    LDX #1      ;Rückgabewert 0  
    PLA        ;Wert in Akku  
  
_readFIFO3:  
    DEX  
    CLI        ;Erlaube ab hier wieder Interrupts  
    RTS  
;-----
```

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

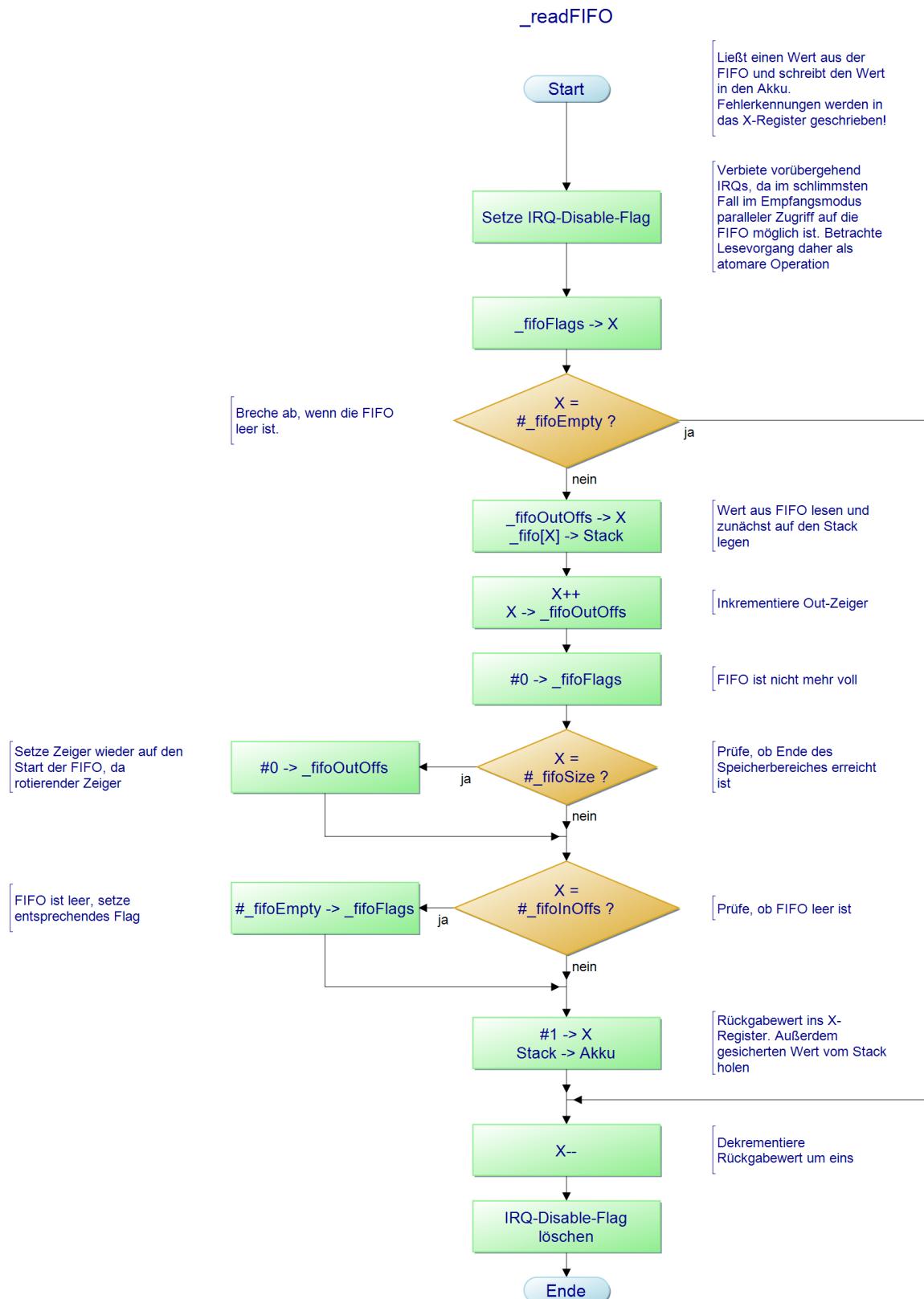


Abbildung 57 PAP readFIFO

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

; =====
; Zählt die Anzahl der belegten Bytes in der FIFO und gibt das
; Ergebnis im Akkumulator zurück.
; Es werden keine Veränderungen der FIFO durchgeführt, Interrupts
; müssen hier also nicht gesperrt werden!
; =====

_countFIFO:
  SEC
  LDA _fifoInOffs
  SBC _fifoOutOffs
  BCC _countFIFO1 ;IN < OUT
  BNE _countFIFO2 ;IN > OUT

  ; IN = OUT
  LDA _fifoFlags
  AND #2
  BNE _countFIFO2 ;FIFO ist leer

  ; FIFO ist voll
  CLC
_countFIFO1:
  ADC #_fifoSize
_countFIFO2:
  RTS
; -----

```

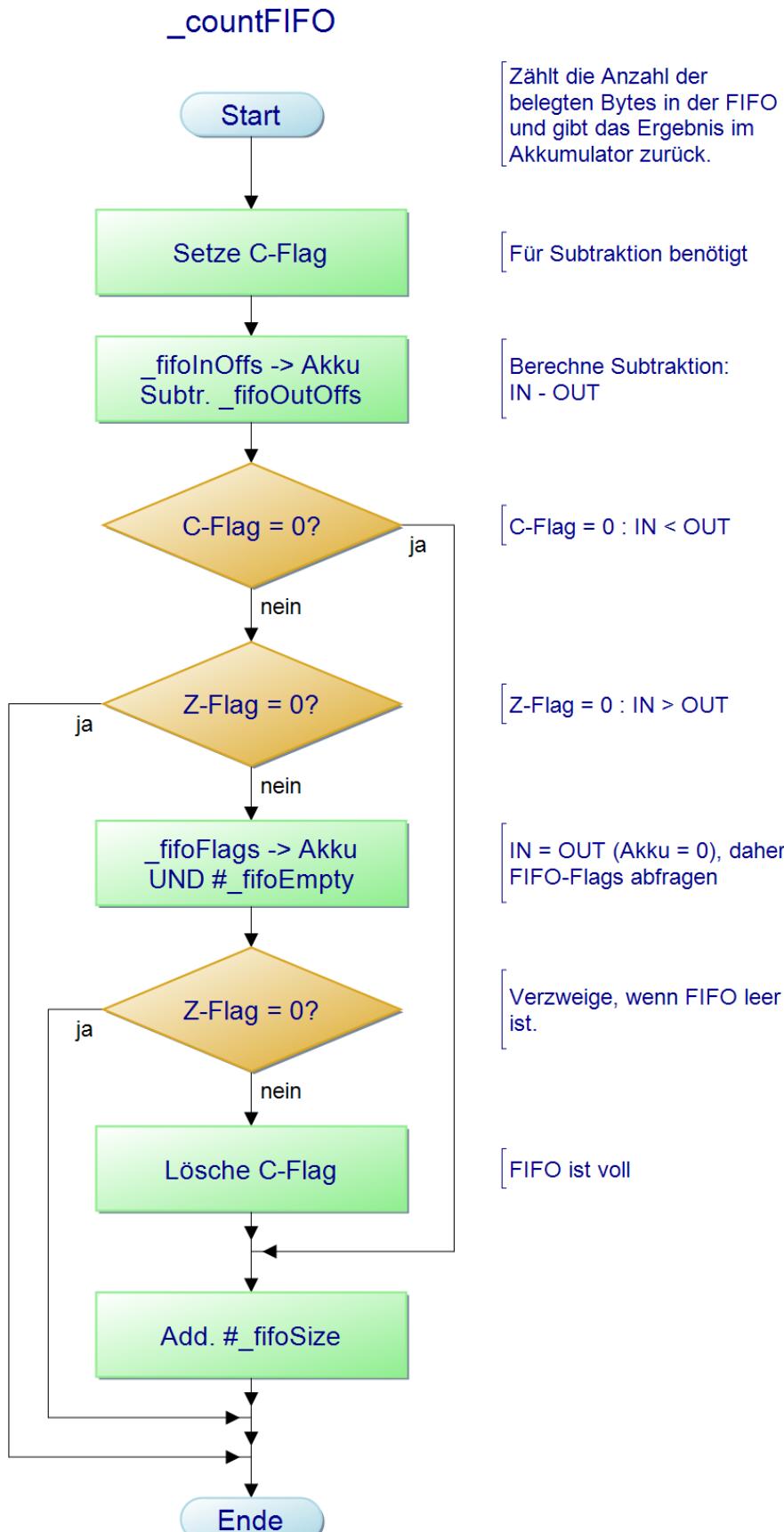


Abbildung 58 PAP countFIFO

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

; =====
; Zählt die Anzahl an freien Bytes in der FIFO und gibt das Ergebnis
; im Akkumulator zurück.
; Es werden keine Veränderungen der FIFO durchgeführt, Interrupts
; müssen hier also nicht gesperrt werden!
; =====
_countEmptyFIFO:
    JSR _countFIFO ; ermittle Anzahl der belegten Bytes in der FIFO
    SEC
    SBC #_fifoSize

    ; Zweierkomplement bilden. Es wird dadurch keine temporäre
    ; Variable benötigt
    EOR #$FF
    SEC
    ADC #0
    RTS
; -----

```

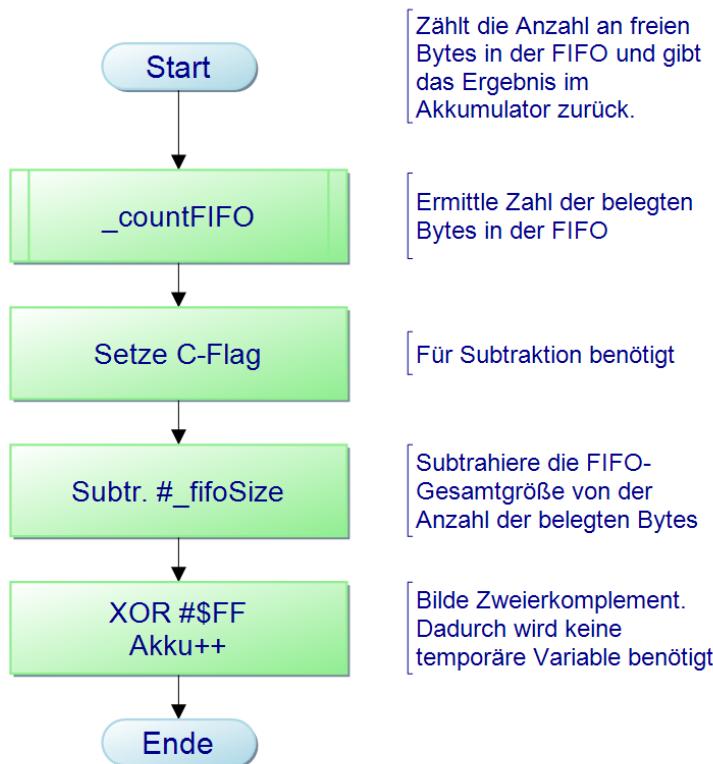
_countEmptyFIFO

Abbildung 59 PAP countEmptyFIFO

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

.ORG progAddr + $04F0
=====
;Bit 0 -> Status Flag der Leitung CB1
; Annahme: zu Beginn = 1
;Bit 1 -> Empfangsmodus erlaubt
;Bit 2 -> Empfangsmodus aktiv
;Bit 3 -> Sendemodus aktiv
;Bit 4 -> StartBit
;Bit 5 -> StopBit
;Bit 6 -> Parität
;Bit 7 -> Fehlerkennung
_serFlags: .BYTE $01
=====

=====;
;Spezielle Flags:
;Bit 0 -> Empfang über die Leitung 0 des Port B erlauben
_serSpecFlags: .BYTE $00
=====

_serCountBits: .BYTE $00
_serBuf: .WORD $0000

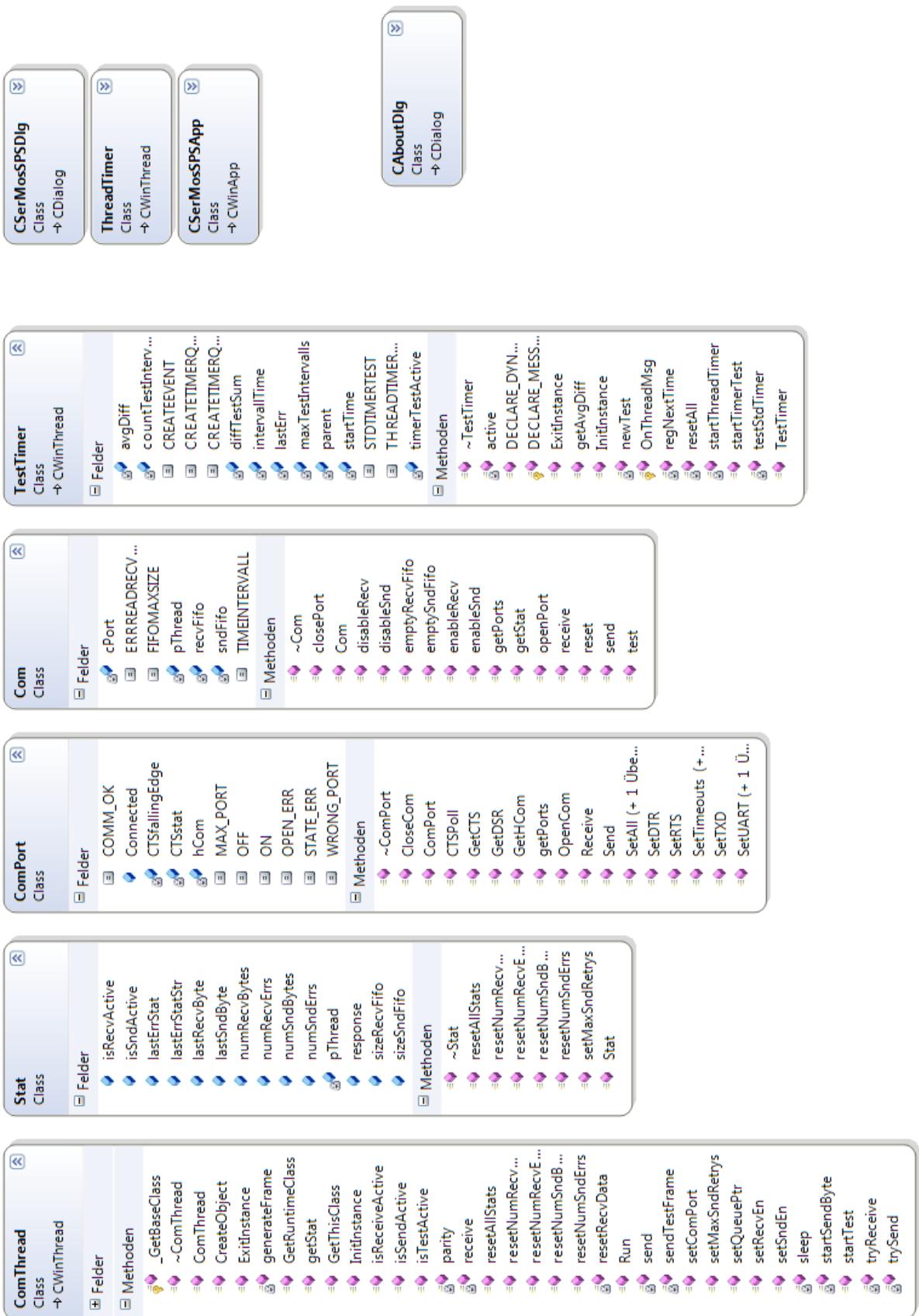
=====

.ORG _fifoAddr
_fifo: .BYTE $00

.ORG _fifoAddr + _fifoSize
;Flags:
;Bit 0: FIFO ist voll
;Bit 1: FIFO ist leer
_fifoFlags: .BYTE $00
_fifoInOffs: .BYTE $00
_fifoOutOffs: .BYTE $00
=====
```

Visual-C++ Quelltext

Klassendiagramm



Klasse: SerMosSPSDlg

```
// SerMosSPSDlg.h : Headerdatei
//

#include "TestTimer.h"
#include "Com.h"
#include "ComThread.h"

#pragma once

//-----
//-----
//-----

// CSerMosSPSDlg-Dialogfeld
class CSerMosSPSDlg : public CDialog
{
// Konstruktion
public:
    CSerMosSPSDlg(CWnd* pParent = NULL);      // Standardkonstruktor
    CSerMosSPSDlg::~CSerMosSPSDlg();           // Destruktor

// Dialogfelddaten
    enum { IDD = IDD_SERMOSSPS_DIALOG };

protected:
    // DDX/DDV-Unterstützung
    virtual void DoDataExchange(CDataExchange* pDX);

// Implementierung
protected:
    HICON m_hIcon;

    // Generierte Funktionen für die Meldungstabellen
    virtual BOOL OnInitDialog();
    afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
    afx_msg void OnPaint();
    afx_msg HCURSOR OnQueryDragIcon();
    afx_msg void OnTimer(UINT nIDEvent);
    DECLARE_MESSAGE_MAP()

private:
    TestTimer* testTimer;
    Com com;

    bool sendOnNextTimer;
    int nextQueueByte;

// Konstanten
static const int MAXQUEUENUM = 100;
```

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```
public:  
    afx_msg void OnBnClickedOk();  
    afx_msg void OnBnClickedButton1();  
    afx_msg void OnBnClickedButton3();  
    afx_msg void OnBnClickedButton2();  
    afx_msg void OnBnClickedButton4();  
    afx_msg void OnBnClickedButton5();  
    afx_msg void OnBnClickedButton6();  
    afx_msg void OnBnClickedButton7();  
    afx_msg void OnBnClickedButton8();  
    afx_msg void OnBnClickedButton9();  
    afx_msg void OnBnClickedButton10();  
    afx_msg void OnBnClickedButton11();  
    afx_msg void OnBnClickedButton12();  
    afx_msg void OnBnClickedButton13();  
    afx_msg void OnBnClickedButton14();  
    afx_msg void OnBnClickedButton15();  
    afx_msg void OnBnClickedButton16();  
    afx_msg void OnBnClickedCheck1();  
    afx_msg void OnBnClickedCheck2();  
    afx_msg void OnBnClickedCheck5();  
    afx_msg LRESULT OnTestTimerFinished(WPARAM, LPARAM);  
    void comConnected(void);  
    afx_msg void OnBnClickedButton17();  
};  
  
//-----  
//-----  
//-----
```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

// SerMosSPSDlg.cpp : Implementierungsdatei
// Beispiel-Anwendung, die die Com-Klasse verwendet.

#include <windows.h>
#include "stdafx.h"
#include "SerMosSPS.h"
#include "SerMosSPSDlg.h"
#include "ThreadTimer.h"
#include "TestTimer.h"
#include "Com.h"
#include "ComThread.h"
#include "Stat.h"

#ifndef _DEBUG
#define new DEBUG_NEW
#endif

// CAboutDlg-Dialogfeld für Anwendungsbefehl "Info"
//-----
//-----
//-
class CAboutDlg : public CDialog
{
public:
    CAboutDlg();

    // Dialogfelddaten
    enum { IDD = IDD_ABOUTBOX };

protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV-Unterstützung

    // Implementierung
protected:
    DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD) {}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)

END_MESSAGE_MAP()

// CSerMosSPSDlg-Dialogfeld
CSerMosSPSDlg::CSerMosSPSDlg(CWnd* pParent /*=NULL*/)
: CDialog(CSerMosSPSDlg::IDD, pParent)
// , curTime(0)
{
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}
//-----

CSerMosSPSDlg::~CSerMosSPSDlg(void)
{
    // Dynamisch erzeugte Objekte löschen
    delete(testTimer);
}
//-----

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

void CSerMosSPSDlg::DoDataExchange(CDataExchange* pDX) {
    CDialog::DoDataExchange(pDX);
}

//-----



// Festlegung der Ereignisse
BEGIN_MESSAGE_MAP(CSerMosSPSDlg, CDIALOG)
    ON_WM_SYSCOMMAND()
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    //} }AFX_MSG_MAP
    ON_BN_CLICKED(IDC_DOK, &CSerMosSPSDlg::OnBnClickedOk)
    ON_BN_CLICKED(IDC_BUTTON1, &CSerMosSPSDlg::OnBnClickedButton1)
    ON_BN_CLICKED(IDC_BUTTON2, &CSerMosSPSDlg::OnBnClickedButton2)
    ON_BN_CLICKED(IDC_BUTTON3, &CSerMosSPSDlg::OnBnClickedButton3)
    ON_BN_CLICKED(IDC_BUTTON4, &CSerMosSPSDlg::OnBnClickedButton4)
    ON_BN_CLICKED(IDC_BUTTON5, &CSerMosSPSDlg::OnBnClickedButton5)
    ON_BN_CLICKED(IDC_BUTTON6, &CSerMosSPSDlg::OnBnClickedButton6)
    ON_BN_CLICKED(IDC_BUTTON7, &CSerMosSPSDlg::OnBnClickedButton7)
    ON_BN_CLICKED(IDC_BUTTON8, &CSerMosSPSDlg::OnBnClickedButton8)
    ON_BN_CLICKED(IDC_BUTTON9, &CSerMosSPSDlg::OnBnClickedButton9)
    ON_BN_CLICKED(IDC_BUTTON10, &CSerMosSPSDlg::OnBnClickedButton10)
    ON_BN_CLICKED(IDC_BUTTON11, &CSerMosSPSDlg::OnBnClickedButton11)
    ON_BN_CLICKED(IDC_BUTTON12, &CSerMosSPSDlg::OnBnClickedButton12)
    ON_BN_CLICKED(IDC_BUTTON13, &CSerMosSPSDlg::OnBnClickedButton13)
    ON_BN_CLICKED(IDC_BUTTON14, &CSerMosSPSDlg::OnBnClickedButton14)
    ON_BN_CLICKED(IDC_BUTTON15, &CSerMosSPSDlg::OnBnClickedButton15)
    ON_BN_CLICKED(IDC_BUTTON16, &CSerMosSPSDlg::OnBnClickedButton16)
    ON_BN_CLICKED(IDC_CHECK1, &CSerMosSPSDlg::OnBnClickedCheck1)
    ON_BN_CLICKED(IDC_CHECK2, &CSerMosSPSDlg::OnBnClickedCheck2)
    ON_BN_CLICKED(IDC_CHECK5, &CSerMosSPSDlg::OnBnClickedCheck5)
    ON_MESSAGE(WM_TESTTIMER_FINISHED, OnTestTimerFinished)
    ON_WM_TIMER()
        ON_BN_CLICKED(IDC_BUTTON17, &CSerMosSPSDlg::OnBnClickedButton17)
END_MESSAGE_MAP()
//-----


// CSerMosSPSDlg-Meldungshandler

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

BOOL CSerMosSPSDlg::OnInitDialog() {
    CDialog::OnInitDialog();
    // Hinzufügen des Menübefehls "Info..." zum Systemmenü.
    // IDM_ABOUTBOX muss sich im Bereich der Systembefehle befinden.
    ASSERT((IDM_ABOUTBOX & 0xFFFF0) == IDM_ABOUTBOX);
    ASSERT(IDM_ABOUTBOX < 0xF000);

    CMenu* pSysMenu = GetSystemMenu(FALSE);
    if (pSysMenu != NULL) {
        CString strAboutMenu;
        strAboutMenu.LoadString(IDS_ABOUTBOX);
        if (!strAboutMenu.IsEmpty())
        {
            pSysMenu->AppendMenu(MF_SEPARATOR);
            pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
        }
    }

    // Symbol für dieses Dialogfeld festlegen. Wird automatisch erledigt
    // wenn das Hauptfenster der Anwendung kein Dialogfeld ist
    SetIcon(m_hIcon, TRUE);           // Großes Symbol verwenden
    SetIcon(m_hIcon, FALSE);          // Kleines Symbol verwenden

    // TODO: Hier zusätzliche Initialisierung einfügen
    // Deaktiviere zunächst alle Radio-Buttons
    GetDlgItem(IDC_RADIO1)->EnableWindow(false);
    GetDlgItem(IDC_RADIO2)->EnableWindow(false);
    GetDlgItem(IDC_RADIO3)->EnableWindow(false);
    GetDlgItem(IDC_RADIO4)->EnableWindow(false);

    // Prüfe die vorhanden COM-Schnittstellen
    CString ports[4];
    com.getPorts(ports);
    if(ports[0] != "")
        GetDlgItem(IDC_RADIO1)->EnableWindow(true);
    if(ports[1] != "")
        GetDlgItem(IDC_RADIO2)->EnableWindow(true);
    if(ports[2] != "")
        GetDlgItem(IDC_RADIO3)->EnableWindow(true);
    if(ports[3] != "")
        GetDlgItem(IDC_RADIO4)->EnableWindow(true);

    // Variablen-Initialisierungen
    sendOnNextTimer = false;
    //sendTestQueue = false;
    testTimer = NULL;

    // Ein Byte hat max. 3 Ziffern (Dez.)
    ((CEdit*)GetDlgItem(IDC_EDIT4))->LimitText(3);

    // Starte Timer mit 1s
    SetTimer(ID_Clock_Timer, 3000, NULL);

    // Zunächst sind unendlich viele Sendeversuche erlaubt
    ((CButton*)GetDlgItem(IDC_CHECK5))->SetCheck(true);

    // Geben Sie TRUE zurück, außer ein Steuerelement soll den Fokus erhalten
    return TRUE;
}
//-----

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

void CSerMosSPSDlg::OnSysCommand(UINT nID, LPARAM lParam) {
    if ((nID & 0xFFFF0) == IDM_ABOUTBOX) {
        CAboutDlg dlgAbout;
        dlgAbout.DoModal();
    }
    else {
        CDialog::OnSysCommand(nID, lParam);
    }
}

//-----
// Wenn Sie dem Dialogfeld eine Schaltfläche "Minimieren" hinzufügen, benötigen Sie
// den nachstehenden Code, um das Symbol zu zeichnen. Für MFC-Anwendungen, die das
// Dokument/Ansicht-Modell verwenden, wird dies automatisch ausgeführt.

void CSerMosSPSDlg::OnPaint() {
    if (IsIconic()) {
        CPaintDC dc(this); // Gerätekontext zum Zeichnen

        SendMessage(WM_ICONERASEBKGND,
                    reinterpret_cast<WPARAM>(dc.GetSafeHdc()), 0);

        // Symbol in Clientrechteck zentrieren
        int cxIcon = GetSystemMetrics(SM_CXICON);
        int cyIcon = GetSystemMetrics(SM_CYICON);
        CRect rect;
        GetClientRect(&rect);
        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;

        // Symbol zeichnen
        dc.DrawIcon(x, y, m_hIcon);
    }
    else {
        CDialog::OnPaint();
    }
}

//-----
// Die System ruft diese Funktion auf, um den Cursor abzufragen,
// der angezeigt wird, während der Benutzer
// das minimierte Fenster mit der Maus zieht.
HCURSOR CSerMosSPSDlg::OnQueryDragIcon() {
    return static_cast<HCURSOR>(m_hIcon);
}

void CSerMosSPSDlg::OnTimer(UINT nIDEvent) {
    Stat stat = com.getStat();
    //CString strTmp;
    char pszNum[100] = {0};
    short response = 0;

    // int test;
    switch(nIDEvent) {
        //Timer-Test
        case ID_Clock_Timer:
            //Timer zunächst stoppen
            KillTimer(ID_Clock_Timer);

            //Senden aktiv?
            ((CButton*)GetDlgItem(IDC_CHECK3))->SetCheck(stat.isSndActive);

            //Empfang aktiv?
            ((CButton*)GetDlgItem(IDC_CHECK4))->SetCheck(stat.isRecvActive);
    }
}

```

```

//Anzahl der übertragenen Bytes
    _itoa_s(stat.numSndBytes, pszNum, 100, 10);
    GetDlgItem(IDC_EDIT5)->SetWindowTextW(CString(pszNum));

//Anzahl der empfangen Bytes
    _itoa_s(stat.numRecvBytes, pszNum, 100, 10);
    GetDlgItem(IDC_EDIT6)->SetWindowTextW(CString(pszNum));

//Anzahl der Sendefehler
    _itoa_s(stat.numSndErrs, pszNum, 100, 10);
    GetDlgItem(IDC_EDIT7)->SetWindowTextW(CString(pszNum));

//Anzahl der Empfangsfehler
    _itoa_s(stat.numRecvErrs, pszNum, 100, 10);
    GetDlgItem(IDC_EDIT8)->SetWindowTextW(CString(pszNum));

//Letzter Fehler
    _itoa_s(stat.lastErrStat, pszNum, 100, 10);
    GetDlgItem(IDC_EDIT13)->SetWindowTextW(CString(pszNum));
    GetDlgItem(IDC_EDIT10)->SetWindowTextW(stat.lastErrStatStr);

//Zuletzt gesendetes Byte
    _itoa_s(stat.lastSndByte, pszNum, 100, 16);
    GetDlgItem(IDC_EDIT11)->SetWindowTextW(_T("0x") + CString(pszNum));

//Zuletzt empfangenes Byte
    _itoa_s(stat.lastRecvByte, pszNum, 100, 16);
    GetDlgItem(IDC_EDIT12)->SetWindowTextW(_T("0x") + CString(pszNum));
//Anzahl der Elemente in der Sende-FIFO
    _itoa_s(stat.sizeSndFifo, pszNum, 100, 10);
    GetDlgItem(IDC_EDIT15)->SetWindowTextW(CString(pszNum));

//Anzahl der Elemente in der Empfangs-FIFO
    _itoa_s(stat.sizeRecvFifo, pszNum, 100, 10);
    GetDlgItem(IDC_EDIT14)->SetWindowTextW(CString(pszNum));

//Aufbau: 0|0|T0|...|T13
response = stat.response;
((CButton*)GetDlgItem(IDC_CHECK38))->SetCheck(response & 1);
((CButton*)GetDlgItem(IDC_CHECK35))->SetCheck((response >= 1) & 1);
((CButton*)GetDlgItem(IDC_CHECK34))->SetCheck((response >= 1) & 1);
((CButton*)GetDlgItem(IDC_CHECK33))->SetCheck((response >= 1) & 1);
((CButton*)GetDlgItem(IDC_CHECK32))->SetCheck((response >= 1) & 1);
((CButton*)GetDlgItem(IDC_CHECK31))->SetCheck((response >= 1) & 1);
((CButton*)GetDlgItem(IDC_CHECK30))->SetCheck((response >= 1) & 1);
((CButton*)GetDlgItem(IDC_CHECK29))->SetCheck((response >= 1) & 1);
((CButton*)GetDlgItem(IDC_CHECK28))->SetCheck((response >= 1) & 1);
((CButton*)GetDlgItem(IDC_CHECK27))->SetCheck((response >= 1) & 1);
((CButton*)GetDlgItem(IDC_CHECK26))->SetCheck((response >= 1) & 1);
((CButton*)GetDlgItem(IDC_CHECK25))->SetCheck((response >= 1) & 1);
((CButton*)GetDlgItem(IDC_CHECK24))->SetCheck((response >= 1) & 1);
((CButton*)GetDlgItem(IDC_CHECK23))->SetCheck((response >= 1) & 1);
((CButton*)GetDlgItem(IDC_CHECK22))->SetCheck((response >= 1) & 1);

//Starte Timer neu
SetTimer(ID_Clock_Timer, 100, NULL);
break;

default:
break;
}

//-----

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

void CSerMosSPSDlg::OnBnClickedOk() {
    OnOK();
}

//-----

void CSerMosSPSDlg::OnBnClickedButton1() {
    if(!testTimer) {

        // Timer-Test wird in einem Thread gestartet
        testTimer = (TestTimer*) AfxBeginThread(RUNTIME_CLASS(TestTimer),
                                                THREAD_PRIORITY_NORMAL,
                                                NULL,
                                                CREATE_SUSPENDED,
                                                NULL);
        testTimer->ResumeThread();
    }
    testTimer->startTimerTest(TestTimer::STDTIMERTEST,
                               AfxGetMainWnd()->m_hWnd, 200, Com::TIMEINTERVALL);
}

//-----


void CSerMosSPSDlg::OnBnClickedButton3() {

    if(!testTimer) {
        // Timer-Test wird in einem Thread gestartet
        testTimer = (TestTimer*) AfxBeginThread(RUNTIME_CLASS(TestTimer),
                                                THREAD_PRIORITY_NORMAL,
                                                NULL,
                                                CREATE_SUSPENDED,
                                                NULL);
        testTimer->ResumeThread();
    }
    testTimer->startTimerTest(TestTimer::THREADETIMERTEST,
                               AfxGetMainWnd()->m_hWnd, 200, Com::TIMEINTERVALL);

    //testTimer->PostThreadMessage(WM_THREAD_TERMINATED,
    //                           WM_THREAD_TERMINATED, NULL);
}
//-----


LRESULT CSerMosSPSDlg::OnTestTimerFinished(WPARAM wParam, LPARAM lParam) {
    CString strTmp;
    strTmp.Format(L"%f", testTimer->getAvgDiff());
    if((int)lParam == TestTimer::THREADETIMERTEST)
        GetDlgItem(IDC_EDIT2)->SetWindowText(strTmp + _T(" ms"));
    else
        GetDlgItem(IDC_EDIT1)->SetWindowText(strTmp + _T(" ms"));
    return 0;
}
//-----
```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

void CSerMosSPSDlg::OnBnClickedButton2() {
    int x = GetCheckedRadioButton(IDC_RADIO1, IDC_RADIO4);
    switch(x) {
        case IDC_RADIO1:
            if(!com.openPort(1))
                MessageBox(_T("Konnte Port 1 nicht öffnen!"),
                           _T("Error"), MB_OK);
            else
                comConnected();
            break;
        case IDC_RADIO2:
            if(!com.openPort(2))
                MessageBox(_T("Konnte Port 2 nicht öffnen!"),
                           _T("Error"), MB_OK);
            else
                comConnected();
            break;
        case IDC_RADIO3:
            if(!com.openPort(3))
                MessageBox(_T("Konnte Port 3 nicht öffnen!"),
                           _T("Error"), MB_OK);
            else
                comConnected();
            break;
        case IDC_RADIO4:
            if(!com.openPort(4))
                MessageBox(_T("Konnte Port 4 nicht öffnen!"),
                           _T("Error"), MB_OK);
            else
                comConnected();
            break;
    }
}

//-----

void CSerMosSPSDlg::comConnected(void) {
    GetDlgItem(IDC_EDIT3)->SetWindowText(
        _T("Verbindung wurde\r\nhergestellt!"));
}

//-----

void CSerMosSPSDlg::OnBnClickedButton4() {
    com.closePort();
    GetDlgItem(IDC_EDIT3)->SetWindowText(
        _T("Verbindung\r\nwurde getrennt!"));
}

//-----

void CSerMosSPSDlg::OnBnClickedCheck1() {
    bool receiveEnabled = ((CButton*)GetDlgItem(IDC_CHECK1))->GetCheck() == 1;
    if(receiveEnabled)
        com.enableRecv();
    else
        com.disableRecv();
}

//-----

void CSerMosSPSDlg::OnBnClickedCheck2() {
    bool sendEnabled = ((CButton*)GetDlgItem(IDC_CHECK2))->GetCheck() == 1;
    if(sendEnabled)
        com.enableSnd();
    else
        com.disableSnd();
}

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

void CSerMosSPSDlg::OnBnClickedCheck5() {
    bool set = ((CButton*)GetDlgItem(IDC_CHECK5))->GetCheck() == 1;
    if(set) {
        GetDlgItem(IDC_EDIT9)->EnableWindow(false);
        GetDlgItem(IDC_BUTTON16)->EnableWindow(false);
        com.getStat().setMaxSndRetrys(-1);
    }
    else {
        GetDlgItem(IDC_EDIT9)->EnableWindow(true);
        GetDlgItem(IDC_BUTTON16)->EnableWindow(true);

        wchar_t buf[10];
        CEdit* edit = ((CEdit*)GetDlgItem(IDC_EDIT9));
        edit->GetWindowText(buf, 7);
        int x = _wtoi(buf);
        com.getStat().setMaxSndRetrys(x);
    }
}
//-----

void CSerMosSPSDlg::OnBnClickedButton5() {
    if(!com.send((BYTE)0x55))
        MessageBox(_T("Senden-FIFO ist voll!!!"), MB_OK);
}
//-----


void CSerMosSPSDlg::OnBnClickedButton6() {
    int val;
    if((val = com.receive()) != -1) {
        CString strTmp;
        strTmp.Format(L"%x", val);
        MessageBox(_T("Byte (Hex): 0x") + strTmp,
                   _T("Lesen aus Empfangs-Puffer"), MB_OK);
    }
    else
        MessageBox(_T("FIFO ist leer!!"), _T("Error"), MB_OK);
}
//-----


void CSerMosSPSDlg::OnBnClickedButton7() {
    int val;
    CComboBox * combo = ((CComboBox *)GetDlgItem(IDC_COMBO1));
    combo->ResetContent();
    bool isFirst = true;
    while((val = com.receive()) != -1) {
        CString strTmp;
        strTmp.Format(L"0x%x", val);
        combo->AddString(strTmp);
        if(isFirst)
            combo->SelectString(0, strTmp);
        isFirst = false;
    }
}
//-----


void CSerMosSPSDlg::OnBnClickedButton8() {
    for(int i=0;i<MAXQUEUENUM;i++) {
        com.send((BYTE)i);
    }
}
//-----
```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

void CSerMosSPSDlg::OnBnClickedButton10() {
    //CString str;
    wchar_t buf[10];
    CEdit* button = ((CEdit*)GetDlgItem(IDC_EDIT4));
    button->GetWindowText(buf, 5);
    int x = _wtoi(buf);
    if(x < 0 || x > 255)
        MessageBox(_T("Zahl nicht im gültigen Bereich!!!!"), MB_OK);
    else
        com.send((BYTE)x);
}
//-----

void CSerMosSPSDlg::OnBnClickedButton11() {
    com.emptyRecvFifo();
    OnBnClickedButton7();
}
//-----

void CSerMosSPSDlg::OnBnClickedButton9() {
    com.emptySndFifo();
}
//-----

void CSerMosSPSDlg::OnBnClickedButton12() {
    com.getStat().resetNumSndBytes();
}
//-----

void CSerMosSPSDlg::OnBnClickedButton13() {
    com.getStat().resetNumRecvBytes();
}
//-----

void CSerMosSPSDlg::OnBnClickedButton14() {
    com.getStat().resetNumSndErrs();
}
//-----

void CSerMosSPSDlg::OnBnClickedButton15() {
    com.getStat().resetNumRecvErrs();
}
//-----

void CSerMosSPSDlg::OnBnClickedButton16() {
    wchar_t buf[10];
    CEdit* edit = ((CEdit*)GetDlgItem(IDC_EDIT9));
    edit->GetWindowText(buf, 7);
    int x = _wtoi(buf);
    com.getStat().setMaxSndRetrys(x);
}
//-----
  
```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

void CSerMosSPSDlg::OnBnClickedButton17() {
    //Senden der vorgegebenen Testfolge
    unsigned short testFrame = 0;
    unsigned short bit = 0;
    //Frage die einzelnen Check-Boxen ab
    testFrame |= ((CButton*)GetDlgItem(IDC_CHECK6))->GetCheck() << bit++;
    testFrame |= ((CButton*)GetDlgItem(IDC_CHECK7))->GetCheck() << bit++;
    testFrame |= ((CButton*)GetDlgItem(IDC_CHECK8))->GetCheck() << bit++;
    testFrame |= ((CButton*)GetDlgItem(IDC_CHECK9))->GetCheck() << bit++;
    testFrame |= ((CButton*)GetDlgItem(IDC_CHECK10))->GetCheck() << bit++;
    testFrame |= ((CButton*)GetDlgItem(IDC_CHECK11))->GetCheck() << bit++;
    testFrame |= ((CButton*)GetDlgItem(IDC_CHECK12))->GetCheck() << bit++;
    testFrame |= ((CButton*)GetDlgItem(IDC_CHECK13))->GetCheck() << bit++;
    testFrame |= ((CButton*)GetDlgItem(IDC_CHECK14))->GetCheck() << bit++;
    testFrame |= ((CButton*)GetDlgItem(IDC_CHECK15))->GetCheck() << bit++;
    testFrame |= ((CButton*)GetDlgItem(IDC_CHECK16))->GetCheck() << bit++;
    testFrame |= ((CButton*)GetDlgItem(IDC_CHECK17))->GetCheck() << bit++;
    testFrame |= ((CButton*)GetDlgItem(IDC_CHECK18))->GetCheck() << bit++;
    testFrame |= ((CButton*)GetDlgItem(IDC_CHECK19))->GetCheck() << bit++;
    testFrame |= ((CButton*)GetDlgItem(IDC_CHECK20))->GetCheck() << bit++;
    testFrame |= ((CButton*)GetDlgItem(IDC_CHECK21))->GetCheck() << bit++;

    //Format von testFrame:
    //IDLE|SwapBit|...|D7|...|D0|StartBit|IDLE
    //WICHTIG: Abfragen wie in startSendByte einfügen (zumindest einige)
    if(!com.test(testFrame))
        MessageBox(_T("Konnte Test nicht starten, da noch aktiv!"), MB_OK);
    return;
}

```

Klasse : ComPort

```
#ifndef CPURT_H
#define CPURT_H "cport.h"

#include <windows.h>

//-
//-
//-
class ComPort
{
private:
    HANDLE hCom;
    BOOL CTSstat;           //Gibt den Zustand nach dem letzten Pollen an
    BOOL CTSfallingEdge;
public:
    static const int OFF = 0;
    static const int ON = 1;
    static const int MAX_PORT = 4;
    static const int STATE_ERR = -3;
    static const int OPEN_ERR = -2;
    static const int WRONG_PORT = -1;
    static const int COMM_OK = 0;
    bool Connected;

    //Konstruktor / Destruktor
    ComPort(void);
    ~ComPort(void);

    HANDLE GetHCom(void);
    int OpenCom(int portnr);
    void getPorts(CString *ports);
    void CloseCom(void);

    BOOL SetUART(DCB *dcb);
    BOOL SetUART(long baud, char bytes, char parity, char stopbit);

    BOOL SetTimeouts(COMMTIMEOUTS *timeouts);
    BOOL SetTimeouts(long interval, int multiplier, int constant);

    void SetDTR(int kz);
    void SetRTS(int kz);
    void SetTXD(int kz);
    void SetAll(int kz);
    void SetAll(int dtr, int rts, int txd);

    BOOL GetCTS(void);
    BOOL GetDSR(void);

    unsigned long Send(const char *text);
    unsigned long Receive(char *text, size_t maxsize);

    BOOL CTSPoll(void);
};

//-
//-
//-
#endif
```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

//Diese Klasse ist für den Zugriff und die Kommunikation mit
//der seriellen Schnittstelle zuständig. Hier finden sich Methoden
//zum Öffnen und Schließen von COM-Ports, sowie zur Kommunikation
//und Verwaltung.
//-----

#include "StdAfx.h"
#include "ComPort.h"

//Konstruktor
ComPort::ComPort(void) {
    hCom=INVALID_HANDLE_VALUE;
    Connected=false;
    CTSstat = OFF;
}
//-----

//Destruktor
ComPort::~ComPort(void) {
    CloseCom();
}
//-----

//liefert das Handle des Ports
HANDLE ComPort::GetHCom(void) {
    return hCom;
}
//-----

//Öffnet einen Port mit der Nummer, die übergeben wird
int ComPort::OpenCom(int portnr) {
    DCB dcb;
    BOOL res;

    //Wurde auch eine korrekte Port-Nummer übergeben???
    if(portnr<=0 || portnr>MAX_PORT)
        return(WRONG_PORT);

    //Ermittle Port
    int index = portnr - 1;

    //Prüfe auf falsche Port_Nr
    if(index<=-1 || index > ComPort::MAX_PORT)
        return -1;

    //Bezeichnungen der einzelnen Ports
    CString port[4] = {_T("COM1"), _T("COM2"), _T("COM3"), _T("COM4")};

    //Versuche auf den entsprechenden Port zuzugreifen...
    //Behandlung als Datei
    hCom = CreateFile( port[index],
                      GENERIC_READ | GENERIC_WRITE,
                      0,
                      0,
                      OPEN_EXISTING,
                      FILE_FLAG_OVERLAPPED,
                      0);

    //Falls Port nicht geöffnet werden konnte
    if(hCom == INVALID_HANDLE_VALUE)
        return(OPEN_ERR);

    //Hole den DCB-Block zum Port (Statusinformationen)
    res = GetCommState(hCom, &dcb);
    if(!res) {
        CloseCom();
        return(STATE_ERR);
    }
}

```

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

//Verbindung zum Port konnte erfolgreich hergestellt werden
Connected = true;
return(COMM_OK);
}

//-----
//Gebe eine Liste der verfügbaren Schnittstellen zurück
//Dazu wird ganz einfach versucht eine gewisse Anzahl an Ports
//zu öffnen.
//Es muss ein Array von C-Strings mit mindestens 4 Elementen übergeben
//werden!
void ComPort::getPorts(CString *ports) {
    CString port[4] = {_T("COM1"), _T("COM2"), _T("COM3"), _T("COM4")};
    for(int i=0;i<4;i++) {
        // Prüfe, ob Port geöffnet werden kann
        if(OpenCom(i+1) == COMM_OK) {
            ports[i].SetString(port[i]);
        }
        //Geöffneter Port sollte wieder geschlossen werden
        CloseCom();
    }
}
//-----

//Geöffneter Port wird wieder geschlossen
void ComPort::CloseCom() {
    CloseHandle(hCom);
    hCom = INVALID_HANDLE_VALUE;
    Connected = false;
}

//-----
//Grundeinstellungen wie Parität, baud-Zahl etc. vornehmen
BOOL ComPort::SetUART(DCB *dcb) {
    if(hCom != INVALID_HANDLE_VALUE)
        return(SetCommState(hCom, dcb));
    else
        return(false);
}

//siehe obere Methode
BOOL ComPort::SetUART(long baud, char bytes, char parity, char stopbit) {
    if(hCom!=INVALID_HANDLE_VALUE) {
        // DCB ist eine Struktur für die Einstellungen der
        // COM-Schnittstelle(siehe MSDN)
        DCB dcb;
        dcb.BaudRate=baud;
        dcb.ByteSize=bytes;
        dcb.Parity=parity;
        dcb.StopBits=stopbit;
        return(SetCommState(hCom, &dcb));
    }
    else
        return(false);
}

//-----

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

//Time-Out Parameter festlegen
BOOL ComPort::SetTimeouts(COMMTIMEOUTS *timeouts) {
    if(hCom!=INVALID_HANDLE_VALUE)
        return(SetCommTimeouts(hCom, timeouts));
    else
        return(false);
}

//-----

//Time-Out Parameter festlegen
BOOL ComPort::SetTimeouts(long interval, int multiplier, int constant) {
    //Struktur für die Time-Out-Parameter(siehe MSDN)
    if(hCom!=INVALID_HANDLE_VALUE) {
        COMMTIMEOUTS timeouts;
        timeouts.ReadIntervalTimeout=interval;
        timeouts.ReadTotalTimeoutMultiplier=multiplier;
        timeouts.ReadTotalTimeoutConstant=constant;
        return(SetCommTimeouts(hCom, &timeouts));
    }
    else
        return(false);
}

//-----

//DTR-Pin des COM-Ports direkt setzen
void ComPort::SetDTR(int kz) {
    if(hCom!=INVALID_HANDLE_VALUE) //Falls Zugriff auf den Port besteht
        if(kz==ON)
            EscapeCommFunction(hCom, SETDTR); // setzen
        else
            EscapeCommFunction(hCom, CLRDRTR); // Loeschen
}
//-----

//RTS-Pin des COM-Ports direkt setzen
void ComPort::SetRTS(int kz) {
    if(hCom!=INVALID_HANDLE_VALUE) //Falls Zugriff auf den Port besteht
        if(kz==ON)
            //EscapeCommFunction(hCom, SETRTS); // setzen
            EscapeCommFunction(hCom, CLRRTS); // setzen
        else
            //EscapeCommFunction(hCom, CLRRTS); // setzen
            EscapeCommFunction(hCom, SETRTS);
}
//-----

//TXD-Pin des COM-Ports direkt setzen
void ComPort::SetTXD(int kz) {
    if(hCom!=INVALID_HANDLE_VALUE) //Falls Zugriff auf den Port besteht
        if(kz==ON)
            EscapeCommFunction(hCom, CLRBREAK); // Loeschen
        else
            EscapeCommFunction(hCom, SETBREAK); // setzen
}
//-----

//DTR, RTS und TXD direkt auf den gleichen Pegel setzen
void ComPort::SetAll(int kz) {
    SetAll(kz, kz, kz);
}
//-----

```

```

//DTR, RTS und TXD setzen
void ComPort::SetAll(int dtr, int rts, int txd) {
    SetDTR(dtr);
    SetRTS(rts);
    SetTXD(txd);
}
//-----

//Eingang CTS lesen
BOOL ComPort::GetCTS(void) {
    if(hCom!=INVALID_HANDLE_VALUE) {
        DWORD COMStatus;

        GetCommModemStatus(hCom, &COMStatus);
        //Falls ein LOW am Eingang liegt(das Bit für CTS ist nicht gesetzt)
        //Bei der Kommunikation mit dem MOSES ist dies genau invertiert,
        //daher wird auf HIGH angefragt
        if(COMStatus & MS_CTS_ON)
            return OFF;
        return ON;
    }
    else
        return OFF;
}
//-----

//Eingang DSR lesen
//Wird nicht für die Kommunikation mit dem MOSES benötigt, ansonsten müssen
//die Abfragen auch invertiert werden s.o.
BOOL ComPort::GetDSR(void) {
    if(hCom!=INVALID_HANDLE_VALUE) {
        DWORD COMStatus;

        GetCommModemStatus(hCom, &COMStatus);
        //Falls ein HIGH am Eingang liegt(das Bit für DSR ist gesetzt)
        if(COMStatus & MS_DSR_ON)
            return ON;
        return OFF;
    }
    else
        return OFF;
}
//-----

//Daten über die Schnittstelle versenden
unsigned long ComPort::Send(const char *text) {
    //Breche ab, wenn Port nicht geöffnet
    if(hCom!=INVALID_HANDLE_VALUE) {
        unsigned long sent;

        //Schreibe auf Port
        WriteFile(hCom, text, strlen(text), &sent, NULL);
        return(sent);
    }
    else
        return(0);
}
//-----

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

//Daten über die Schnittstelle empfangen
unsigned long ComPort::Receive(char *text, size_t maxsize) {
    //Breche ab, wenn Port nicht geöffnet
    if(hCom!=INVALID_HANDLE_VALUE) {
        unsigned long received;

        ReadFile(hCom, text, maxsize, &received, NULL);
        text[received] = 0;

        return(received);
    }
    else
        return(0);
}
//-----

// Da eine echte Flankensteuerung des CTS-Pins ohne weiteres nicht möglich
// ist, muss diese Software-technisch realisiert werden. Dazu muss in
// gewissen Zeitabständen (ca. 2ms) der Zustand des Pins geprüft werden.
// Jedoch prüft diese Methode nur auf fallende Flanken. Bei einem
// Abtastintervall von T muss eine durchschnittliche Abweichung von T/2
// beachtet werden. In gewissen Fällen kann die CPU-Auslastung nach oben
// schnellen. Die Verwendung dieser Methode sollte daher nur dann
// stattfinden, wenn dies wirklich nötig ist
BOOL ComPort::CTSPoll(void) {
    BOOL tmpCTS = GetCTS();
    CTSfallingEdge = CTSstat && !tmpCTS;
    CTSstat = tmpCTS; //Neuer Zustand der Leitung CTS
    return CTSfallingEdge;
}
//-----

```

Klasse: ComThread

```

#pragma once
#include "stdafx.h"
#include <windows.h>
#include "ComPort.h"
#include <queue>

using namespace std;

class Stat;
//-----
//-----
//-----
class ComThread : public CWinThread {

private:
    bool recvEnabled;
    bool sndEnabled;
    bool sndActive;
    bool recvActive;
    short dataBuf;

    //TestFrame übertragen
    bool testEnabled;
    BYTE numBits;
    BYTE errStat;
    BYTE sendACK;

    //Zeiger auf Sende- bzw. Empfangs-FIFO
    queue<BYTE>* recvFifo;
    queue<BYTE>* sndFifo;

    //Senden im Fehlerfall wiederholen
    unsigned int retrySnd; //Zähler der Wiederholungen
    unsigned int maxSndRetrys; //max. Zahl an Wiederholungen (-1 = unendlich)

    //Zeiger auf die Com-Schnittstelle
    ComPort *cPort;

    //Statistik-Elemente
    unsigned int numSndBytes;
    unsigned int numRecvBytes;
    unsigned int numSndErrs;
    unsigned int numRecvErrs;
    int lastSndByte;
    int lastRecvByte;
    Stat *stat;

    //Nur bei Benutzen des Testframes verändert
    short response;
    unsigned short testFrame;

    //Allgemeine Konstanten
    //Wartezeit, falls Senden und Empfang nicht erlaubt
    static const unsigned int PAUSETIME = 1000;

    //Für RTS- / CTS-Leitungen
    static const bool OFF = false;
    static const bool ON = true;
    static const unsigned int POLLTIME = 2;

    //Konstanten betreffend numBits
    //Sender:
    static const BYTE WAITBIT = 11;
    static const BYTE FINALBIT = 12;
    static const BYTE SWAPBIT = 13;

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

//Empfänger:
static const BYTE STARTBIT = 0;
static const BYTE STOPBIT = 10;
static const BYTE ACKBIT = 11;
static const BYTE ENDBIT = 12;

//Fehler-Konstanten für errStat
//Sender:
static const BYTE NOERR = 0;
static const BYTE ERRLINEZERO = 1;
static const BYTE ERRACK = 2;

//Empfänger:
static const BYTE ERRWRONGSTOP = 3;
static const BYTE ERRWRONGPARITY = 4;
static const BYTE ERRWRONGSTART = 5;
static const BYTE ERRRECVFIFOFULL = 6;

//Fehler-Strings;
static const wchar_t *ERRSTATSTR[];

//Strings max. 22 Zeichen

//Methoden
bool sleep(unsigned int msec);
void tryReceive();
void resetRecvData();
bool receive();
void trySend();
bool startSendByte(BYTE val);
bool send(void);
void sendTestFrame();
short generateFrame(BYTE val);
bool parity(BYTE x);
//-----

public:
DECLARE_DYNCREATE(ComThread)
ComThread();
~ComThread();

virtual BOOL InitInstance();
virtual int ExitInstance();
virtual int Run();

void setQueuePtr(queue<BYTE> *sndFifo, queue<BYTE> *recvFifo);
void setComPort(ComPort* cPort);
void setRecvEn(bool enabled);
void setSndEn(bool enabled);
bool isSendActive(void);
bool isReceiveActive(void);
Stat getStat();
void resetAllStats();
void resetNumSndBytes();
void resetNumRecvBytes();
void resetNumSndErrs();
void resetNumRecvErrs();
void setMaxSndRetrys(int trys);
void startTest(unsigned short tFrame);
bool isTestActive();
//-----

protected:
};

//-----
//-----
//-----

```

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

// Diese Klasse ist von CWinThread abgeleitet. Beim Erzeugen einer Instanz
// dieser Klasse wird also ein neuer Thread gestartet. Die Kommunikation mit
// dem MOSES läuft also parallel zur eigentlichen Anwendung ab, wodurch der
// der Anwender auch zeit-intensive Programme schreiben kann, ohne dass
// dadurch zeitliche Probleme bei der Übertragung auftreten.
// Da ein Thread den selben Adressraum wie der eigentliche Vaterprozess
// besitzt, können sehr leicht gemeinsam genutzte Betriebsmittel erzeugt
// werden. In diesem Fall sind dies die Sende- und Empfangs-FIFOs.
// Bei der Verwendung von Threads muss besonders bei Operationen auf
// gemeinsam genutzte Betriebsmittel darauf geachtet werden, dass bei
// gleichzeitigem Zugriff Inkonsistenzen auftreten können. In solchen Fällen
// sollten Semaphore oder ähnliche Konstrukte verwendet werden, um den
// exklusiven Zugriff zu gewährleisten. Obwohl uns keine Probleme im
// Zusammenhang mit den FIFOs aufgefallen sind, so sind diese doch
// theoretisch möglich. Ggf. müssen die Semaphore also nachträglich
// eingeführt werden.
//-----

#include "StdAfx.h"
#include "ComThread.h"
#include "Com.h"
#include "Stat.h"
#include <windows.h>
#include <queue>

using namespace std;

//Fehlermeldungen für die Kommunikation
const wchar_t* ComThread::ERRSTATSTR[] = {_T("NO ERROR"),
                                            _T("SND CTS NOT ZERO"),
                                            _T("SND NEGATIVE ACK"),
                                            _T("RCV WRONG STOPBIT"),
                                            _T("RCV WRONG PARITY"),
                                            _T("RCV WRONG STARTBIT"),
                                            _T("RCV FIFO FULL")};

IMPLEMENT_DYNCREATE(ComThread, CWinThread)

//Konstruktor
ComThread::ComThread() {
    //Pointer-Initialisierungen
    sndFifo = NULL; //Sende-FIFO
    recvFifo = NULL; //Empfangs-FIFO
    cPort = NULL; //Serielle Schnittstelle

    //Initialisierungen für Sendewiederholungen
    //infiniteSndTrys = false; //Erlaube unendlich viele Wiederholungen
    retrySnd = 0; //Zähler der Wiederholungen
    maxSndRetrys = -1; //max. Zahl an Wiederholungen

    //Status-Informationen
    sndActive = false;
    recvActive = false;
    recvEnabled = false;
    sndEnabled = false;
    testEnabled = false;
    dataBuf = 0;

    //Sende- bzw. Empfangsdaten
    numBits = 0;
    errStat = 0;
    sendACK = 0;

    //Statistikinformationen
    stat = new Stat(this);
}

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

//weitere Statusinformationen
numSndBytes = 0;
numRecvBytes = 0;
numSndErrs = 0;
numRecvErrs = 0;
lastSndByte = 0;
lastRecvByte = 0;
response = 0;
}

//-----
//Destruktor

ComThread::~ComThread() {
    //Dynamisch erzeugte Objekte entfernen
    delete(stat);
}
//-----

//Methode der Klasse CWinThread muss überschrieben werden
BOOL ComThread::InitInstance() {
    return TRUE;
}
//-----

//Methode der Klasse CWinThread muss überschrieben werden
int ComThread::ExitInstance() {

    return 0;
}
//-----
```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

// Methode der Klasse CWinThread wird überschrieben. Wird nach dem Start des
// Threads ausgeführt...
int ComThread::Run() {

    //Es muss ein ComPort-Objekt angelegt sein
    ASSERT(cPort != NULL);

    //Sende-FIFO-Objekt muss bereits erzeugt sein
    ASSERT(sndFifo != NULL);

    //Empfangs-FIFO-Objekt muss bereits erzeugt sein
    ASSERT(recvFifo != NULL);

    // Diese Schleife wird über die gesamte Lebensdauer des Threads
    // abgearbeitet
    while(1) {

        //Prüfe, ob Verbindung zu einem COM-Port hergestellt wurde.
        //Weitere Operationen machen erst dann einen Sinn.
        if(cPort->Connected) {
            //Prüfe, ob der Empfang erlaubt ist...
            if(recvEnabled)
                tryReceive();

            //Prüfe, ob die Sendefreigabe gegeben ist
            if(sndEnabled)
                trySend();

            // Ansonsten wird der Thread eine gewisse Zeit schlafen gelegt
            // damit keine CPU-Zeit verschwendet wird
            else if(!sndEnabled && !recvEnabled)
                Sleep(PAUSETIME); //Senden und Empfang sind nicht erlaubt
        }
        else
            // Legt Thread schlafen
            Sleep(PAUSETIME);
    }
    return 0;
}
//-----

// Es muss bekannt sein, aus welchen Sende-FIFO gelesen und in welchen
// Empfangs-FIFO geschrieben werden muss. Diese Methode muss daher durch
// das entsprechende Com-Objekt aufgerufen werden
void ComThread::setQueuePtr(queue<BYTE> *sndFifo, queue<BYTE> *recvFifo) {
    this->sndFifo = sndFifo;
    this->recvFifo = recvFifo;
}
//-----

// Es muss bekannt sein, welcher COM-Port zur Kommunikation verwendet wird
void ComThread::setComPort(ComPort* cPort) {
    this->cPort = cPort;
}
//-----

// Setzen der Empfangsfreigabe
void ComThread::setRecvEn(bool enabled) {
    recvEnabled = enabled;
}
//-----
  
```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

//Setzen der Sendefreigabe
void ComThread::setSndEn(bool enabled) {
    sndEnabled = enabled;
}
//-----

// Ggf. einen Empfangsvorgang einleiten
void ComThread::tryReceive() {
    //10-Zeitintervalle lang Leitung auf fallende Flanke prüfen
    for(int i=0;i<10;i++) {
        if(cPort->CTSPoll()) { //Fallende Flanke erkannt??
            recvActive = true;

            //Für Empfang benötigte Daten zurücksetzen
            resetRecvData();

            //In den Empfangsmodus wechseln
            cPort->SetRTS(ComPort::OFF);

            //Zunächst halbe Periodendauer warten
            sleep(Com::TIMEINTERVALL / 2);

            //Empfang durchführen
            while(!receive())
                sleep(Com::TIMEINTERVALL);
            cPort->CTSPoll();
            break; //Beende Methode nach Empfang
        }
        sleep(POLLTIME);
    }
}

//-----

// Setze die für einen Empfangsvorgang nötigen Informationen zurück
void ComThread::resetRecvData() {
    numBits = 0;
    errStat = 0;
    sendACK = 0;
    errStat = NOERR;
}

//-----

// Methode zum bitweisen Empfang der Daten
bool ComThread::receive() {
    if(numBits == STARTBIT) {
        //Wechsle in den Empfangsmodus (TxD = 0)
        cPort->SetRTS(ComPort::OFF);
    }
    if(numBits == ACKBIT) {
        //Schreibe den Erfolg der Operation auf die Leitung
        if(errStat == NOERR)
            cPort->SetRTS(ComPort::ON);
        else
            numRecvErrs++;
        numBits++;
        return false;
    }
    else if(numBits == ENDBIT) {
        //In den Zustand Idle wechseln
        cPort->SetRTS(ComPort::ON);
        recvActive = false;
        return true; //Übertragung ist abgeschlossen
    }
}

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

//Lese nächstes Bit von der Leitung
//Vorsicht: Das Datenbyte ist zunächst falsch herum
dataBuf <= 1;
dataBuf |= (cPort->GetCTS() ? 1 : 0);

//Nach dem Lesen des StopBits muss Parität berechnet werden,
//sowie Start- bzw. StopBit überprüft werden
if(numBits == STOPBIT) {
    //Label6->Caption = dataBuf;
    //Prüfe StopBit
    if(!(dataBuf & 1))
        errStat = ERRWRONGSTOP;

    dataBuf >>= 1;

    //Paritäts-Bit lesen
    bool parityBit = (dataBuf & 1) != 0;
    dataBuf >>= 1;

    BYTE dataByte = 0;

    //DatenBits in die richtige Reihenfolge bringen
    for(short i=0;i<8;i++, dataBuf >>= 1, dataByte <<= 1)
        if(dataBuf & 1)
            dataByte |= 1;
    //Datenbyte wurde eins zuviel geshiftet
    dataByte >>= 1;

    //Parität über die DatenBits ermitteln
    if(parityBit != parity(dataByte))
        errStat = ERRWRONGPARITY;

    //Prüfe StartBit
    if(dataBuf & 1)
        errStat = ERRWRONGSTART;

    //Trage in die FIFO, wenn kein Fehler ermittelt wurde
    if(errStat == NOERR)
        //Jedoch muss noch Platz in der FIFO sein
        if(recvFifo->size() < Com:::FIFOMAXSIZE) {
            recvFifo->push(dataByte);
            numRecvBytes++;
            lastRecvByte = dataByte;
        }
        else
            //Fehler: Empfangs-FIFO ist voll
            errStat = ERRRECVFIFOFULL;
    }
    numBits++; //Inkrementiere Anzahl der übertragenen Bits
    return false; //Übertragung ist noch nicht abgeschlossen
}
//-----

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

// Versuche ein Datenbyte zu übertragen
void ComThread::trySend() {
    if(!sndFifo->empty()) { //Prüfe, ob Wert in der Fifo
        BYTE sndByte = sndFifo->front();

        //Kann ein Sendevorgang gestartet werden?
        if(startSendByte(sndByte)) {
            //Führe die bitweise Übertragung durch
            do
                sleep(Com::TIMEINTERVALL);
            while(!send());

            //Prüfe, ob Fehler während des Empfangs aufgetreten ist
            if(errStat != NOERR) {
                retrySnd++;
                numSndErrs++;
            }
            else {
                // Kein Fehler
                retrySnd = 0;
                numSndBytes++;
                // Setze das zuletzt übertragene Byte auf neuen Wert
                lastSndByte = sndByte;

                // Datenbyte kann der FIFO entnommen werden
                sndFifo->pop();
            }
            if(retrySnd > maxSndRetrys) {
                // Max. Anzahl an Sendever suchen wurde überschritten
                if(maxSndRetrys != -1)
                    // Programmausführung wird unterbrochen
                    ASSERT(retrySnd <= maxSndRetrys); //Fehler melden
                retrySnd = 0;
            }
            if(!recvEnabled)
                sleep(Com::TIMEINTERVALL);
        }
    }
    else if(testEnabled) {
        sendTestFrame();
    }
    //Lege Thread schlafen, wenn Empfang nicht erlaubt und SndFIFO leer
    else if(!recvEnabled)
        sleep(PAUSETIME);
}
//-----

// Versuche einen Sendevorgang zu initiieren
bool ComThread::startSendByte(BYTE val) {
    bool started = false;

    //Die serielle Schnittstelle muss geöffnet sein
    //Es darf kein Sendevorgang aktiv sein
    //Es darf kein Empfangsvorgang aktiv sein
    //Empfänger muss seine RTS-Leitung im Zustand Idle haben
    if(cPort->Connected && !sndActive && !recvActive
        && cPort->GetCTS() == ComPort::ON) {
        sndActive = started = true;
        dataBuf = generateFrame(val); //Erzeuge Frame
        if(dataBuf & 1) //Setze RTS-Leitung (Regelfall: 0)
            cPort->SetRTS(ComPort::ON);
        else
            cPort->SetRTS(ComPort::OFF);
        dataBuf >>= 1; //Entferne Start-Bit
        numBits = 1; //Startbit liegt ja bereits auf der Leitung
    }
    return started; //Wurde eine Übertragung gestartet?
}
//-----
  
```

```

// Sende ein Datenbit
bool ComThread::send(void) {
    //Prüfe, ob Übertragung abgeschlossen ist und lese Erfolg der Operation
    // ggfs. ein
    if(numBits == FINALBIT) {
        sendACK = cPort->GetCTS(); // Lese ACK / NACK
        numBits++;
        return false;
    }

    //Prüfe, ob Leitung des Empfängers auf NULL
    if(numBits <= WAITBIT && cPort->GetCTS() == ComPort::ON) {
        //Wieder in den Zustand Idle wechseln
        cPort->SetRTS(ComPort::ON);

        //Fehler-Flag setzen
        errStat = ERRLINEZERO;
        sndActive = false;
        return true;
    }
    if(numBits == WAITBIT) {
        //Nach dem Stop-Bit in den Zustand Null wechseln
        cPort->SetRTS(ComPort::OFF);
        numBits++;
        return false; //Sofort abbrechen
    }
    if(numBits == SWAPBIT) {
        cPort->CTSPoll(); //Zunächst alle Flanken löschen

        //Wieder in den Zustand Idle wechseln
        cPort->SetRTS(ComPort::ON);
        errStat = sendACK ? NOERR : ERRACK;
        sndActive = false;
        return true; //Übertragung ist abgeschlossen
    }

    //Schreibe nächstes Bit auf die Leitung
    const int nextBit = ((dataBuf & 1) ? ComPort::ON : ComPort::OFF);
    cPort->SetRTS(nextBit);
    dataBuf >>= 1; //Entferne dieses Bit aus dem Frame
    numBits++; //Inkrementiere Anzahl der übertragenen Bits
    return false; //Übertragung ist noch nicht abgeschlossen
}
//-----

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

void ComThread::sendTestFrame() {
    unsigned short tframe = testFrame; //Wert

    //Die serielle Schnittstelle muss geöffnet sein
    //Es darf kein Sendevorgang aktiv sein
    //Es darf kein Empfangsvorgang aktiv sein
    if(cPort->Connected && !sndActive && !recvActive) {
        //Verhalten des Empfängers mit response aufzeichnen
        response = 0;

        //Wechsle in den Zustand IDLE1
        //Dazu: Leitung auf Wert setzen und ca. 300ms pausieren
        //Dadurch werden direkte Reaktionen des Empfängers auf
        //Änderungen vermieden
        cPort->SetRTS(tframe & 1 ? ComPort::ON : ComPort::OFF);
        sleep(Com::TIMEINTERVALL * (SWAPBIT + 2));

        //Schreibe verbleibenden 15-Bits auf die Leitung
        for(short i=0;i<15;i++) {
            //zunächst Empfangsleitung einlesen
            response |= (cPort->GetCTS() == ComPort::ON ? 1 : 0);
            response <= 1;

            //Schreibe Bit auf RTS-Leitung
            cPort->SetRTS((tframe >>= 1) & 1 ? ComPort::ON : ComPort::OFF);
            sleep(Com::TIMEINTERVALL);
        }

        //Eins zu weit geshiftet, dies rückgängig machen
        response >>= 1;
        //Nach ca. 300ms wieder in den normal-Betrieb übergehen
        sleep(Com::TIMEINTERVALL * (SWAPBIT + 2));
        cPort->SetRTS(ComPort::ON);
    }
    testEnabled = false;
}

//-----
void ComThread::startTest(unsigned short tFrame) {
    testFrame = tFrame;
    testEnabled = true;
}

//-----
bool ComThread::isTestActive() {
    return testEnabled;
}

//-----
// Erzeuge den Datenframe
short ComThread::generateFrame(BYTE val) {
    short frame = val; //LSB werden benutzt
    frame <<= 1; //Start-Bit wird hinzugefügt
    frame |= 0x400; //Stop-Bit wird hinzugefügt
    frame |= (parity(val) << 9); //Parität
    return frame;
}

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

// Berechne die ungerade Parität zu einem Datenbyte
bool ComThread::parity(BYTE x) {
    BYTE count = 0;
    while(x) { //Anzahl der gesetzten Bits bestimmen
        count++;
        x &= (x-1);
    }
    return (count & 1); //Parität bestimmen
}
//-----

// Prüfe, ob ein Sendevorgang zur Zeit aktiv ist
bool ComThread::isSendActive(void) {
    return sndActive;
}
//-----

// Prüfe, ob ein Empfangsvorgang zur Zeit aktiv ist
bool ComThread::isReceiveActive(void) {
    return recvActive;
}
//-----

// Besorge alle möglichen Statusinformationen und gebe diese zurück
Stat ComThread::getStat() {
    stat->isSndActive = sndActive;
    stat->isRecvActive = recvActive;
    stat->numSndBytes = numSndBytes;
    stat->numRecvBytes = numRecvBytes;
    stat->lastErrStat = errStat;
    stat->lastErrStatStr.Format(L"%s", ERRSTATSTR[errStat]);
    stat->numSndErrs = numSndErrs;
    stat->numRecvErrs = numRecvErrs;
    stat->lastSndByte = lastSndByte;
    stat->lastRecvByte = lastRecvByte;
    stat->response = response;
    return *stat;
}
//-----

// Setze alle Statistik-Informationen zurück
void ComThread::resetAllStats() {
    resetNumSndBytes();
    resetNumRecvBytes();
    resetNumSndErrs();
    resetNumRecvErrs();
}
//-----

// Setze die Anzahl der übertragenen Bytes zurück
void ComThread::resetNumSndBytes() {
    numSndBytes = 0;
}
//-----

// Setze die Anzahl der empfangenen Datenbytes zurück
void ComThread::resetNumRecvBytes() {
    numRecvBytes = 0;
}
//-----
  
```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

// Setze die Anzahl der Sendefehler zurück
void ComThread::resetNumSndErrs() {
    numSndErrs = 0;
}
//-----

// Setze die Anzahl der Empfangsfehler zurück
void ComThread::resetNumRecvErrs() {
    numRecvErrs = 0;
}
//-----

// Setze die max. Anzahl an Sendever suchen
void ComThread::setMaxSndRetrys(int trys) {
    maxSndRetrys = trys;
}
//-----

//Dieser Timer wird vermutlich nicht benötigt
VOID CALLBACK _timerStopped(PVOID lpParam, BOOLEAN TimerOrWaitFired) {
    //if (lpParam == NULL) {
        //Sollte nicht auftreten
    //}
    SetEvent((HANDLE)lpParam);
}

//-----

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

// Thread wird kurzzeitig pausiert.
bool ComThread::sleep(unsigned int msec) {
    /*HANDLE hTimer = NULL;
    HANDLE hTimerQueue = NULL;

    //Erstelle Timer-Queue
    hTimerQueue = CreateTimerQueue();
    if (NULL == hTimerQueue) {
        return false;
    }

    //Erzeuge ein Event
    HANDLE gDoneEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
    if (NULL == gDoneEvent) {
        return false;
    }

    //Timer mit der gewünschten Zeit einstellen und starten
    if (!CreateTimerQueueTimer( &hTimer, hTimerQueue,
        &_timerStopped, gDoneEvent, msec, 0, 0)) {
        return false;
    }

    // Warte darauf, dass timer-queue thread beendet
    //mithilfe des Event-Objektes
    // Der Thread wird dies nach der vorgegeben Zeit vornehmen
    if (WaitForSingleObject(gDoneEvent, INFINITE) != WAIT_OBJECT_0)
        return false;

    CloseHandle(gDoneEvent);

    // Lösche Timer in der Queue
    if (!DeleteTimerQueue(hTimerQueue))
        return false;
    return true;*/

    /*Dieser Teil ist zwar sehr genau, da
    er aber in einer Schleife läuft wird viel zu viel unnötige Prozessor-
    Zeit verschwendet!!!*/
    __int64 end_count, start_count;
    __int64 freq;
    float time;

    //Frequenz ermitteln
    QueryPerformanceFrequency((LARGE_INTEGER*)&freq);

    // Startzeitpunkt ermitteln
    QueryPerformanceCounter((LARGE_INTEGER*)&start_count);

    // Warte solange, bis Zeit abgelaufen ist
    do {
        QueryPerformanceCounter((LARGE_INTEGER*)&end_count);
        time = (float)(end_count - start_count) / (float)freq;
    }
    while(time < (float)msec / 1000);
    return true;
}
//-----

```

Klasse: Stat

```
#pragma once

#include "stdafx.h"
#include "ComThread.h"

//-----
//-----
//-
class Stat
{
private:
    ComThread *pThread;
public:
    bool isSndActive;
    bool isRecvActive;
    int numSndBytes;
    int numRecvBytes;
    int lastErrStat;
    CString lastErrStatStr;
    int numSndErrs;
    int numRecvErrs;
    int lastSndByte;
    int lastRecvByte;
    short response;
    int sizeRecvFifo;
    int sizeSndFifo;

    //Methoden
    Stat(ComThread *pThread);
    ~Stat(void);
    void resetNumSndBytes();
    void resetNumRecvBytes();
    void resetNumSndErrs();
    void resetNumRecvErrs();
    void resetAllStats();
    void setMaxSndRetrys(int trys);
};

//-----
//-
//-
```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

// Diese Klasse ist zur Verwaltung des Status-Informationen der Kommunikation
// zuständig. Der Anwender kann sich ein Status-Objekt geben lassen und
// anschließend entsprechend auswerten

#include "StdAfx.h"
#include "Stat.h"

// Konstruktor
Stat::Stat(ComThread *pThread) {
    //Thread-Objekt, dass die eigentliche Kommunikation mit dem MOSES
    //übernimmt, muss übergeben werden
    this->pThread = pThread;
}
//-----

// Destruktor
Stat::~Stat(void) {
}
//-----


// Setze alle Status-Informationen zurück
void Stat::resetAllStats() {
    pThread->resetAllStats();
}
//-----


// Setze Anzahl der gesendeten Bytes zurück
void Stat::resetNumSndBytes() {
    pThread->resetNumSndBytes();
}
//-----


// Setze Anzahl der empfangenen Bytes zurück
void Stat::resetNumRecvBytes() {
    pThread->resetNumRecvBytes();
}
//-----


// Setze Anzahl der Sende-Fehler zurück
void Stat::resetNumSndErrs() {
    pThread->resetNumSndErrs();
}
//-----


// Setze Anzahl der Empfangs-Fehler zurück
void Stat::resetNumRecvErrs() {
    pThread->resetNumRecvErrs();
}
//-----


// Setze max. Anzahl Sende-Versuche im Fehlerfall
void Stat::setMaxSndRetrys(int trys) {
    pThread->setMaxSndRetrys(trys);
}
//-----
```

Klasse: TestTimer

```
#include "stdafx.h"
#include <windows.h>

#pragma once

//-----
//-----
//-
class TestTimer : public CWinThread
{
private:
    __int64 startTime;
    unsigned int countTestIntervall;
    unsigned int maxTestIntervalls;
    float diffTestSum;
    float avgDiff;
    unsigned int intervallTime;
    bool timerTestActive;
    HWND parent;

    int lastErr;

    //Methoden
    void newTest(unsigned int numTestIntervalls,
                 unsigned int intervallTime);
    bool regNextTime();
    bool active(void);
    void resetAll();
    void startThreadTimer(unsigned int time);
    void testStdTimer();

public:
    DECLARE_DYNCREATE(TestTimer)
    TestTimer(void);
    ~TestTimer(void);

    virtual BOOL InitInstance();
    virtual int ExitInstance();

    float getAvgDiff(void);
    bool startTimerTest(int testType, HWND parent,
                        unsigned int numTestIntervalls, unsigned int intervallTime);

    //Konstanten
    static const int THREADTIMERTEST = 1;
    static const int STDTIMERTEST = 2;

    //Fehler-Konstanten
    static const int CREATETIMERQUEUE = 1;
    static const int CREATEEVENT = 2;
    static const int CREATETIMERQUEUEUTIMER = 3;

protected:
    void OnThreadMsg(WPARAM idEvent, LPARAM param);
    DECLARE_MESSAGE_MAP()
};

//-----
//-----
//-
```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

#include "StdAfx.h"
#include "TestTimer.h"
#include "ThreadTimer.h"
#include <windows.h>
#include <math.h>

//using namespace std;

IMPLEMENT_DYNAMIC(TestTimer, CWinThread)

TestTimer::TestTimer(void) {
    timerTestActive = false;
    parent = NULL;
}
//-----

TestTimer::~TestTimer(void) {
}
//-----


BEGIN_MESSAGE_MAP(TestTimer, CWinThread)
    ON_THREAD_MESSAGE(WM_THREAD_TERMINATED, OnThreadMsg)
    ON_THREAD_MESSAGE(WM_THREAD_START_STDTIMERTEST, OnThreadMsg)
END_MESSAGE_MAP()
//-----


BOOL TestTimer::InitInstance() {
    return TRUE;
}
//-----


int TestTimer::ExitInstance() {
    return 0;
}
//-----


void TestTimer::OnThreadMsg(WPARAM idEvent, LPARAM param) {
    switch(idEvent) {
        case WM_THREAD_START_STDTIMERTEST:
            testStdTimer();
            break;
        case WM_THREAD_TERMINATED:
            if(active()) {
                //Prüfe, ob alle Tests durchgelaufen sind
                if(regNextTime()) {
                    ::SendMessage(parent, WM_TESTTIMER_FINISHED,
                                WM_TESTTIMER_FINISHED, THREADTIMERTEST);
                    timerTestActive = false;
                }
            } else {
                //Starte neuen Thread
                startThreadTimer((unsigned int)intervallTime);

            }
        }
        break;
    default:
        break;
    }
}
//-----
  
```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

VOID CALLBACK TimerRoutine(PVOID lpParam, BOOLEAN TimerOrWaitFired) {
    if (lpParam == NULL) {
        ;//TODO: Exception
    }
    SetEvent((HANDLE)lpParam);
}

//-----
void TestTimer::testStdTimer() {
    HANDLE hTimer = NULL;
    HANDLE hTimerQueue = NULL;

    //Erstelle Timer-Queue
    hTimerQueue = CreateTimerQueue();
    if (NULL == hTimerQueue) {
        lastErr = CREATETIMERQUEUE;
        return;
    }

    //Führe x Timer-Tests durch
    do {
        //Erzeuge ein Event
        HANDLE gDoneEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
        if (NULL == gDoneEvent) {
            lastErr = CREATEEVENT;
            return;
        }

        //Timer mit der gewünschten Zeit einstellen und starten
        if (!CreateTimerQueueTimer(&hTimer, hTimerQueue,
        &TimerRoutine, gDoneEvent, intervalTime, 0, 0)) {
            lastErr = CREATETIMERQUEUEUTIMER;
            return;
        }

        // Warte darauf, dass timer-queue
        //thread beendet mithilfe des Event-Objektes
        // Der Thread wird dies nach der vorgegeben Zeit vornehmen
        if (WaitForSingleObject(gDoneEvent, INFINITE) != WAIT_OBJECT_0)
            lastErr = lastErr;//TODO: Exception

        CloseHandle(gDoneEvent);
        //Sleep(20);
    }
    while(!regNextTime());

    //Benachrichtige Haupt-Anwendung
    ::SendMessage(parent, WM_TESTTIMER_FINISHED, WM_TESTTIMER_FINISHED,
    STDTIMERTEST);
    timerTestActive = false;
    // Lösche alle Timer in der Queue
    if (!DeleteTimerQueue(hTimerQueue))
        lastErr = lastErr;//TODO: Exception
}
//-----
  
```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

bool TestTimer::startTimerTest(int testType, HWND parent,
                               unsigned int numTestIntervalls, unsigned int
intervalTime) {
    if(!active()) {
        resetAll();
        this->parent = parent;
        newTest(numTestIntervalls, intervalTime);

        if(testType == THREADMERTEST) //ThreadTimer
            startThreadTimer(intervalTime);
        else if(testType == STDTIMERTEST) //Standard-Timer
            //testStdTimer();
            this->PostThreadMessage(WM_THREAD_START_STDTIMERTEST,
                                     WM_THREAD_START_STDTIMERTEST, NULL);
        else
            return false; //Fehler: falscher Parameter
        return true;
    }
    return false; //Test ist noch aktiv
}
//-----



void TestTimer::newTest(unsigned int numTestIntervalls,
                       unsigned int intervalTime) {

    //Anzahl der Durchläufe
    maxTestIntervalls = numTestIntervalls;

    //Anzahl der Durchläufe zurücksetzen
    countTestInterval = 0;

    //Summe der Abweichungen zurücksetzen
    diffTestSum = 0.0;

    //Durchschnittliche Abweichung zurücksetzen
    avgDiff = 0.0;

    //Dauer eines Zeitintervalls
    this->intervalTime = intervalTime;

    //Speichere Startzeit
    QueryPerformanceCounter((LARGE_INTEGER*)&startTime);

    //Test ist jetzt aktiv
    timerTestActive = true;
}
//-----
  
```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

bool TestTimer::regNextTime() {
    __int64 end_count;
    __int64 freq;
    float time;

    // Get the frequency and save it, it shouldn't change
    QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
    QueryPerformanceCounter((LARGE_INTEGER*)&end_count);

    //Zeitdifferenz berechnen
    time = (float)(end_count - startTime) / (float)freq;

    //Addiere Differenz zur Summe
    diffTestSum += fabs((float)(time - (float)intervallTime / 1000));

    //Durchläufe hochzählen
    if(++countTestIntervall >= maxTestIntervalls) {
        avgDiff = (float)1000.0 * diffTestSum / (float) countTestIntervall;
        timerTestActive = false; //Test ist abgeschlossen
        return true; //Alle Durchläufe abgeschlossen
    }
    else
        QueryPerformanceCounter((LARGE_INTEGER*)&startTime);
    return false;
}
//-----


float TestTimer::getAvgDiff(void) {
    return avgDiff;
}
//-----


bool TestTimer::active(void) {
    return timerTestActive;
}
//-----


void TestTimer::resetAll() {
    startTime = 0;
    countTestIntervall = 0;
    maxTestIntervalls = 0;
    diffTestSum = 0.0;
    avgDiff = 0.0;
    intervallTime = 0;
    timerTestActive = false;
}
//-----


void TestTimer::startThreadTimer(unsigned int time) {
    ThreadTimer* adapter = new ThreadTimer();
    //adapter->SethWnd(AfxGetMainWnd()->m_hWnd);
    adapter->CreateThread(CREATE_SUSPENDED);
    adapter->m_bAutoDelete = true;
    //adapter->sName = _T("Test");
    adapter->setParent(this);

    //Für QueryPerformanceTimer benötigt
    //float sec = (float)time / (float)1000.0;
    //adapter->setWaitTime(sec);
    adapter->setWaitTime((float)time);
    adapter->ResumeThread();
}
//-----
```

Klasse: Com

```

#include "ComPort.h"
#include "ComThread.h"
#include "Stat.h"
#include <queue>

#pragma once

using namespace std;

//-----
//-----
//-
class Com
{
private:
    ComPort cPort;
    queue<BYTE> recvFifo;
    queue<BYTE> sndFifo;

    ComThread *pThread;

public:
    Com(void);
    ~Com(void);
    void reset();
    void emptyRecvFifo(void);
    void emptySndFifo(void);
    void getPorts(CString *ports);
    bool openPort(int num);
    void closePort();
    void disableRecv();
    void enableRecv();
    void disableSnd();
    void enableSnd();
    int receive();
    bool send(BYTE data);
    Stat getStat();
    bool test(unsigned short testFrame);

    //FIFO-Konstanten
    static const int FIFOMAXSIZE = 1000;
    static const int ERRREADRECVERR = -1;

    //Kommunikations-Konstanten
    static const unsigned int TIMEINTERVALL = 20; //Periode in ms
};

//-----
//-----
//-

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

// Diese Klasse dient als Schnittstelle zwischen der Anwender-Klasse
// und dem eigentlichen kommunikations-Thread. Hier sind alle nötigen
// Methoden für die Kommunikation mit dem MOSES definiert und können
// vom Anwender verwendet werden
//-----

#include "StdAfx.h"
#include "Com.h"
#include "ComPort.h"

using namespace std;

// Konstruktor
Com::Com(void) {

    //FIFOs etc. zurücksetzen
    reset();

    // Erzeuge den eigentlichen Kommunikations-Thread
    // Verwendete Klasse: ComThread
    // Priorität: Normal
    // Thread zunächst pausiert erzeugen
    pThread = (ComThread*) AfxBeginThread( RUNTIME_CLASS(ComThread),
                                             THREAD_PRIORITY_NORMAL,
                                             NULL,
                                             CREATE_SUSPENDED,
                                             NULL);

    //Prüfe, ob Thread erfolgreich gestartet werden konnte
    if (pThread) {
        // Setze Sende- und Empfangs-FIFO im Kommunikations-Thread
        pThread->setQueuePtr(&sndFifo, &recvFifo);

        // Zunächst Empfang und Senden deaktivieren
        disableRecv();
        disableSnd();

        // Teile Kommunikations-Thread den verwendeten COM-Port mit
        pThread->setComPort(&cPort);

        // Starte den Thread
        pThread->ResumeThread();
    }

    //Fehler: Thread konnte nicht gestartet werden
    ASSERT(pThread != NULL);
}

//-----

// Destruktor
Com::~Com(void) {
    //Stoppe Thread und lösche Thread-Objekt
    pThread->SuspendThread();

    //Dynamisch erzeugtes Objekt muss entfernt werden
    delete(pThread);
}

```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

// Nur bei Objekt-Erzeugung aufrufen
void Com::reset() {
    //Sende- und Empfangs-FIFO zurücksetzen
    emptyRecvFifo();
    emptySndFifo();

    //Pointer zurücksetzen
    pThread = NULL;
}
//-----

// Alle Elemente des Empfangs-FIFOs entfernen
void Com::emptyRecvFifo(void) {
    while(!recvFifo.empty()) {
        recvFifo.pop();
    }
}
//-----

// Alle Elemente aus der Sende-FIFO löschen
void Com::emptySndFifo(void) {
    while(!sndFifo.empty()) {
        sndFifo.pop();
    }
}
//-----

// Gebe die verfügbaren COM-Schnittstellen zurück
void Com::getPorts(CString *ports) {
    cPort.getPorts(ports);
}
//-----

//Öffne den übergebenen COM-Port
bool Com::openPort(int num) {
    cPort.CloseCom(); //ggfs. alten Port schließen
    int stat = cPort.OpenCom(num);
    if(stat == ComPort::COMM_OK) {
        //In den Zustand Idle wechseln
        cPort.SetRTS(ComPort::ON);
        return true;
    }
    return false;
}
//-----

// Schließe COM-Port
void Com::closePort() {
    cPort.CloseCom();
}
//-----

// Deaktiviere Empfang
void Com::disableRecv() {
    pThread->setRecvEn(false);
}
//-----

// Setze Empfangsfreigabe
void Com::enableRecv() {
    pThread->setRecvEn(true);
}
//-----
  
```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

// Sendefreigabe weg nehmen
void Com::disableSnd() {
    pThread->setSndEn(false);
}
//-----

// Setze Sendefreigabe
void Com::enableSnd() {
    pThread->setSndEn(true);
}
//-----


// Lese ein Datenbyte aus dem Empfangs-FIFO
int Com::receive() {
    // Prüfe, ob Empfangs-FIFO leer ist
    if(!recvFifo.empty()) {
        int val = (int)recvFifo.front();
        recvFifo.pop();
        return val;
    }
    return ERRREADRECVERR; //Fifo ist leer
}
//-----


// Schreibe ein Datenbyte in das Sende-FIFO
bool Com::send(BYTE data) {
    // FIFO darf nicht die vorgeschriebene Größe nicht überschreiten
    if(sndFifo.size() == FIFOMAXSIZE)
        return false;

    // Schreibe in FIFO
    sndFifo.push(data);
    return true;
}
//-----


// Hole Status-Informationen vom Kommunikations-Thread
Stat Com::getStat() {
    Stat stat = pThread->getStat();
    //Zwei weitere Informationen hinzufügen
    stat.sizeRecvFifo = recvFifo.size();
    stat.sizeSndFifo = sndFifo.size();
    return stat;
}

//-----
bool Com::test(unsigned short testFrame) {
    if(pThread->isTestActive())
        return false;
    pThread->startTest(testFrame);
    return true;
}
//-----
```

Klasse: ThreadTimer

```
#pragma once

//-----
//-----
//-----
class ThreadTimer : public CWinThread
{
private:
    //HWND parentHWND;
    CWinThread* parent;
    float waitTime;
public:
    ThreadTimer();
    virtual ~ThreadTimer();
    virtual BOOL InitInstance();
    virtual int Run();
    //void SetHWnd(HWND hWnd);
    void setWaitTime(float time);
    void setParent(CWinThread* parent);
    CString sName ;
} ;
//-----
//-----
//-----
```

 Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

```

// Dies ist ein Timer, der in einem Thread läuft und bei Ablauf den
// Vater-Prozess benachrichtigt. Ursprünglich sollte dieser Timer genauer
// sein als die Standard-Timer. Auf gewissen PCs ist die Abweichung dennoch
// zu groß, sodass auf die Verwendung verzichtet wird.
#include "StdAfx.h"
#include "ThreadTimer.h"

// Konstruktor
ThreadTimer::ThreadTimer(void) {
    waitTime = (float) 0.02;
}
//-----

// Destruktor
ThreadTimer::~ThreadTimer(void) {
}
//-----


// Geerbte Methode wird überschrieben
BOOL ThreadTimer::InitInstance() {
    return TRUE;
}
//-----


int ThreadTimer::Run() {
    /*Dieser auskommentierte Teil ist zwar sehr genau, da
    er aber in einer Schleife läuft wird viel zu viel unnötige Prozessor-
    Zeit verschwendet!!!*/
    __int64 end_count, start_count;
    __int64 freq;
    float time;

    //Frequenz ermitteln
    QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
    QueryPerformanceCounter((LARGE_INTEGER*)&start_count);
    do {
        QueryPerformanceCounter((LARGE_INTEGER*)&end_count);
        time = (float)(end_count - start_count) / (float)freq;
    }
    while(time < waitTime);*/

    //Auch extrem genau
    Sleep((int)waitTime);

    //Nachricht an Vater-Prozess schicken, dass Zeit abgelaufen ist
    parent->PostThreadMessage(WM_THREAD_TERMINATED, WM_THREAD_TERMINATED, NULL);
    return 0;
}
//-----


// Vater-Prozess setzen, für Benachrichtigung benötigt
void ThreadTimer::setParent(CWinThread* parent) {
    this->parent = parent;
}
//-----


// Stelle Wartezeit ein
void ThreadTimer::setWaitTime(float time) {
    waitTime = time;
}
//-----
```

13. Abbildungsverzeichnis

Abbildung 1 Beispiel einer Realisierung	8
Abbildung 2 Grafische Darstellung des Programmablaufes des Moses	12
Abbildung 3 Verdrahtungsplan SPS - Moses	15
Abbildung 4 Zeitablauf einer Übertragung.....	18
Abbildung 5 Ablauf des Organisations-Bausteins OB1	31
Abbildung 6 Ablauf des ComMoses FB's	32
Abbildung 7 Ablauf der Funktion RcvMoses	33
Abbildung 8 Ablauf der Funktion SndMoses	34
Abbildung 9 Startbildschirm WinCC	36
Abbildung 10 Verdrahtungsplan für das Interface PC-SPS/Moses	39
Abbildung 11 Bildschirm C++-Programm	40
Abbildung 12 Übertragung gesamt	43
Abbildung 13 Zeitmessung zwischen Stop- und ACK-Bit	43
Abbildung 14 Letzen Schritte der Kommunikation.....	45
Abbildung 15 Übertragung eines Frames mit Paritätsfehler.....	46
Abbildung 16 Negative Quittierung.....	46
Abbildung 17 Fehlerhaftes Stop-Bit	48
Abbildung 18 Fehlerhaftes Start-Bit	49
Abbildung 19 Vollständig belegtes Empfangs-FIFO	50
Abbildung 20 Allgemeiner Test: Vollständiger Sendevorgang	51
Abbildung 21 Zoom zum letzten Teil der Übertragung	51
Abbildung 22 Verpasstes / Ignoriertes Start-Bit.....	53
Abbildung 23 Frühzeitiges Verlassen des Empfangsmodus (1)	54
Abbildung 24 Frühzeitiges Verlassen des Empfangsmodus(2).....	55
Abbildung 25 Empfänger meldet NACK (Negative Quittierung).....	56
Abbildung 26 Abwechselndes Senden / Empfangen (1)	57
Abbildung 27 Abwechselndes Senden / Empfangen (2)	58
Abbildung 28 Abwechselndes Senden / Empfangen (3)	58
Abbildung 29 PAP Hauptprogramm MOSES.....	61
Abbildung 30 PAP Beispiel: Empfangsprogramm.....	63
Abbildung 31 PAP Sichere Übertragung.....	65
Abbildung 32 PAP SPSAusDigEin	67
Abbildung 33 PAP SPSAusFeld.....	70
Abbildung 34 PAP putFIFO	71
Abbildung 35 PAP SendNum	74
Abbildung 36 PAP InitFIFO MOSES	79
Abbildung 37 PAP DeleteFIFO MOSES.....	81
Abbildung 38 PAP ReadFIFO MOSES	84
Abbildung 39 PAP writeFIFO	91
Abbildung 40 PAP getFIFOPointer.....	93
Abbildung 41 PAP fifoMult	95
Abbildung 42 Prim-Funktion (1)	100
Abbildung 43 Prim-Funktion (2)	101
Abbildung 44 Prim-Funktion (3)	102

Realisierung einer seriellen Standard-Schnittstelle zwischen SPS und µP-System

Abbildung 45 PAP SPSIni	108
Abbildung 46 PAP receiveCheck.....	111
Abbildung 47 PAP SPSAus	118
Abbildung 48 PAP USR_IRQ.....	121
Abbildung 49 PAP IRQ_A	121
Abbildung 50 PAP IRQ_T	122
Abbildung 51 PAP IRQ_T1	126
Abbildung 52 PAP IRQ_T2	131
Abbildung 53 PAP IRQ_B	135
Abbildung 54 PAP parity.....	137
Abbildung 55 PAP initFIFO.....	138
Abbildung 56 PAp writeFIFO	141
Abbildung 57 PAP readFIFO	144
Abbildung 58 PAP countFIFO.....	146
Abbildung 59 PAP countEmptyFIFO	147