CAN マルチネットワークボード
取扱説明書　ソフトウェア編

ルネサス エレクトロニクス社　RX231, RL78/F15, RA2L1 搭載
HSB シリーズ応用キット

-本書を必ずよく読み、ご理解された上でご利用ください

株式会社北斗電子

REV.1.0.0.0

ー目　次ー

注意事項

本書を必ずよく読み、ご理解された上でご利用ください

【ご利用にあたって】

1. 本製品をご利用になる前には必ず取扱説明書をよく読んで下さい。また、本書は必ず保管し、使用上不明な点がある場合は再読し、よく理解して使用して下さい。
2. 本書は株式会社北斗電子製マイコンボードの使用方法について説明するものであり、ユーザシステムは対象ではありません。
3. 本書及び製品は著作権及び工業所有権によって保護されており、全ての権利は弊社に帰属します。本書の無断複写・複製・転載はできません。
4. 弊社のマイコンボードの仕様は全て使用しているマイコンの仕様に準じております。マイコンの仕様に関しましては製造元にお問い合わせ下さい。弊社製品のデザイン・機能・仕様は性能や安全性の向上を目的に、予告無しに変更することがあります。また価格を変更する場合や本書の図は実物と異なる場合もありますので、御了承下さい。
5. 本製品のご使用にあたっては、十分に評価の上ご使用下さい。
6. 未実装の部品に関してはサポート対象外です。お客様の責任においてご使用下さい。

【限定保証】

1. 弊社は本製品が頒布されているご利用条件に従って製造されたもので、本書に記載された動作を保証致します。
2. 本製品の保証期間は購入戴いた日から1年間です。

【保証規定】

保証期間内でも次のような場合は保証対象外となり有料修理となります

1. 火災・地震・第三者による行為その他の事故により本製品に不具合が生じた場合
2. お客様の故意・過失・誤用・異常な条件でのご利用で本製品に不具合が生じた場合
3. 本製品及び付属品のご利用方法に起因した損害が発生した場合
4. お客様によって本製品及び付属品へ改造・修理がなされた場合

【免責事項】

弊社は特定の目的･用途に関する保証や特許権侵害に対する保証等、本保証条件以外のものは明示・黙示に拘わらず一切の保証は致し兼ねます。また、直接的・間接的損害金もしくは欠陥製品や製品の使用方法に起因する損失金・費用には一切責任を負いません。損害の発生についてあらかじめ知らされていた場合でも保証は致し兼ねます。
ただし、明示的に保証責任または担保責任を負う場合でも、その理由のいかんを問わず、累積的な損害賠償責任は、弊社が受領した対価を上限とします。本製品は「現状」で販売されているものであり、使用に際してはお客様がその結果に一切の責任を負うものとします。弊社は使用または使用不能から生ずる損害に関して一切責任を負いません。
保証は最初の購入者であるお客様ご本人にのみ適用され、お客様が転売された第三者には適用されません。よって転売による第三者またはその為になすお客様からのいかなる請求についても責任を負いません。
本製品を使った二次製品の保証は致し兼ねます。

安全上のご注意

製品を安全にお使いいただくための項目を次のように記載しています。絵表示の意味をよく理解した上でお読み下さい。

表記の意味

警告 取扱を誤った場合、人が死亡または重傷を負う危険が切迫して生じる可能性がある事が想定される

注意 取扱を誤った場合、人が軽傷を負う可能性又は、物的損害のみを引き起こすが可能性がある事が想定される

絵記号の意味

	**一般指示** 使用者に対して指示に基づく行為を強制するものを示します		**一般禁止** 一般的な禁止事項を示します
	**電源プラグを抜く** 使用者に対して電源プラグをコンセントから抜くように指示します		**一般注意** 一般的な注意を示しています

警告

以下の警告に反する操作をされた場合、本製品及びユーザシステムの破壊・発煙・発火の危険があります。マイコン内蔵プログラムを破壊する場合もあります。

1. 本製品及びユーザシステムに電源が入ったままケーブルの抜き差しを行わないでください。
2. 本製品及びユーザシステムに電源が入ったままで、ユーザシステム上に実装されたマイコンまたはIC等の抜き差しを行わないでください。
3. 本製品及びユーザシステムは規定の電圧範囲でご利用ください。
4. 本製品及びユーザシステムは、コネクタのピン番号及びユーザシステム上のマイコンとの接続を確認の上正しく扱ってください。

発煙・異音・異臭にお気付きの際はすぐに使用を中止してください。

電源がある場合は電源を切って、コンセントから電源プラグを抜いてください。そのままご使用すると火災や感電の原因になります。

⚠注意

以下のことをされると故障の原因となる場合があります。

1. 静電気が流れ、部品が破壊される恐れがありますので、ボード製品のコネクタ部分や部品面には直接手を触れないでください。
2. 次の様な場所での使用、保管をしないでください。
   ホコリが多い場所、長時間直射日光があたる場所、不安定な場所、衝撃や振動が加わる場所、落下の可能性がある場所、水分や湿気の多い場所、磁気を発するものの近く
3. 落としたり、衝撃を与えたり、重いものを乗せないでください。
4. 製品の上に水などの液体や、クリップなどの金属を置かないでください。
5. 製品の傍で飲食や喫煙をしないでください。

ボード製品では、裏面にハンダ付けの跡があり、尖っている場合があります。

取り付け、取り外しの際は製品の両端を持ってください。裏面のハンダ付け跡で、誤って手など怪我をする場合があります。

CD メディア、フロッピーディスク付属の製品では、故障に備えてバックアップ(複製)をお取りください。

製品をご使用中にデータなどが消失した場合、データなどの保証は一切致しかねます。

アクセスランプがある製品では、アクセスランプ点灯中に電源の切断を行わないでください。

製品の故障や、データの消失の原因となります。

本製品は、医療、航空宇宙、原子力、輸送などの人命に関わる機器やシステム及び高度な信頼性を必要とする設備や機器などに用いられる事を目的として、設計及び製造されておりません。

医療、航空宇宙、原子力、輸送などの設備や機器、システムなどに本製品を使用され、本製品の故障により、人身や火災事故、社会的な損害などが生じても、弊社では責任を負いかねます。お客様ご自身にて対策を期されるようご注意ください。

1. 概要

本書では、CAN マルチネットワークボードのソフトウェアに関して記載します。

1.1. CAN マルチネットワーク　ソースコード CD に関して

「CAN マルチネットワーク　ソースコード CD」は、以下の内容が含まれます。

フォルダ		
HSB_CAN_MULTI¥	HSB_CAN_MULTI_RX231¥	RX231 向けプロジェクトフォルダ（CS+プロジェクト）
	HSB_CAN_MULTI_RL78F15¥	RL78/F15 向けプロジェクトフォルダ（CS+プロジェクト）
	HSB_CAN_MULTI_RA2L1.zip	RA2L1 向けプロジェクトフォルダ（e2studio アーカイブ）
HSB_LIN_COMM¥	RL78_G11_LIN_COMM¥	HSB_LIN_COMM 向けプロジェクトフォルダ（CS+プロジェクト）
PC¥	HSB_CAN_MULTI_DEMO¥	HSB_CAN_MULTI_DEMO.exe プロジェクトフォルダ(VisualC#)
	HSB_CAN_MULTI_DEMO2¥	HSB_CAN_MULTI_DEMO2.exe プロジェクトフォルダ(VisualC#)

CAN マルチネットワークボード、HSB_CAN_MULTI_n(RX231), HSB_CAN_MULTI_n(RL78F15)向けのソースは、CS+(CS+forCC)向けのプロジェクトとなっていますので、CS+forCC（ver8.09.00 以降）をご用意ください。
e2studio+CC-RX, e2studio+CC-RL の組み合わせでもプロジェクトの読み込みは可能ですので、開発環境として e2studio を使用する事も可能です。

CAN マルチネットワークボード、HSB_CAN_MULTI_n(RA2L1)向けのソースは、e2studio+FSP 向けのプロジェクトとなっていますので、アーカイブをワークスペースに展開して、ソースの変更、ビルドを行ってください。

HSB_LIN_COMM は、CS+(CS+forCC)向けのプロジェクトとなっています。CS+forCC または、e2studio+CC-RL で開発可能です。

PC 向けのアプリケーション、HSB_CAN_MULTI_DEMO, HSB_CAN_MULTI_DEMO2 は、Visual C#で作成されています。Visual Studio 2019 以降の環境でビルドを行ってください。

1.2. HSB_CAN_MULTI ボード向け変更箇所

HSB_CAN_MULTI_1 ~ HSB_CAN_MULTI_4 は、1 つのプロジェクトで、4 種類のバイナリ（マイコンボードに書き込む mot/srec ファイル）を生成できるように構成されています。

ターゲットボードの設定は、

main¥main.h

```
#ifndef __MAIN_H__
#define __MAIN_H__

 /*---------------------------------------------------------------------
   ターゲットボードタイプ
---------------------------------------------------------------------*/
#define HSB_CAN_MULTI_1  0
#define HSB_CAN_MULTI_2  1
#define HSB_CAN_MULTI_3  2
#define HSB_CAN_MULTI_4  3

//CPU_CARD_RA2L1_64 を挿すボードに合わせて変更

//※変更要（いずれか 1 つを選択）

//#define BOARD_TYPE  HSB_CAN_MULTI_1
//#define BOARD_TYPE  HSB_CAN_MULTI_2
#define BOARD_TYPE  HSB_CAN_MULTI_3
//#define BOARD_TYPE  HSB_CAN_MULTI_4

//変更要　ここまで※
```

main.h ファイル内の先頭のターゲットボードタイプのところを変更する必要があります。

BOARD_TYPE の定数設定値を
HSB_CAN_MULTI_1 ~ HSB_CAN_MULTI_4 のいずれかとしてください
（4 行の内、1 つのみコメントアウトを外してください）

その後、プロジェクトをビルドする事により、ターゲットボードに合わせた mot/srec ファイルが出力されますので、そのファイルを HSB_CAN_MULTI_n(n=1~4)ボードに書き込んでください。

プログラムのビルド、書き込みに関しては「取扱説明書　開発環境編」を参照してください。

2. 全体の流れ

2.1. HSB_CAN_MULTI

HSB_CAN_MULTI_RX231
HSB_CAN_MULTI_RL78F15
HSB_CAN_MULTI_RA2L1
プロジェクトの説明です。3 種のプロジェクトは、ターゲットマイコンが異なりますが、処理内容は同一です。

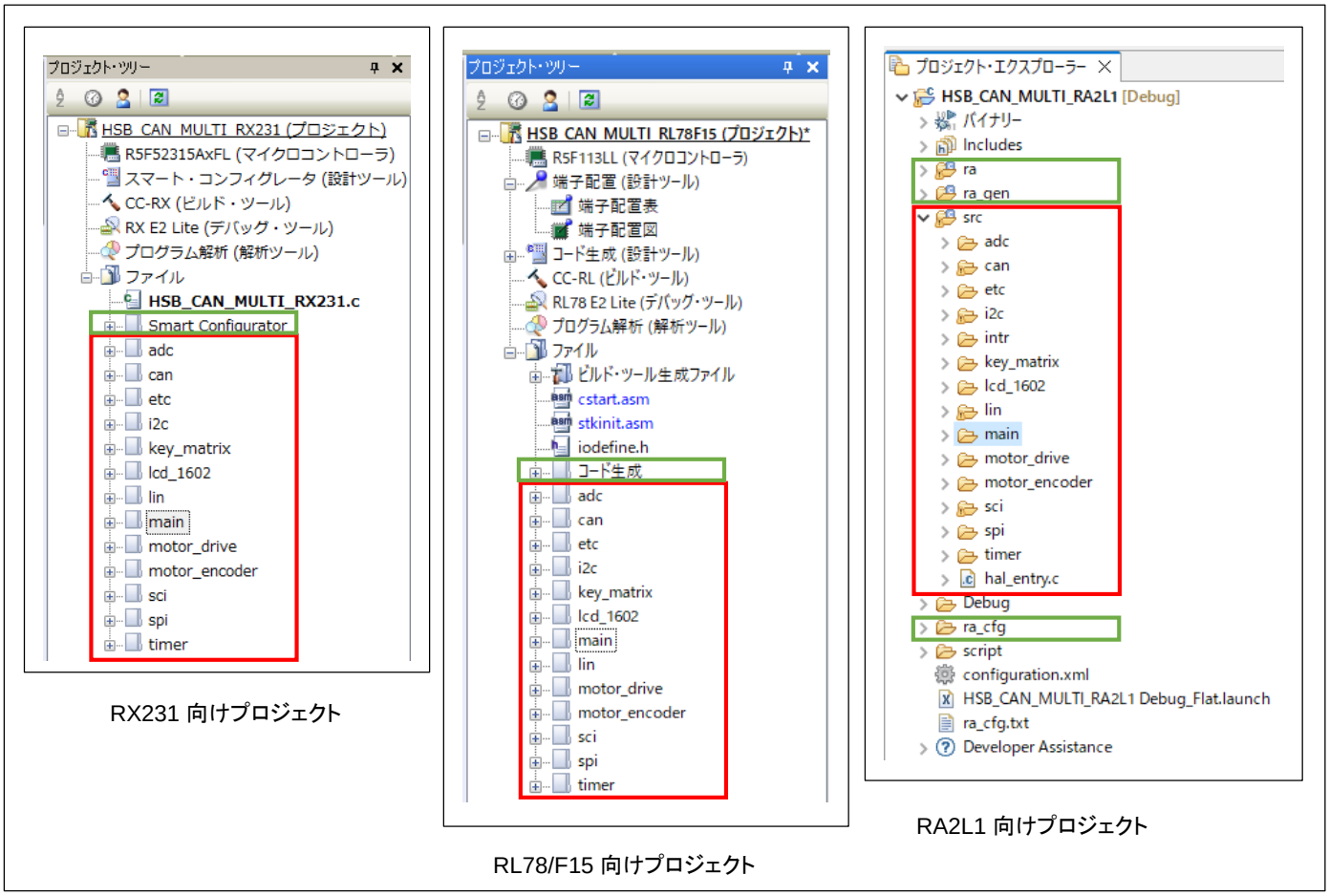
プロジェクトは、HSB_CAN_MULTI_1 ~ HSB_CAN_MULTI_4 までの 4 種類のバイナリ（mot ファイル）を、定数設定で変更して出力出来る様になっています。

2.1.1. ソースコード階層（ユーザ作成コード）

ソースコードは、自動生成(*1)とユーザ作成コードに 2 分されます。

(*1)RX231 では「スマートコンフィグレータ」、RL78/F15 では「コード生成」、RA2L1 では「FSP」がマイコンを動かす上で必須となる基本的なプログラムコードを自動で生成しています。
マイコンを深く理解し、隅々まで制御したい場合は、自動コード生成でどのようなコードが生成されて実行されているのか理解する必要がありますが、アプリケーションソフトを構築するという観点では、ツールによる自動コード生成機能は便利なものであると思います。本プロジェクトにおいても、自動コード生成で処理可能な内容は自動コード生成で出力されたコードを使用しています。

ユーザ作成コードは、



赤で囲まれた部分がユーザ作成コードです。緑で囲まれた部分が、自動生成ツールが出力するコードです。本書では、「ユーザ作成コード」に関して説明します。（自動生成ツールが出力する緑で囲まれた部分は、「取扱説明書　開発環境編」で説明しています。

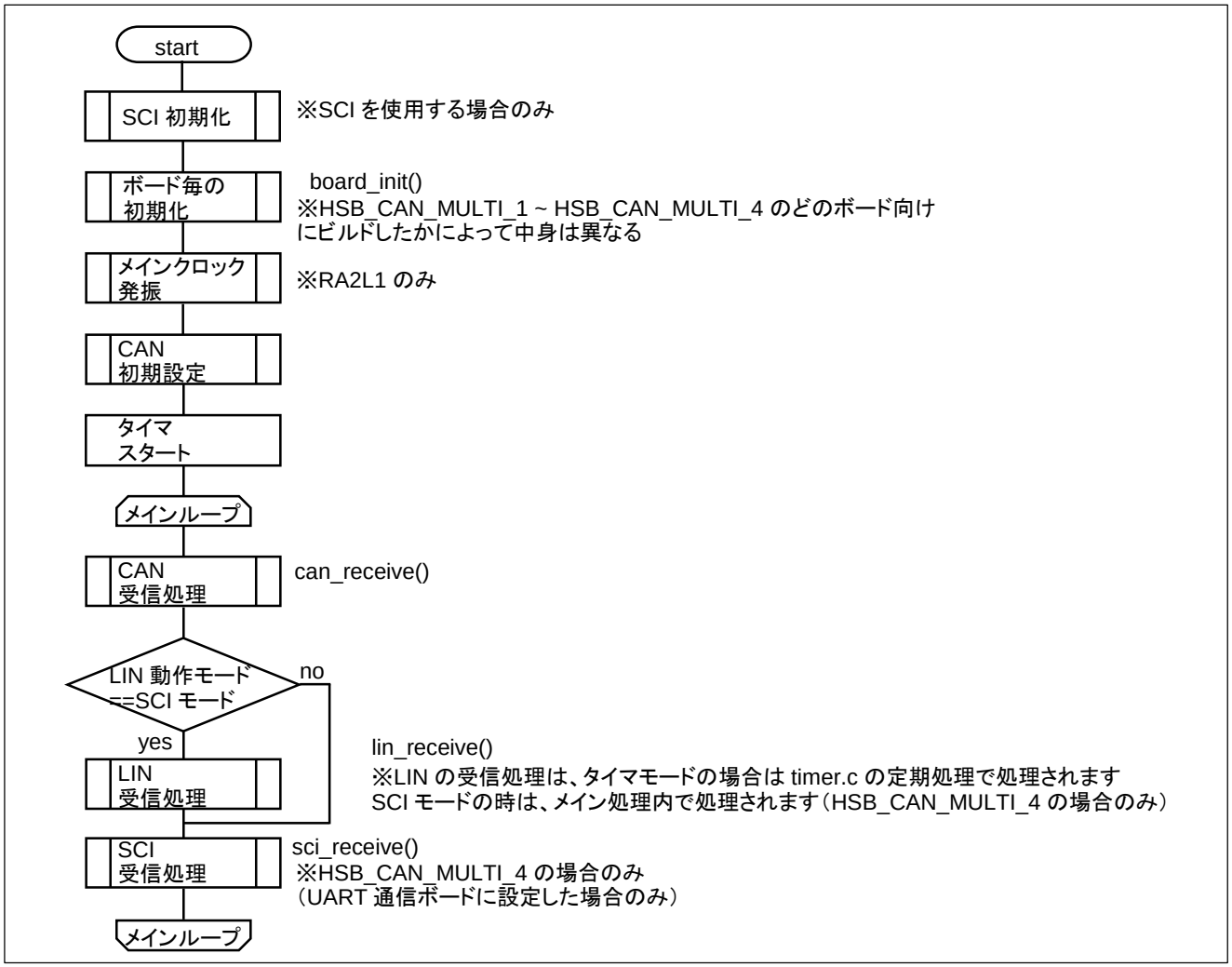
ユーザ作成コード

フォルダ	ファイル	内容
adc	adc.c adc.h	HSB_CAN_MULTI_3 向け フォトダイオード（明るさセンサ）の読み取りに使用する A/D 変換処理
can	※CPU タイプによりソースコード構成が異なる	CAN 通信処理
etc	etc.h	コマンドの定数定義
i2c	i2c.c i2c.h	HSB_CAN_MULTI_3 向け I2C の温度センサにアクセスする処理
intr	intr.c intr.h	割り込みの初期化 ※RA2L1 のみ
key_matrix	key_matrix.c key_matrix.h key_matrix_board_def_CPU_CARD_*XXXX*.h	HSB_CAN_MULTI_2 向け キーマトリックスの読み取り処理 ※XXXX は CPU タイプ毎に相違
lcd_1602	lcd_1602.c lcd_1602.h lcd_1602_board_def_CPU_CARD_*XXXX*.h	HSB_CAN_MULTI_2 向け LCD の表示処理 ※XXXX は CPU タイプ毎に相違
lin	lin.c lin.h lin_operation.h	HSB_CAN_MULTI_4 向け LIN 通信処理
main	main.c main.h	メイン処理 ID 定義、ターゲットボード(HSB_CAN_MULTI_*n*)の定義等

motor_drive	motor_drive.c motor_drive.h	HSB_CAN_MULTI_1 向け モータ駆動処理
motor_encoder	motor_encoder.c motor_encoder.h	HSB_CAN_MULTI_1 向け モータ回転数取得処理
sci	sci.c sci.h readme.txt	HSB_CAN_MULTI_4 向け UART(SCI)通信処理 ※readme.txt は SCI モジュールの説明ファイル
spi	spi.c spi.h	HSB_CAN_MULTI_3 向け SPI フラッシュメモリにアクセスする処理
timer	timer.c timer.h	定期処理

ユーザ作成コードは、特定のボード(HSB_CAN_MULTI_*n*)向けの処理と、どのボードでも使用する処理（CAN や timer の定期処理等）があります。

2.1.2. メイン処理(main.c, usr_main())



マイコンボードを動作させる上で必要な前処理（クロックの設定等）は、コード自動生成で出力されて実行されています。ここではユーザ関数のメイン処理(usr_main())の処理内容を示します。

HSB_CAN_MULTI_4 は、PC と UART(SCI)で通信を行うので、ターゲットを HSB_CAN_MULTI_4 に設定した場合は SCI の初期化や SCI の受信処理が含まれます。SCI を使用しない、HSB_CAN_MULTI_1 ~ HSB_CAN_MULTI_3 の場合は、SCI の初期化、受信は行いません。）

[参考]ボード構成に HSB_CAN_MULTI_4 が含まれない場合は、定義ファイルの設定で HSB_CAN_MULTI_1 ~ HSB_CAN_MULTI_3 のいずれかのボードに、UART 通信ボードの機能を担わせる事も可能です。

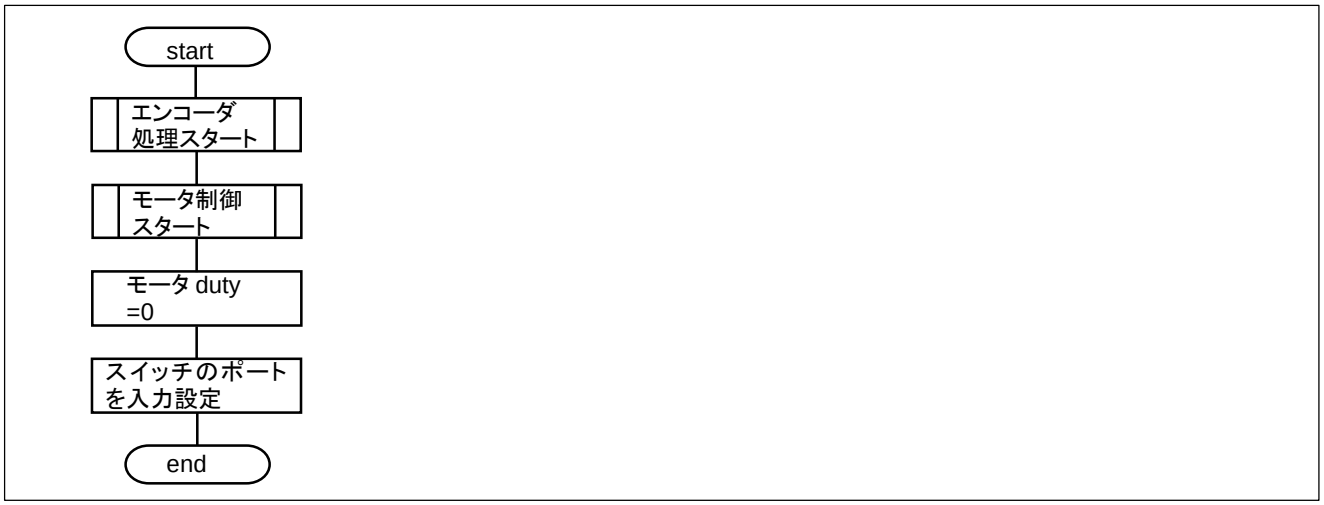
メインクロックの発振処理は、RA2L1 のみ行っていますが、CPU のコアクロックは RA2L1 はマイコン内蔵のオシレータが生成します。CAN の通信処理でメインクロック（外付けの水晶振動子ベースのクロック）を使用しています。そのため、メインクロックの発振処理が含まれます。RX231, RL78/F15 は CPU のコアクロックを、メインクロックベースとしていますので、ユーザメイン関数到達前にメインクロック発振処理が行われています。

メイン処理以外の処理は、タイマ処理、割り込み処理（受信等）があります。

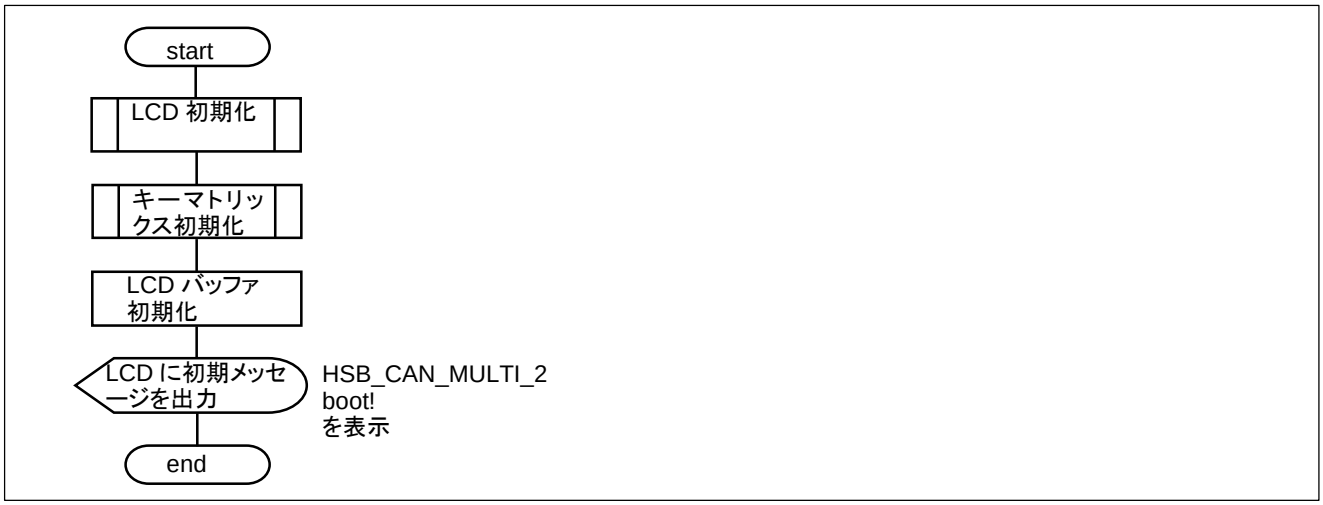
2.1.3. ボード毎の初期化(main.c, board_init())

ボード毎の初期化処理は、HSB_CAN_MULTI_1 ~ HSB_CAN_MULTI_4 で、条件コンパイル（#ifdef ~ #endif）で処理内容を変えています。

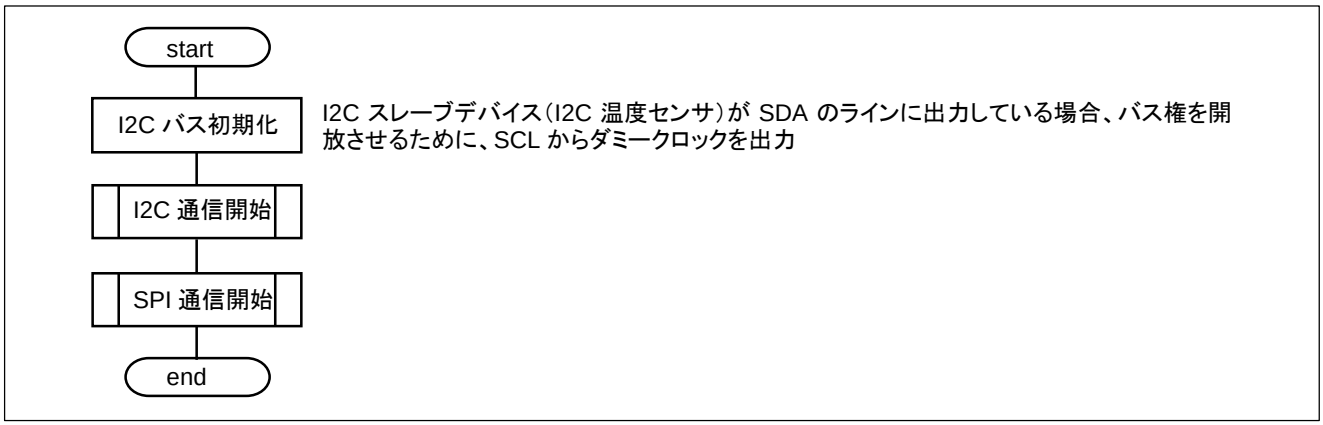
(1)HSB_CAN_MULTI_1



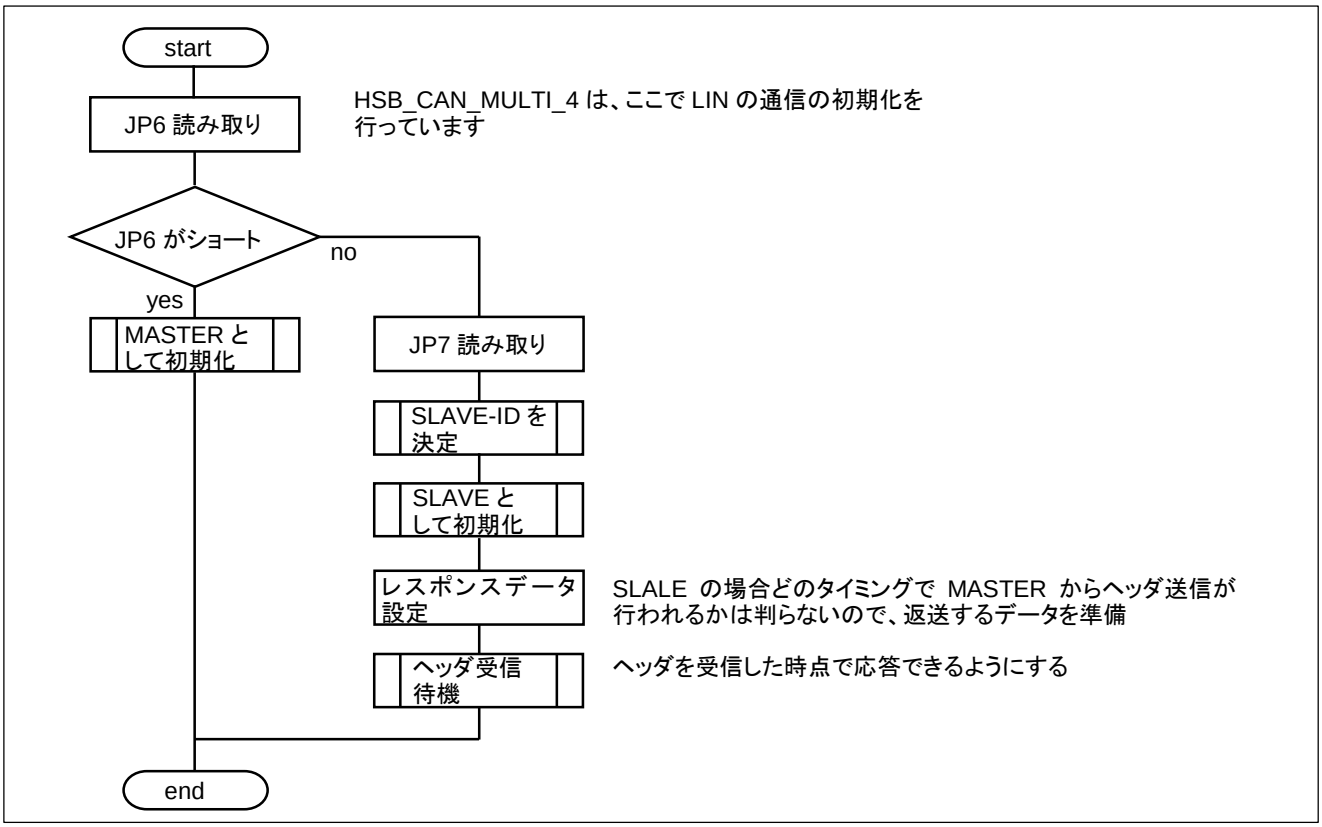
(2)HSB_CAN_MULTI_2



(3)HSB_CAN_MULTI_3

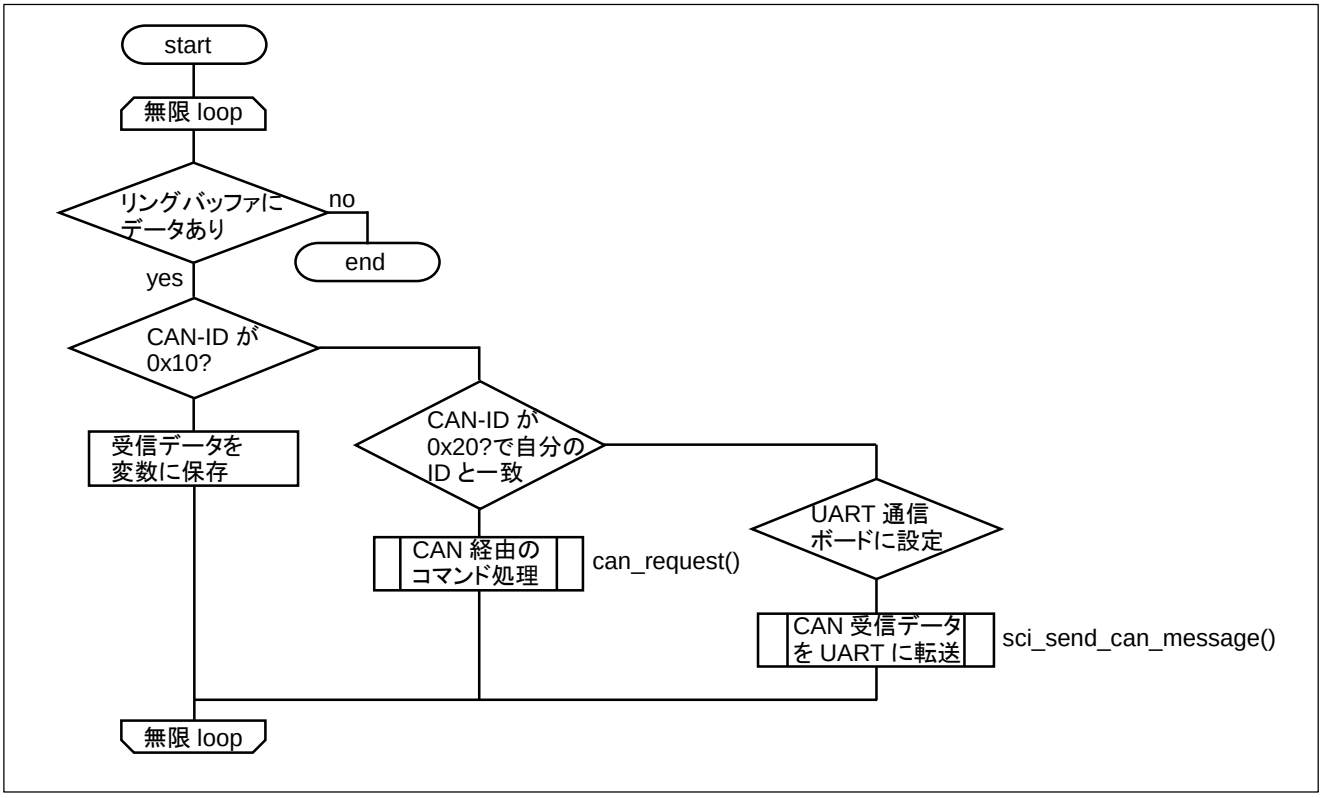


(4)HSB_CAN_MULTI_4



2.1.4. CAN 受信処理(main,c, can_receive())

CAN のデータ受信自体は、割り込みでリングバッファに保存する処理となっています。リングバッファを読み込んで、受信データが存在する場合、本関数で処理が行われます。



受信した CAN データの ID が 0x10?の場合は、各ボードが定期的(500ms 毎)に発信している、ボードの情報です。この場合、受信したデータを変数に保存する(自分以外のボードがどのようなデータを送ってきているかは気にしておく)という動作としています。

受信したデータの ID が 0x201 かつ、受信したボードが HSB_CAN_MULTI_1 の場合、CAN パケットで自分のボードに指示が送られてきているので、その指示に従います(can_request()関数が呼ばれます)。

ボードが UART 通信ボード(デフォルトでは HSB_CAN_MULTI_4 が UART 通信ボード)に設定されている場合は、ID=20?のデータは UART に転送します。(PC 上で実行される HSB_CAN_MULTI_DEMO2.exe で表示させるためです。)

2.1.5. LIN 受信処理(main,c, lin_receive())

LIN のデータ受信自体は、割り込みでリングバッファに保存する処理となっています。リングバッファを読み込んで、受信データが存在する場合、本関数で処理が行われます。

ボードが UART 通信ボード（デフォルトでは HSB_CAN_MULTI_4 が UART 通信ボード）に設定されている場合は、受信した LIN のデータは UART に転送します。（PC 上で実行される HSB_CAN_MULTI_DEMO2.exe で表示させるためです。）

2.1.6. 定期処理(timer.c)

timer.c 内では、タイマを使い定期的に実行する処理が実行されます。定期処理は、ボード毎に実行される内容が異なります。ボード毎に異なる処理の実体は 6 章で処理内容を示します。ここでは、定期処理でどのような内容が実行されるかを示します。

(1)HSB_CAN_MULTI_1

・CAN データ送信（500ms 毎）

モータ duty 値(0xXX)
モータ回転数(0xYYYY)
を CAN のデータパケットに埋め込み

ID	RTR	IDE	DLC	data							
0x00000101	0b0	0b1	0b1000	0xXX	0xYY	0xYY	0x00	0x00	0x00	0x00	0x00

上記の 8 バイトの CAN データとして送信。

・スイッチ読み取り（1ms 毎）

ボード上の SW2（duty を+25[%]する）、SW3（duty を-25[%]する）を読み込んで、モータ duty 値に反映させます。

duty は、+100[%]（正回転） ~ -100[%]（逆回転）まで変動します。

SW2, SW3 を押した場合、duty 値は、25 で割り切れる値に切り上げ、切り捨てされます（30%の時に SW2 を押すと、duty は 50%になります）。

スイッチは、押されている状態から離したタイミングで、反応します。

・回転数取得(50ms 毎）

モータの回転数（10ms 間にエンコーダを何回横切ったかをカウント）を変数に保存します。

(2)HSB_CAN_MULTI_2

・CAN データ送信（500ms 毎）

LCD 表示フラグ(0xXX)
マトリックスキーの状態 0xYYYY)
を CAN のデータパケットに埋め込み

ID	RTR	IDE	DLC	data							
0x00000102	0b0	0b1	0b1000	0xXX	0xYY	0xYY	0x00	0x00	0x00	0x00	0x00

上記の 8 バイトの CAN データとして送信。

・マトリックスキーの読み取り（1ms 毎）

マトリックスキーの状態を読み取り、変数に保存します。

・LCD 表示更新（50ms 毎）

LCD の表示内容（CAN パケットを使い外部から制御可能）に応じて、LCD の表示更新を行います。

(3)HSB_CAN_MULTI_3

・CAN データ送信（500ms 毎）

明るさセンサの A/D 変換値(0xXXXX)
温度センサの温度値 0xYYYY)
を CAN のデータパケットに埋め込み

ID	RTR	IDE	DLC	data							
0x00000103	0b0	0b1	0b1000	0xXX	0xXX	0xYY	0xYY	0x00	0x00	0x00	0x00

上記の 8 バイトの CAN データとして送信。

・センサ値取得（50ms 毎）

明るさセンサ（フォトダイオード）A/D 変換値
I2C で温度センサの温度値の読み取り

を行います。

(4)HSB_CAN_MULTI_4

・CAN データ送信（500ms 毎）

ID	RTR	IDE	DLC	data							
0x00000104	0b0	0b1	0b1000	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00

上記の 8 バイトの CAN データとして送信。（データは全て 0x00、ボードの存在確認のために送信）

・LIN 送信（50ms 毎）

LIN の動作がタイマモードの時、かつ MASTER モードの時 50ms の 2 回に 1 回（100ms 毎）に、ID=0x30~0x34 の MASTER ヘッダ送信（ID=0x30 の時はレスポンス送信も）行います。

2.2. HSB_LIN_COMM

RL78_G11_LIN_COMM

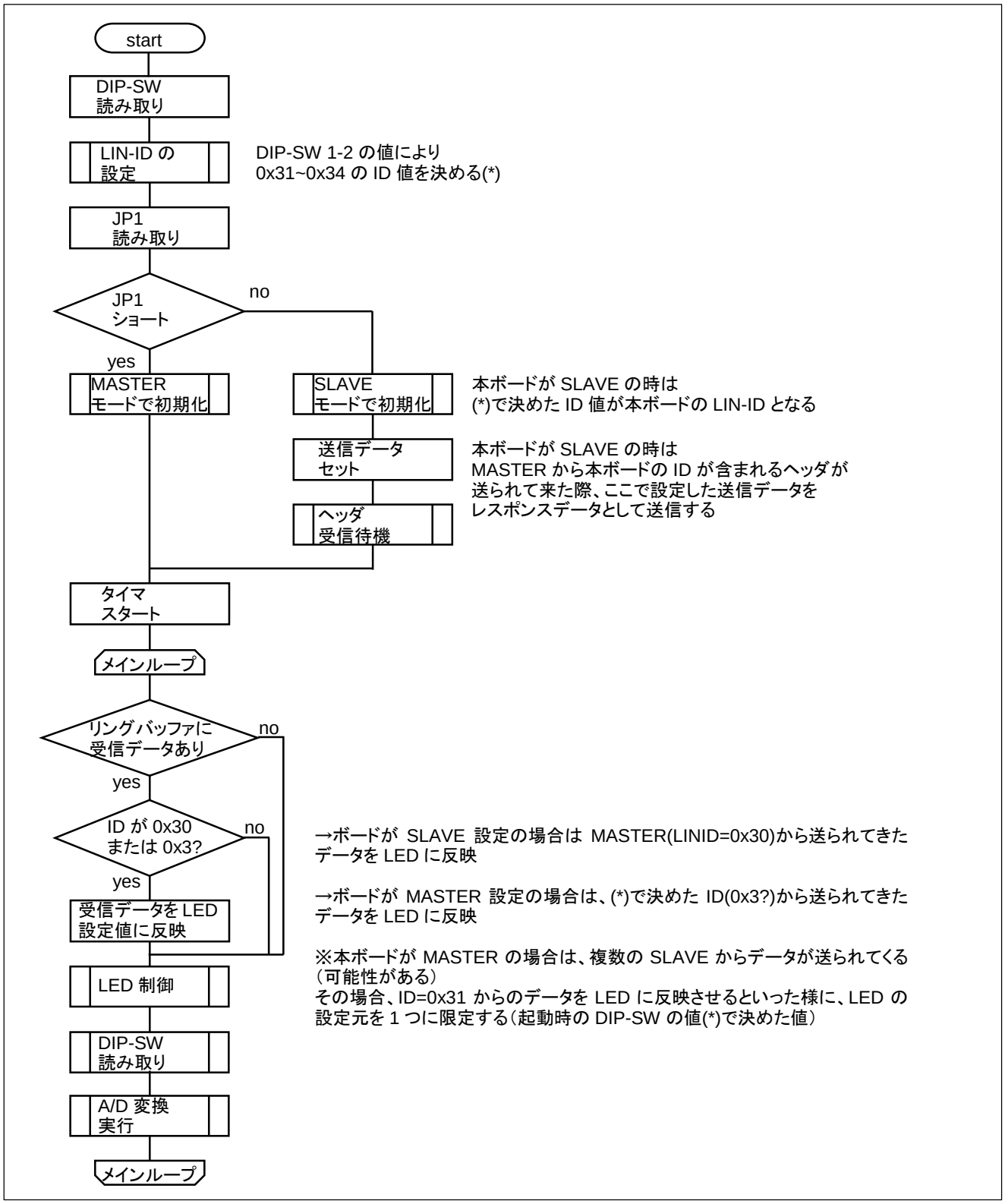
プロジェクトの説明です。

2.2.1. ソースコード階層（ユーザ作成コード）

ユーザ作成コード

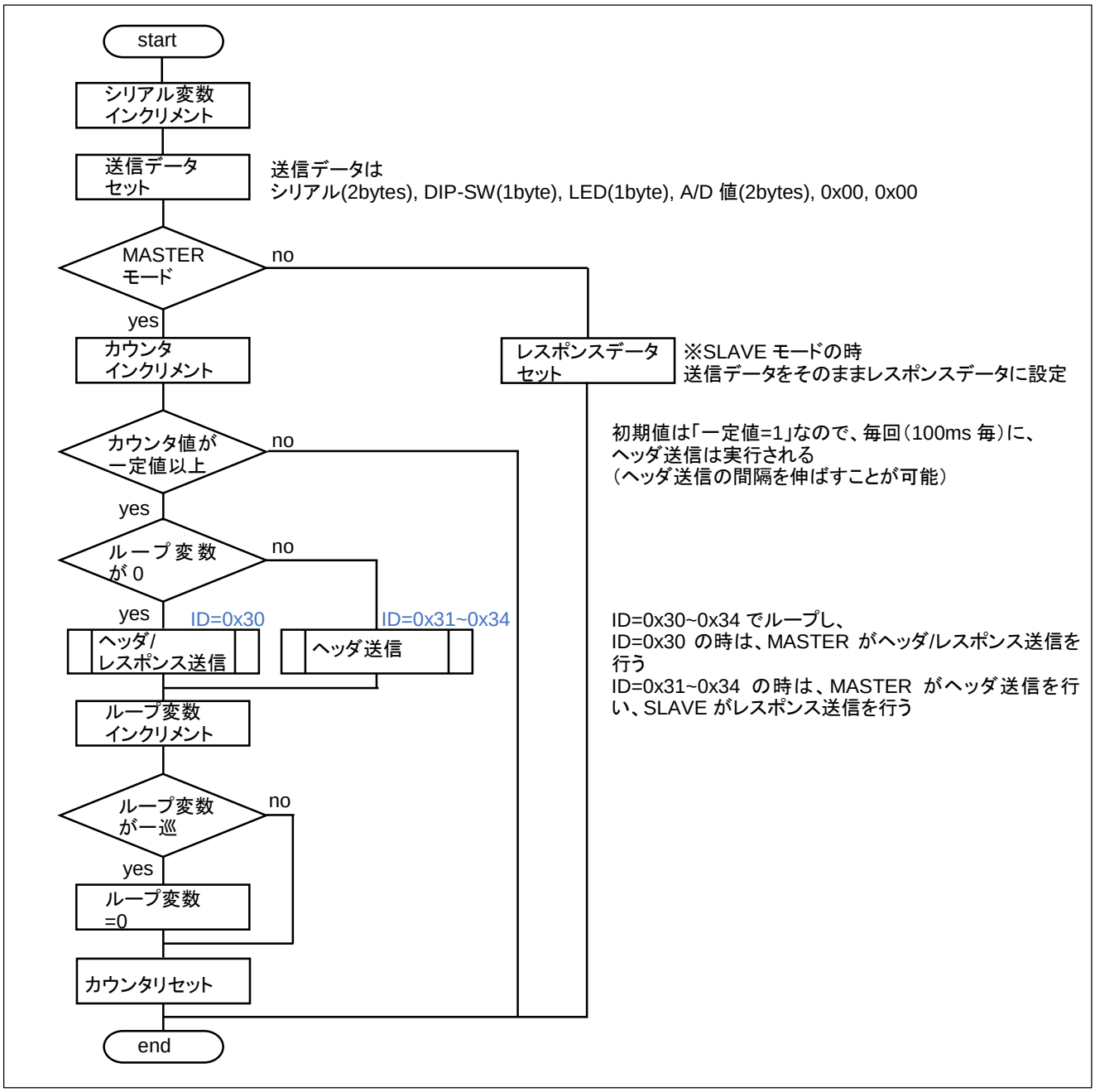
フォルダ	ファイル	内容
lin	lin.c lin.h lin_operation.h	LIN 通信処理
main	main.c main.h	メイン処理
sci	sci.c sci.h readme.txt	UART(SCI)通信処理 ※readme.txt は SCI モジュールの説明ファイル
timer	timer.c timer.h	定期処理

2.2.2. メイン処理(main.c, usr_main())



メインループで受信データの処理が行われるが、受信自体は割り込みで行うようになっています。(受信割り込みで、受信データをリングバッファに格納)

2.2.3. 定期処理（timer.c, interval_operation (), 100ms 毎）



定期処理は、100ms 毎に実行され、MASTER モードの時には、この関数内でヘッダ送信処理が実行されます。SLAVE モードの時は、レスポンスデータの設定のみ行われます。（SLAVE モードの時は、LIN-ID=自分自身の ID のヘッダが送られてきたタイミングで、レスポンスデータの送信が行われます。）

LIN の割り込み処理で実行される部分は HSB_CAN_MULTI ボードと HSB_LIN_COMM ボードで共通となり、4 章の LIN 通信の部分で説明します。

3. CAN 通信

3.1. マイコン搭載の CAN モジュール

マイコン種により、CAN の部分のプログラムは大きく異なります。

マイコン種	CAN モジュール	備考
RX231	RS-CAN	最大 1Mbps 送信バッファ 4 個 受信 FIFO 2 本（最大 16 段）(*) 送受信 FIFO 1 本（最大 16 段）(*) 受信バッファ 16 個(*) (*)受信 FIFO，送受信 FIFO，受信バッファのトータルで 16 個のバッファ 1ch（CAN0 のみ）
RL78/F15	RS-CAN Lite	最大 1Mbps 送信バッファ 4 個/ch 受信 FIFO 4 本（最大 32 段）(*) 送受信 FIFO 1 本/ch（最大 32 段）(*) 受信バッファ 32 個(*) (*)受信 FIFO，送受信 FIFO，受信バッファのトータルで 40 個のバッファ 2ch（CAN0/CAN1 あり）
RA2L1	CAN	最大 1Mbps メールボックス 32 個/ch(*) RXFIFO 1 本（4 段）/ch(*) RXFIFO 1 本（4 段）/ch(*) (*)メールボックスを FIFO に割り当てる方式、トータルで 32 個のメッセージバッファ （CAN ch あたり） ※CAN モジュール自体は多数 ch をサポートしている、RA2L1 では 1ch（CAN0 のみ）

RX231 と RL78/F15 は、どちらも RS-CAN ベースなので類似点があります。

RS-CAN/RS-CAN Lite では、受信バッファで受信した際は割り込みが使えないので、受信処理は「受信 FIFO」を使用しています。（送信は、送受信 FIFO を使っています）

3.2. CAN のクロックと Tq

CAN は、本キットでは 1Mbps で動作させています。（定義ファイルを変更すれば、500kbps, 250kbps, 125kbps に変更可能です）

CAN は、ある程度高速でタイミングにシビアな系なので、クロックのベースは周波数精度の良い、水晶振動子ベースのクロックを使用しています。

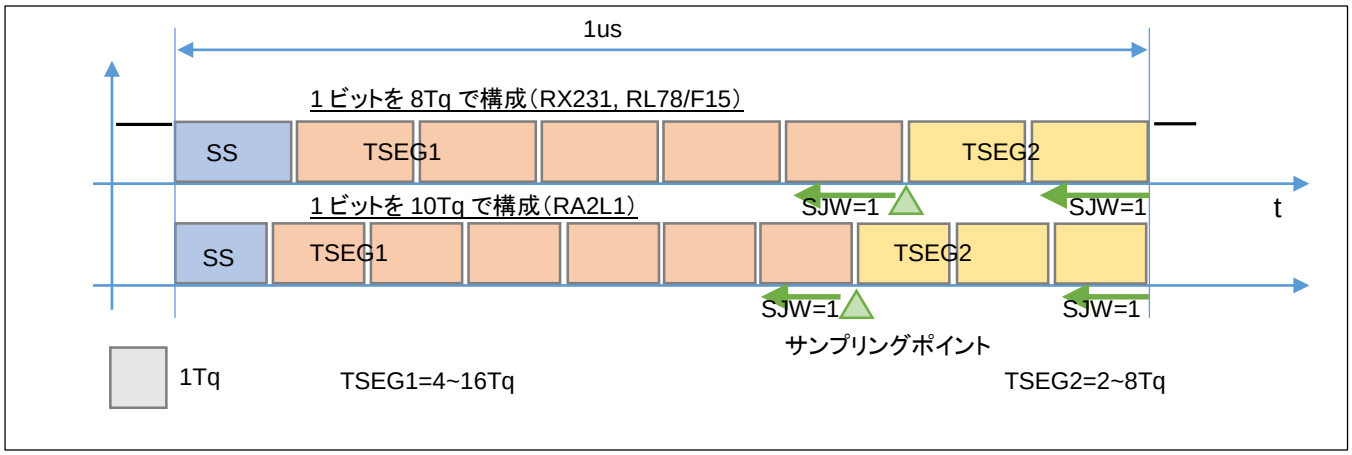


図 3-1 CAN データの 1 ビット

CAN の 1 ビットデータは、複数の Tq（Time Quantum, データの 1 ビットより短いタイミング基準）で構成されます。
1 ビットは、8Tq~25Tq で設定する必要があり、TSEG1>TSEG2>=SJW を満たす必要があります。
1 ビットを 1us とするためには、8Tq で 1 ビットを構成する場合、1Tq=125ns（8MHz のクロック源が必要）となり、10Tq で 1 ビットを構成する場合、1Tq=100ns（10MHz のクロック源が必要）となります。

※サンプリングポイント=(1+TSEG1)/(1+TSEG1+TSEG2)
※SJW:リシンクロナイゼーション ジャンプ幅（セグメントを延長・短縮する補正幅）

ー搭載クロックと CAN クロックー

CPU タイプ	搭載 水晶振動子 (XTAL)	PLL 逓倍 (fPLL)	CPU クロック	CAN クロック (fCAN)
RX231	8MHz	8/2×13.5 (=54MHz)	54MHz	XTAL =8MHz
RL78/F15	8MHz	8×8 (=64MHz)	32MHz	fPLL/2/2 =16MHz
RA2L1	20MHz	なし	48MHz	XTAL =20MHz

CAN クロックは、RX231 では 8MHz（分周比 1, 1Tq=125ns）、RL78/F15 では 16MHz（分周比 2, 1Tq=125ns）、RA2L1 では、20MHz（分周比 2, 1Tq=100ns）としています。

・本プログラムでの設定値（1Mbps 設定時）

マイコン種	クロックソース fCAN	分周比 [BRP] （分周後の クロック）	1Tq	TSEG1	TSEG2	SJW	サンプル ポイント
RX231	8MHz	1(8MHz)	125ns	5	2	1	75%
RL78/F15	16MHz	2(8MHz)	125ns	5	2	1	75%
RA2L1	20MHz	2(10MHz)	100ns	6	3	1	70%

※500kbps, 250kbps, 125kbps 設定時は分周比 BRP の値を変更し、8Tq(RX231, RL78/F15), 10Tq(RA2L1)で 1 ビットを構成するところは変更しない

3.3. ソースコード構成

can フォルダ内

・RX231(RS-CAN)

ファイル名	説明
can_board_setting.h	ボードに依存する設定（クロックや使用端子の設定）
can_operation.h	通信速度と、標準/拡張フォーマットの設定
rscan.c	CAN の通信処理
rscan.h	↑ヘッダファイル

・RL78/F15(RS-CAN Lite)

ファイル名	説明
can.c	CAN の ch に依存しない処理
can.h	↑ヘッダファイル
can_board_setting.h	ボードに依存する設定（クロックや使用端子の設定）
can_ch0.c	ch0 に依存する処理
can_ch0.h	↑ヘッダファイル
can_ch1.c	ch1 に依存する処理（本プログラムでは未使用）
can_ch1.h	↑ヘッダファイル

ファイル名	説明
can_intr.c	割り込みに関する処理
can_operation.h	通信速度と、標準/拡張フォーマットの設定

・RA2L1(CAN)

ファイル名	説明
can.c	CAN の ch に依存しない処理
can.h	↑ヘッダファイル
can_board_setting.h	ボードに依存する設定（クロックや使用端子の設定）
can_ch0.c	ch0 に依存する処理
can_ch0.h	↑ヘッダファイル
can_operation.h	通信速度と、標準/拡張フォーマットの設定

3.4. CAN 初期設定

・RX231

```
//CAN 初期化
can_init();          受信は RX-FIFO(0)を使う   拡張フォーマット   データフレーム   受信対象の CAN-ID

//CAN 受信設定       ID=0x10?のデータを受信する設定
can_receive_rule_set(0, CAN_RULE_RXFIFO0, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_1_ID1);
can_receive_rule_set(1, CAN_RULE_RXFIFO0, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_2_ID1);
can_receive_rule_set(2, CAN_RULE_RXFIFO0, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_3_ID1);
can_receive_rule_set(3, CAN_RULE_RXFIFO0, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_4_ID1);

                     ID=0x20?のデータを受信する設定
can_receive_rule_set(4, CAN_RULE_RXFIFO0, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_1_ID2);
can_receive_rule_set(5, CAN_RULE_RXFIFO0, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_2_ID2);
can_receive_rule_set(6, CAN_RULE_RXFIFO0, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_3_ID2);
can_receive_rule_set(7, CAN_RULE_RXFIFO0, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_4_ID2);

                     自ボードに対するリモートフレームを受信する設定
can_receive_rule_set(8, CAN_RULE_RXFIFO0, CAN_ID_FORMAT_EID, CAN_REMOTE_FRAME, CAN_ID2);

can_start();         CAN_ID2 は HSB_CAN_MULTI_1 向けにビルドした場合は 0x201
```

```
#define HSB_CAN_MULTI_1_ID1  0x101  //自ボードの情報を定期的にデータ送信に使用
#define HSB_CAN_MULTI_1_ID2  0x201  //自ボードに対する外部からの要求に使用

#define HSB_CAN_MULTI_2_ID1  0x102
#define HSB_CAN_MULTI_2_ID2  0x202

#define HSB_CAN_MULTI_3_ID1  0x103
#define HSB_CAN_MULTI_3_ID2  0x203

#define HSB_CAN_MULTI_4_ID1  0x104
#define HSB_CAN_MULTI_4_ID2  0x204
```

CAN の受信は、

・ID=0x101~0x104 で流れているデータフレームを受信

・ID=0x201~0x204 で流れているデータフレームを受信

・自分自身の ID(HSB_CAN_MULTI_1 であれば 0x201）向けに送られてきたリモートフレームを受信

の受信ルールを定義しています。(逆に言えば、上記以外の ID を持つデータ、自分自身の ID 以外のリモートフレームは、受信しないという設定です。)

・RL78/F15

```
//CAN 初期化
can_reset();
can0_init();
receive_rule_conf();          受信は RX-FIFO(0)を使う

//CAN 受信設定          ID=0x10?のデータを受信する設定
can0_receive_rule_set(0, CAN_RULE_RXFIFO0, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_1_ID1);
can0_receive_rule_set(1, CAN_RULE_RXFIFO0, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_2_ID1);
can0_receive_rule_set(2, CAN_RULE_RXFIFO0, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_3_ID1);
can0_receive_rule_set(3, CAN_RULE_RXFIFO0, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_4_ID1);

                      ID=0x20?のデータを受信する設定
can0_receive_rule_set(4, CAN_RULE_RXFIFO0, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_1_ID2);
can0_receive_rule_set(5, CAN_RULE_RXFIFO0, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_2_ID2);
can0_receive_rule_set(6, CAN_RULE_RXFIFO0, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_3_ID2);
can0_receive_rule_set(7, CAN_RULE_RXFIFO0, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_4_ID2);

                      自ボードに対するリモートフレームを受信する設定
can0_receive_rule_set(8, CAN_RULE_RXFIFO0, CAN_ID_FORMAT_EID, CAN_REMOTE_FRAME, CAN_ID2);

//CAN 動作モード
can_operate();
can0_operate();
intr_setup();
```

RL78/F15 は、CAN を 2ch サポートしているので、全体のリセット(can_reset())と CAN0 の初期化(can0_init)に分かれていたり、RX231 とは多少相違点があります。

reveice_rule_conf()は、合計で 40 個あるバッファを CAN ch0/CAN ch1 にどう割り振るかの設定、intr_setup()は、CAN の割り込みの設定を行っています。

・RA2L1

```
//CAN 初期化
can0_init();

//CAN の割り込み設定
intr_priority(INTR_CAN0_RXM, INTR_PRI1);
intr_priority(INTR_CAN0_TXM, INTR_PRI1);
intr_enable(INTR_CAN0_RXM);
intr_enable(INTR_CAN0_TXM);

//CAN 受信設定(MB0~MB8)
can0_mb_set_receive(0, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_1_ID1);
can0_mb_set_receive(1, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_2_ID1);
can0_mb_set_receive(2, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_3_ID1);
can0_mb_set_receive(3, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_4_ID1);

can0_mb_set_receive(4, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_1_ID2);
can0_mb_set_receive(5, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_2_ID2);
can0_mb_set_receive(6, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_3_ID2);
can0_mb_set_receive(7, CAN_ID_FORMAT_EID, CAN_DATA_FRAME, HSB_CAN_MULTI_4_ID2);

can0_mb_set_receive(8, CAN_ID_FORMAT_EID, CAN_REMOTE_FRAME, CAN_ID2);

//メールボックス 9~19 は予約


//CAN 送信設定
can0_mb_set_send(CAN_SEND_MB_1)      //自分自身のボードから発信(ID=0x10X)（MB=20 で設定）
can0_mb_set_send(CAN_SEND_MB_2);      //リモートフレームに対する返答(ID=0x20X)（MB=21 で設定）
can0_mb_set_send(CAN_SEND_MB_3);      //SCI 経由で受信した送信要求を処理（MB=22 で設定）
```

割り込み優先度と割り込み有効化

受信はメールボックス 0~8 を使用

ID=0x10?のデータを受信する設定

ID=0x20?のデータを受信する設定

自ボードに対するリモートフレームを受信する設定

送信はメールボックス 20~22 を使用

RA2L1 では、0~8 のメールボックスを受信に設定しているところは、RX231, RL78/F15 と概ね同じです。送信時に使用するメールボックス(20~22)を、ここで定義している点は、違う点です。

3.5. CAN メッセージ構造体

can.h 内で、CAN メッセージ構造体を定義しています。

```
typedef struct{
   unsigned long id;
   unsigned char rtr;
   unsigned char ide;
   unsigned char dlc;
   unsigned char data[8];
   unsigned short ts;
} can_message;
```

CAN メッセージ構造体は、id, rtr, ide, dlc, data, ts をひとまとめにしたものです。送信関数や受信関数では、この構造体を使用しています。

3.6. CAN データ送信

・RX231

```
msg.id = CAN_ID1;                    HSB_CAN_MULTI_1 ボードでは、0x101 が入ります
msg.rtr = CAN_DATA_FRAME;
msg.ide = CAN_ID_FORMAT_EID;         定数値
msg.dlc = CAN_PACKET_SIZE;

msg.data[0] = (unsigned char)(g_m1_motor_duty & 0x00ff);
msg.data[1] = (unsigned char)((g_m1_motor_rotation_speed & 0xff00) >> 8);
msg.data[2] = (unsigned char)(g_m1_motor_rotation_speed & 0x00ff);

for(i=3; i<CAN_PACKET_SIZE; i++) msg.data[i] = 0x00;     CAN_PACKET_SIZE = 8
                                                         （送信は、3 バイト送る場合でも、残りを 0x00 埋めとして、8 バイト送信しています）
can_srfifo_send(msg);
```

HSB_CAN_MULTI_1 ボードが、CAN-ID=0x101 でモータの duty と回転数を送信する部分のプログラムコードです。

送信関数は、CAN のメッセージ構造体(msg)を引数に指定します。

msg.id(CAN-ID), msg.rtr（データフレーム/リモートフレーム区分）, msg.ide（拡張フォーマット/標準フォーマット区分）, msg.dlc（送信バイト数，リモートフレームの際に送信を要求するバイト数）, msg.data[]（CAN のデータ部分，最大 8 バイト）を代入後に送信関数を呼び出します。送信時は、タイムスタンプ ts は使用しません。（受信時には、ts にタイムスタンプ値が入ります。）

can_srfifo_send()が送信を行う関数で、送受信 FIFO を使って送信を行っています。

・RL78/F15

```
msg.id = CAN_ID1;
msg.rtr = CAN_DATA_FRAME;
msg.ide = CAN_ID_FORMAT_EID;
msg.dlc = CAN_PACKET_SIZE;

msg.data[0] = (unsigned char)(g_m1_motor_duty & 0x00ff);
msg.data[1] = (unsigned char)((g_m1_motor_rotation_speed & 0xff00) >> 8);
msg.data[2] = (unsigned char)(g_m1_motor_rotation_speed & 0x00ff);

for(i=3; i<CAN_PACKET_SIZE; i++) msg.data[i] = 0x00;

can0_srfifo_send(msg);
```

RL78/F15 の場合は、CAN ch0 から送信を行いたいので、送信関数が can0_srfifo_send()になっています。

・RA2L1

```
msg.id = CAN_ID1;
msg.rtr = CAN_DATA_FRAME;
msg.ide = CAN_ID_FORMAT_EID;
msg.dlc = CAN_PACKET_SIZE;

msg.data[0] = (unsigned char)(g_m1_motor_duty & 0x00ff);
msg.data[1] = (unsigned char)((g_m1_motor_rotation_speed & 0xff00) >> 8);
msg.data[2] = (unsigned char)(g_m1_motor_rotation_speed & 0x00ff);

for(i=3; i<CAN_PACKET_SIZE; i++) msg.data[i] = 0x00;

can0_send(CAN_SEND_MB_1, msg);
```

RA2L1 の場合、送信関数が can0_send()になっている（メールボックスを使用した送信関数）、使用するメールボックス番号を指定するという違いはありますが、処理内容はそれ程変わりません。

3.7. CAN データ受信

CAN のデータ受信は割り込みルーチンで処理されます。

・RX231

```
//CAN  受信割り込み

RSCAN.RFSTS0.BIT.RFIF = 0;//割り込みフラグクリア

while (RSCAN.RFSTS0.BIT.RFEMP != 1)
{
   ret = can_rxfifo0_receive(&msg);//データ受信        受信処理

   if (ret != -2)//can_fifo_receive がエラーではない場合                受信データを  g_can_recv_buf
   {                                                                    リングバッファに格納する
      //受信バッファ書き込みインデックスを進める
      g_can_recv_buf_index1++;
      if (g_can_recv_buf_index1 >= CAN_RECV_BUF_SIZE) g_can_recv_buf_index1 = 0;

      //受信バッファに格納
      g_can_recv_buf[g_can_recv_buf_index1] = msg;

      if (g_can_recv_buf_index1 == g_can_recv_buf_index2)          //書き込みインデックスが読み出しインデックスを超えた
      {
         g_can_recv_buf_index2 = (unsigned short)(g_can_recv_buf_index1 + 1);
         if (g_can_recv_buf_index2 >= CAN_RECV_BUF_SIZE) g_can_recv_buf_index2 = 0;
         //この時点で未読み出しのデータは捨てられるが、最新のデータは未読み出しで有効とする
         g_can_recv_buf_override = 1;
      }
   }
}
```

CAN の受信割り込みが立つと、上記関数が実行され、CAN データの受信と、受信データをリングバッファに保存する処理が行われます。リングバッファに保存されたデータは、任意のタイミングで読み出しが可能です。

・RL78/F15

```
//CAN  受信割り込み

RSCAN.RFSTS0.BIT.RFIF = 0;//割り込みフラグクリア

while(1)
{
   //RXFIFO0 の読み出し
   while((RFSTS0 & 0x0001) == 0)
   {
      //未読データあり

      ret = can_rxfifo_receive(0, &msg);//RXFIFO(0)の受信処理        受信処理

      if (ret != -2)//can_fifo_receive がエラーではない場合                受信データを  g_can_recv_buf
      {                                                                    リングバッファに格納する
         can_receive_buf_store(msg);
      }
```

RL78/F15 でも同様の処理内容となります。

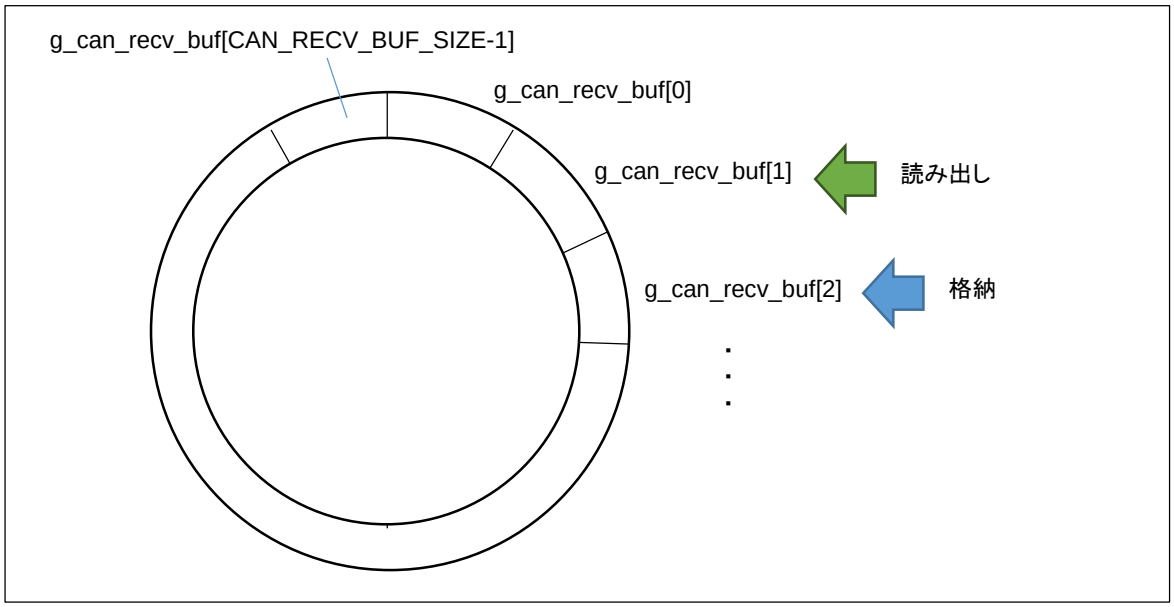
・RA2L1

```
//CAN 受信割り込み
R_ICU->IELSR_b[INTR_CAN0_RXM].IR = 0;       //IR フラグクリア

for (i=0; i<32; i++)        32 個のメールボックスをスキャンして、新しいメッセージが到着しているかどうかを確認
{
      if (R_CAN0->MCTL_RX_b[i].NEWDATA != 1) continue;

      ret = can0_receive((unsigned char)i, &msg);     受信処理

      if (ret != -2)                                  //can_fifo_receive がエラーではない場合    受信データを  g_can_recv_buf
      {                                                                                          リングバッファに格納する
         （中略）
      }
}
```

RA2A1 では、メールボックスに新しいメッセージが到着しているかをスキャンしています。（本プログラムでは、メールボックス番号 0~8 を受信に設定しているので、最低 0~8 のメールボックス番号を確認すれば良い。上記のプログラムでは、全てのメールボックス番号をチェックしています。）

どのマイコン種のプログラムでも、受信データは
g_can_recv_buf[]
という、バッファ（リングバッファ）に保存されます。

リングバッファは、



バッファの最後のインデックス(CAN_RECV_BUF_SIZE-1)まで到達した場合は、再度バッファのインデックスが(0)になる、リング上に連なったバッファです。データ受信のタイミングでデータ格納が行われ、格納インデックスが進む。読み出しを行った際は読み出しのインデックスが進み、格納インデックス=読み出しインデックスとなった時点で全データを読み出した（バッファに溜まっているデータなし）というイメージです。

データの読み出しを行う関数は、

can_read_data(&msg);

で、リングバッファ(g_can_recv_buf[])に保存されているデータを CAN のメッセージ構造体(msg)に取得可能です。

```
int ret;
can_message msg;

while(1)
{
   ret = can_read_data(&msg);
   if (ret != 2)
   {
      //CAN 受信データあり
      （受信データの処理）
   }
}
```

（受信データの処理）部分で、

msg.id　受信した CAN-ID

msg.rtr　データフレームの場合は 0, リモートフレームの場合は 1

msg.ide　定義で拡張 ID を設定しているので、1

msg.dlc　CAN_PACKET_SIZE 定数を 8 に

msg.data[8]　受信したデータ

msg.ts タイムスタンプ（受信時に受信側のマイコンが付与する 2 バイトのタイムスタンプ）
の各データを使用して処理を行います。

3.8. 定数設定

can_operation.h

```
//速度設定
#define CAN_SPEED  CAN_BPS_1M

//いずれかを設定
//CAN_BPS_1M  //1M bps
//CAN_BPS_500K  //500k bps
//CAN_BPS_250K  //250k bps
//CAN_BPS_125K  //125k bps

//ID 設定
#define ID_TYPE  CAN_ID_FORMAT_EID

//いずれかを設定
//CAN_ID_FORMAT_SID  //標準 ID(11bit)フォーマットを使用
//CAN_ID_FORMAT_EID  //拡張 ID(29bit)フォーマットを使用
```

can_operation.h 内に、速度設定と、拡張/標準フォーマットのどちらを使用するかの記載があります。

通信速度を変更したい場合は、上記ファイルを編集後、プログラムをビルドしてください。

3.9. 関数

can_init [RX231]

概要:初期化関数

宣言:

int can_init(void)

説明:

・モジュールストップ解除
・端子設定
・通信設定

を行います

引数:

なし

戻り値:

0: 正常終了

can_reset [RL78/F15]

概要:初期化関数

宣言:

int can_reset(void)

説明:

・モジュールストップ解除
・クロック設定
・CAN リセットモード遷移

を行います

引数:

なし

戻り値:

0: 正常終了

can*n*_init (n=0~1) [RL78/F15]

概要:チャネル初期化関数

宣言:

int can0_init(void) [ch0 向け]

int can1_init(void) [ch1 向け] ※本プログラムでは未使用

説明:

・チャネルリセットモード遷移

・端子設定

・通信速度設定

を行います

引数:

なし

戻り値:

0: 正常終了

can_reset()の実行後、can*n*_init()を実行してください。

can_reset()は全体の処理、can*n*_init()は ch 毎の処理です。

can*n*_init (n=0) [RA2L1]

概要:初期化関数

宣言:

int can0_init(void) [ch0 向け]

説明:

・モジュールストップ解除

・端子設定

・通信設定

を行います

引数:

なし

戻り値:

0: 正常終了

※RA2L1 は ch0 のみですが、CAN モジュールは複数 ch をサポートしていますので ch0 の初期化関数になっています

can_receive_rule_set [RX231]

概要:受信ルール設定関数

宣言:

int can_receive_rule_set(unsigned char num, unsigned char mode, unsigned char ide, unsigned char rtr, unsigned long id);

説明:

・受信ルール設定

を行います

引数:

num: 受信ルール番号(0~15)

mode: 受信先

CAN_RULE_BUF(0x01) 受信バッファで受信

CAN_RULE_FIFO0(0x02) 受信 FIFO(0)で受信

CAN_RULE_FIFO1(0x04) 受信 FIFO(1)で受信

CAN_RULE_SR_FIFO(0x08) 送受信 FIFO で受信

ide: 標準／拡張フォーマット区分

CAN_ID_FORMAT_SID(0) 標準フォーマット(ID=11bit)

CAN_ID_FORMAT_EID(1) 拡張フォーマット(ID=29bit)

rtr: データフレーム／リモートフレーム区分

CAN_DATA_FRAME(0) データフレーム（データ送信）

CAN_REMOTE_FRAME(1) リモートフレーム（相手にデータ要求）

id: ID を指定

戻り値:

0: 正常終了

-1: 引数エラー

-2: 受信ルール設定不可の動作モード

RSCAN モジュール系では本関数で「受信ルール」の設定を行います。受信ルールにマッチしたデータが、本関数で指定した受信先にコピーされるという動作です。

receive_rule_conf [RL78/F15]

概要: 受信ルール数設定関数

宣言:

int receive_rule_conf(void)

説明:

・受信ルール数の設定

を行います

can*n*_init() 後、can*n*_receive_rule_set()実行前に実行してください。

引数:

なし

戻り値:

0: 正常終了

CAN0 と CAN1 の両方有効にした場合は、CAN0/CAN1 それぞれ 20 ルールを設定します(*1)。どちらか一方の ch のみ有効化した場合は、有効化した ch 側に 40 ルールを設定します。

(*1)マイコンの機能としては、トータル 40 という制約があるだけで、CAN0 側に 30, CAN1 側に 10 ルールといった設定も可能です

※本プログラムでは ch0 のみ有効化しているので、ch0 に 40 ルールを適用します

can*n*_receive_rule_set (n=0~1) [RL78/F15]

概要:受信ルール設定関数

宣言:

int can0_receive_rule_set(unsigned char num, unsigned char mode, unsigned char ide, unsigned char rtr, unsigned long id);

int can1_receive_rule_set(unsigned char num, unsigned char mode, unsigned char ide, unsigned char rtr, unsigned long id);

説明:

・受信ルール設定

を行います

引数:

num: 受信ルール番号

mode: 受信先

CAN_RULE_BUF(0x0)　　受信バッファで受信

CAN_RULE_RXFIFO0(0x0001)　　受信 FIFO(0)で受信

CAN_RULE_RXFIFO1(0x0002)　　受信 FIFO(1)で受信

CAN_RULE_RXFIFO2(0x0004)　　受信 FIFO(2)で受信

CAN_RULE_RXFIFO3(0x0008)　　受信 FIFO(3)で受信

CAN_RULE_SRFIFO0(0x0010)　送受信 FIFO(0)で受信

CAN_RULE_SRFIFO1(0x0020)　　送受信 FIFO(1)で受信

ide: 標準／拡張フォーマット区分

CAN_ID_FORMAT_SID(0)　標準フォーマット(ID=11bit)

CAN_ID_FORMAT_EID(1)　拡張フォーマット(ID=29bit)

rtr: データフレーム／リモートフレーム区分

CAN_DATA_FRAME(0)　　データフレーム（データ送信）

CAN_REMOTE_FRAME(1)　リモートフレーム（相手にデータ要求）

id: ID を指定

戻り値:

0: 正常終了

-1: 引数エラー

-2: 受信ルール設定不可の動作モード

補足:

num は、1ch 使用時 0-39, 2ch（CAN0 及び CAN1）使用時 0-19 を指定可能です

can*n*_mb_set_receice　　(n=0)　[RA2L1]

概要:メールボックス設定関数

宣言:

int can0_mb_set_receive(unsigned char mb, unsigned char ide, unsigned char rtr, unsigned long id) [ch0 向け]

説明:

・メールボックスを受信用として設定

を行います

引数:

mb: メールボックス番号(0~31)

ide: 標準／拡張フォーマット区分

CAN_ID_FORMAT_SID(0) 標準フォーマット(ID=11bit)

CAN_ID_FORMAT_EID(1) 拡張フォーマット(ID=29bit)

rtr: データフレーム／リモートフレーム区分

CAN_DATA_FRAME(0) データフレーム（データ送信）

CAN_REMOTE_FRAME(1) リモートフレーム（相手にデータ要求）

id: ID を指定

戻り値:

0: 正常終了

-1: 引数エラー

-2: 送信、受信アボート処理失敗

can*n*_mb_set_send (n=0) [RA2L1]

概要:メールボックス設定関数

宣言:

int can0_mb_set_send(unsigned char mb) [ch0 向け]

説明:

・メールボックスを送信用として設定

を行います

引数:

mb: メールボックス番号(0~31)

戻り値:

0: 正常終了

-1: 引数エラー

-2: 送信、受信アボート処理失敗

can_receive_rule_set [RX231]

can0_receive_rule_set [RL78/F15]

can0_mb_set_receive [RA2L1]

は、受信ルールの設定関数です。これらの関数で、受信したい

・CAN-ID

・IDE（拡張/標準フォーマット区分）

・RTR（データフレーム/リモートフレーム区分）
を指定します。上記関数は、CAN-ID, IDE, RTR が全て一致した時にデータを受信する様に設定する関数です。
※マイコンの機能としては、CAN-ID の部分一致や IDE=0/1 両方を１つのルールでマッチさせるという事も可能です。本プログラムでは、CAN-ID, IDE, RTR は受信ルール設定関数で設定した値と、完全一致としています。（DLC は比較対象から外しています。）（CAN-ID, IDE, RTR の部分一致や比較しないルール、DLC 一致のルールを生成する場合は、関数の改変が必要です。）

can_start [RX231]

概要：動作モード変更関数

宣言：

int can_start(void)

説明：

・CAN モジュールを動作モードに移行する
を行います

引数： なし

戻り値：

0: 正常終了
-2: 動作モード変更 NG

RSCAN モジュール系では、受信ルール設定完了後、本関数を呼び出す事により、CAN モジュールを動作モードに変更します。

can_operate [RL78/F15]
can*n*_operate (n=0~1) [RL78/F15]

概要：動作モード変更関数

宣言：

int can_operate(void)
int can0_operate(void)
int can1_operate(void)

説明：

・CAN モジュールを動作モードに移行する処理
・チャネルを動作モードに移行する処理

を行います

引数: なし

戻り値:

0: 正常終了

-2: 動作モード変更 NG

can_operate 実行（CAN モジュール全体を動作モードに変更）後 ch 毎の動作モードを変更(can*n*_operate)を実行してください。

intr_setup [RL78/F15]

概要:割り込み設定関数

宣言:

int intr_setup(void)

説明:

・割り込みの設定

・割り込みの有効化

を行います

引数: なし

戻り値:

0: 正常終了

can_srfifo_send [RX231]

概要:データ送信関数

宣言:

int can_srfifo_send(can_message msg)

説明:

・送受信 FIFO を使用してデータの送信

を行います

引数:

msg.ide: 標準／拡張フォーマット区分

CAN_ID_FORMAT_SID(0) 標準フォーマット(ID=11bit)

CAN_ID_FORMAT_EID(1) 拡張フォーマット(ID=29bit)

msg.rtr: データフレーム／リモートフレーム区分

CAN_DATA_FRAME(0) データフレーム（データ送信）

CAN_REMOTE_FRAME(1) リモートフレーム（相手にデータ要求）

msg.id: 送信する ID を指定します

msg.dlc: 送信バイト数を指定します(1~8)

msg.data[ ]: 送信するデータを指定します

戻り値:

0: 正常終了

-1: 引数チェックエラー

-2: FIFO フル使用中

can*n*_srfifo_send (n=0~1) [RL78/F15]

概要:データ送信関数

宣言:

int can0_srfifo_send(can_message msg)

int can1_srfifo_send(can_message msg)

説明:

・送受信 FIFO を使用してデータの送信
を行います

引数:

msg.ide: 標準／拡張フォーマット区分

CAN_ID_FORMAT_SID(0) 標準フォーマット(ID=11bit)

CAN_ID_FORMAT_EID(1) 拡張フォーマット(ID=29bit)

msg.rtr: データフレーム／リモートフレーム区分

CAN_DATA_FRAME(0) データフレーム（データ送信）

CAN_REMOTE_FRAME(1) リモートフレーム（相手にデータ要求）

msg.id: 送信する ID を指定します

msg.dlc: 送信バイト数を指定します(1~8):1~8 バイト

msg.data[ ]: 送信するデータを指定します

戻り値:

0: 正常終了

-1: 引数チェックエラー

-2: FIFO フル

補足:

CAN ch0 は SRFIFO(0)で送信を行います

CAN ch1 は SRFIFO(1)で送信を行います

cann_send (n=0) [RA2L1]

概要:データ送信関数

宣言:

int can0_send(unsigned char mb, can_message msg)

説明:

・データの送信

を行います

引数:

mb: 使用するメールボックスを指定します(0~31)

msg.ide: 標準／拡張フォーマット区分

CAN_ID_FORMAT_SID(0) 標準フォーマット(ID=11bit)

CAN_ID_FORMAT_EID(1) 拡張フォーマット(ID=29bit)

msg.rtr: データフレーム／リモートフレーム区分

CAN_DATA_FRAME(0) データフレーム（データ送信）

CAN_REMOTE_FRAME(1) リモートフレーム（相手にデータ要求）

msg.id: 送信する ID を指定します

msg.dlc: 送信バイト数を指定します(1~8)

msg.data[ ]: 送信するデータを指定します

戻り値:

0: 正常終了

-1: 引数チェックエラー

-2: レジスタ値設定タイムアウト

-3: メールボックスが受信に設定されている

-4: メールボックスが現在送信中

can_rxfifo0_receive [RX231]

概要:受信関数

宣言:

int can_rxfifo0_receive(can_message *msg)

説明:

・受信 FIFO(0)を使用したデータの受信

を行います

引数:

msg->ide: 標準／拡張フォーマット区分を格納(unsigned char)

CAN_ID_FORMAT_SID(0) 標準フォーマット(ID=11bit)

CAN_ID_FORMAT_EID(1) 拡張フォーマット(ID=29bit)

msg->rtr: データフレーム／リモートフレーム区分を格納(unsigned char)

CAN_DATA_FRAME(0) データフレーム（データ送信）

CAN_REMOTE_FRAME(1) リモートフレーム（相手にデータ要求）

msg->id: 受信した ID を格納(unsigned long)

msg->data[ ]: 受信したデータを格納（最大 unsigned char [8]）

msg->ts: データ受信時のタイムスタンプ(unsigned short)

戻り値:

0: 受信データなし（DLC=0 のデータはデータなしとして扱う）

1~8: 受信したデータのバイト数

-1: 引数チェックエラー

-2: 受信 FIFO にデータなし

can_rxfifo_receive [RL78/F15]

概要:受信 FIFO 受信関数

宣言:

int can_rxfifo_receive(unsigned char fifo_no, can_message *msg)

説明:

・受信 FIFO を使用したデータの受信

を行います

引数:

fifo_no: 受信 FIFO 番号(0~3)

msg->ide: 標準／拡張フォーマット区分を格納 (unsigned char)

CAN_ID_FORMAT_SID(0) 標準フォーマット(ID=11bit)

CAN_ID_FORMAT_EID(1) 拡張フォーマット(ID=29bit)

msg->rtr: データフレーム／リモートフレーム区分を格納(unsigned char)

CAN_DATA_FRAME(0) データフレーム（データ送信）

CAN_REMOTE_FRAME(1) リモートフレーム（相手にデータ要求）

msg->id: 受信した ID を格納(unsigned long)

msg->data: 受信したデータを格納（最大 unsigned char [8]）

msg->ts: データ受信時のタイムスタンプ(unsigned short)

戻り値:

0: 受信データなし（DLC=0 のデータはデータなしとして扱う）

1~8: 受信したデータのバイト数 [cann_rxfifo_receive]

0x2x: メッセージロストフラグが立っている（受信関数呼び出し前に破棄されたデータあり）

-1: 引数チェックエラー

-2: 受信 FIFO にデータなし

cann_receive (n=0) [RA2L1]

概要:受信関数

宣言:

int can0_receive(unsigned char mb, can_message *msg)

説明:

・データの受信

を行います

引数:

mb: メールボックス(0~31)

msg->ide: 標準／拡張フォーマット区分を格納 (unsigned char)

CAN_ID_FORMAT_SID(0) 標準フォーマット(ID=11bit)

CAN_ID_FORMAT_EID(1) 拡張フォーマット(ID=29bit)

msg->rtr: データフレーム／リモートフレーム区分を格納 (unsigned char)

CAN_DATA_FRAME(0) データフレーム（データ送信）

CAN_REMOTE_FRAME(1) リモートフレーム（相手にデータ要求）

msg->id: 受信した ID を格納 (unsigned long)

msg->data[ ]: 受信したデータを格納（最大 unsigned char [8]）

msg->ts: データ受信時のタイムスタンプ(unsigned short)

戻り値:

0: 受信データなし(DLC=0 のデータはデータなしとして扱う)
1~8: 受信したデータのバイト数
0x1x: (b4=1)受信メールボックス更新中
0x2x: (b5=1)オーバライドフラグが立っている(受信操作前に上書きされたメッセージあり)
-1: 引数チェックエラー
-2: レジスタ設定タイムアウト
-3: メールボックスが受信用に設定されていない

CAN の初期化や送受信関数は、マイコン種に依存する関数名、引数となります。以下、マイコン種に依存しない関数を示します。

can_read_data

概要:受信バッファ読み出し関数

宣言:

int can_read_data(can_message *msg)

説明:

・リングバッファに保存されているデータの読み出し
を行います

引数:

msg->ide: 標準／拡張フォーマット区分 (unsigned char)
CAN_ID_FORMAT_SID(0) 標準フォーマット(ID=11bit)
CAN_ID_FORMAT_EID(1) 拡張フォーマット(ID=29bit)
msg->rtr: データフレーム／リモートフレーム区分(unsigned char)
CAN_DATA_FRAME(0) データフレーム(データ送信)
CAN_REMOTE_FRAME(1) リモートフレーム(相手にデータ要求)
msg->id: 受信した ID(unsigned long)
msg->data: 受信したデータ(最大 unsigned char [8])
msg->ts: データ受信時のタイムスタンプ(unsigned short)

戻り値:

0: 正常終了
1: データの読み出し前に上書きされたデータあり(読み出しは完了，残っているデータで一番古いものを返す)
2: 保存されているデータなし(全て読み出し済み)

can_read_data_size

概要:受信バッファ未読数返却関数

宣言:

int can_read_data_size(void)

説明:

・リングバッファに保存されているデータの数の読み出し
を行います

引数:

なし

戻り値:

0: 未読み出しのデータなし

>0: 受信バッファに溜まっているメッセージ数

can_read_buf_clear

概要:受信バッファクリア関数

宣言:

void can_read_buf_clear(void)

説明:

・リングバッファに保存されているデータのクリア
を行います(保存されているデータを全て読み出し済みと設定します)

引数:

なし

戻り値:

なし

4. LIN 通信

4.1. LIN 通信に使用しているマイコン搭載のモジュール

マイコン種により、LIN の部分のプログラムは大きく異なります。

マイコン種	LIN 通信に使用しているモジュール	備考
RX231	SCI12	拡張シリアル通信機能付きモジュール
RL78/F15	RLIN3	LIN をサポートしたモジュール
RA2L1	SCI0	通常の UART モジュール
RL78/G11 (HSB_LIN_COMM)	UART0	LIN-bus 対応

LIN 通信では、「ブレーク送信・ブレーク検出」「Sync フィールド受信」「(ID 受信，データ受信，チェックサム受信)チェックサム比較」が必要ですが、マイコン(モジュール)毎に、モジュールのハードウェアで対応している項目が異なりますので、ソフトウェアでどこまでカバーするかが異なってきます。

本プログラムでは、通信速度は 9,600bps に設定しています。(LIN の規格上は、通信速度~20kbps です。)

・ブレーク送信

マイコン種（モジュール）	モジュールレベルでのサポート	備考
RX231(SCI12)	○	専用タイマを使用して 13Tbit のブレーク送信が可能
RL78/F15(RLIN3)	◎	レジスタの書き込み(プログラム 1 行のみ)で完結
RA2L1(SCI0)	×	通信速度を変更して 0x00 送信
RL78/G11(UART0)	×	通信速度を変更して 0x00 送信

RA2L1, RL78/G11 では、ブレーク送信機能がないので、6,600bps(9,600×9/13)に速度設定を行い、0x00 を送信する事としています。6,600bps で 0x00 を送信した場合、6,600bps の 9bit 期間 L となり、9,600bps で 13bit 期間 L と同等の信号幅となります。

・ブレーク検出

マイコン種（モジュール）	モジュールレベルでのサポート	備考
RX231(SCI12)	○	専用タイマを使用してブレーク検出が可能
RL78/F15(RLIN3)	◎	モジュールレベルでブレーク検出が可能
RA2L1(SCI0)	×	フレーミングエラーでブレーク検出
RL78/G11(UART0)	○	汎用タイマと組み合わせてブレーク検出が可能

ブレーク検出は、13bit 期間の L パルスを受信して「信号の開始」を検出する処理です。RX231, RL78/G11 ではマイコンのタイマ機能と組み合わせる事で、ブレーク幅の検出を行っています。RA2L1 では、フレーミングエラー(本来異常である L 期間)で、ブレーク検出を行っています。

・Sync フィールド受信

SLAVE 設定時に Sync フィールドを使用して、送信側の通信速度に合わせるオートボーレート設定の可否に関して

マイコン種 （モジュール）	モジュールレベルで のサポート	備考
RX231(SCI12)	○	専用タイマを使用して通信速度検出が可能
RL78/F15(RLIN3)	◎	モジュールレベルでオートボーレート設定が可能
RA2L1(SCI0)	×	当該機能なし（本プログラムでは、通信速度固定で使用）
RL78/G11(UART0)	○	汎用タイマと組み合わせて通信速度検出が可能

4 種のマイコンの内、RA2L1 のみ、SLAVE 時のオートボーレート設定機能を実装していません。

その他のマイコンでは、SLAVE 時、MASTER 側が送信する Sync フィールドのパルス幅に応じて、受信レートを設定します。（RL78/F15 では~20kbps、RX231, RL78/G11 では 9,600bps±15%の通信レートを許容します。）
RL78/F15 ではモジュールレベルで受信レート設定が行われます。RX231, RL78/G11 では、タイマのカウント値から通信レートを計算して、ユーザプログラムで受信レートを設定しています。

・チェックサム比較

受信時に、ID とデータ、チェックサムを受信してデータ化けが無いかを確認する作業があります。

マイコン種 （モジュール）	モジュールレベルで のサポート	備考
RX231(SCI12)	×	ユーザプログラムでチェックサムの計算・比較
RL78/F15(RLIN3)	○	モジュールレベルでチェックサムの計算・比較が可能
RA2L1(SCI0)	×	ユーザプログラムでチェックサムの計算・比較
RL78/G11(UART0)	×	ユーザプログラムでチェックサムの計算・比較

RL78/F15 のみ、モジュールレベルでチェックサムの計算・比較を行います。その他のマイコンではユーザプログラム内で、チェックサムの計算と比較を行います。

LIN に関しては、RL78/F15 が LIN に対応した専用モジュールにより、ユーザ側で行わなければならない処理が圧倒的に少なくて済みます。

4.2. ソースコード構成

lin フォルダ内

ファイル名	説明
lin.c	LIN の通信処理
lin.h	↑ヘッダファイル
lin_operation.h	LIN の ID 設定等

4.3. LIN 初期設定

LIN は、通信途中で MASTER/SLAVE が入れ替わったりしない仕様なので、初期化時に MASTER/SLAVE を決める事としています。

・MASTER として初期化

```
lin_init(LIN_MASTER, LIN_MASTER_ID);       //LIN-MASTER
```

※RL78/F15 の場合は、LIN の ch0/ch1 がありますので、lin0_init()となります。

```
#define LIN_MASTER  0
#define LIN_MASTER_ID  0x30  //MASTER 動作時に使用する ID
```

・SLAVE として初期化

```
lin_init(LIN_SLAVE, LIN_SLAVE_ID_1);
lin_response_data_set(&data[0], LIN_RESPONSE_DATA_SIZE);
lin_slave_header_receive();
```

※RL78/F15 の場合は、LIN の ch0/ch1 がありますので、lin_init()→lin0_init(), lin_response_data_set()→lin0_response_data_set(), lin_slave_header_receive()→lin0_slave_header_receive()となります。

```
#define LIN_SLAVE  1
#define LIN_SLAVE_ID_1  0x31  //SLAVE 動作時に使用する ID
#define LIN_RESPONSE_DATA_SIZE  8
unsigned char data[8] = {0};
```

初期化関数(lin_init())で、MASTER/SLAVE 区分、及び LIN-ID を指定。

SLAVE の場合は、MASTER からヘッダ送信があった場合に応答する必要があるので、
lin_response_data_set() : ヘッダに自分自身の ID が含まれていた場合に送信する、レスポンスデータを設定
lin_slave_header_receive() : ヘッダの受信待機
を行っています。レスポンスデータは、ここでは仮値（0x00 を 8 個）としています。

LIN の送信データサイズは、基本的には 1~8 バイトですが、CAN と同様 8 バイト固定(LIN_RESPONSE_DATA_SIZE= 8)としています。

4.4. LIN メッセージ構造体

lin.h 内で、LIN メッセージ構造体を定義しています。

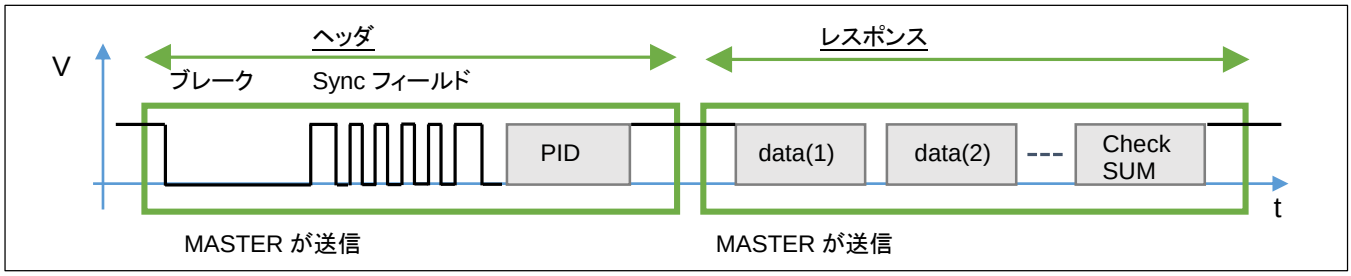
```
typedef struct{
   unsigned char id;
   unsigned char data[8];
   unsigned char sum;
   unsigned char size;
} lin_message;
```

LIN メッセージ構造体は、id, data, sum（チェックサム）, size（データサイズ）をひとまとめにしたものです。送信関数や受信関数では、この構造体を使用しています。

4.5. LIN ヘッダ送信

LIN のヘッダ送信は必ず MASTER 側が行います。

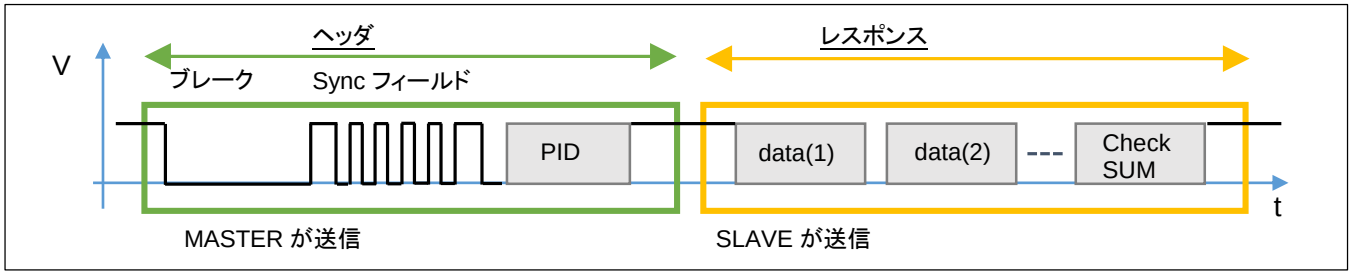
・MASTER レスポンス送信




```
lin_master_header_response_send(LIN_MASTER_ID, &g_lin_send_data[0], LIN_RESPONSE_DATA_SIZE);
```

```
#define LIN_MASTER_ID  0x30
#define LIN  RESPONSE  DATA  SIZE  8
```

・SLAVE レスポンス送信




```
lin_master_header_send(lin_send_id, LIN_RESPONSE_DATA_SIZE);
```

MASTER レスポンス送信は、MASTER からヘッダ＋レスポンスが送信されます。ヘッダに含まれる LIN-ID とデータ、送信バイト数を指定して、lin_master_header_response_send()関数を呼び出します。
（LIN が複数 ch 搭載されている、RL78/F15 では、lin0_master_header_response_send()関数）

SLAVE レスポンス送信では、MASTER はヘッダの送信のみ行います。この場合は、ヘッダに含まれる LIN-ID と、レスポンスとして期待するデータバイト数を指定して、lin_master_header_send()関数を呼び出します。
（LIN が複数 ch 搭載されている、RL78/F15 では、lin0_master_header_send()関数）

4.6. LIN レスポンス送信

SLAVE デバイスは、受信したヘッダに自分自身の ID が含まれている場合、レスポンス部分のデータを送信します。ブレークの検出と、ヘッダに含まれるデータの受信に関しては、割り込みで行っています。

SLAVE のレスポンス送信は、自分では送信するタイミングを決められない（ヘッダを受信した直後のタイミングに限られる）動作です。

そのため、レスポンスで送信するデータは予め設定を行っておく必要があり、その設定を行う部分が前出の LIN 初期化の部分で実行していた

```
lin_response_data_set(&data[0], LIN_RESPONSE_DATA_SIZE);
```

のコードです。このコードは任意のタイミングで実行する事が可能で、（自分宛ての）ヘッダを受信したタイミングで、本関数で設定したデータを送信します。（data[0]~data[7]に送信データを設定）

4.7. LIN レスポンス受信

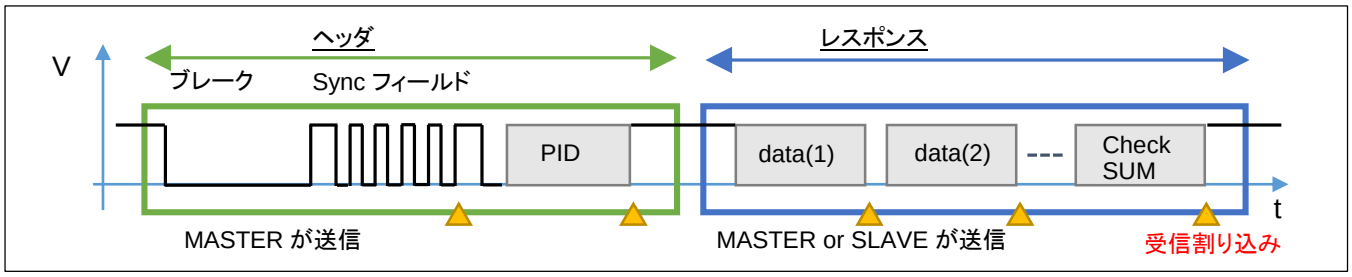
MASTER 側が、ヘッダに SLAVE の LIN-ID を埋め込んで送信した際（ヘッダのみの送信）、MASTER 側ではレスポンスデータを受信する必要があります。また、SLAVE 側で、ヘッダに含まれる LIN-ID が自分自身の ID と違っていた場合は、レスポンスデータの受信を行う様にしています。

SLAVE 設定時、レスポンスの送信、受信は割り込みでデータを受信した際に、ヘッダに含まれる LIN-ID を見て、送信なのか受信なのかを分岐で処理しています。

MASTER 設定時は、自分で（MASTER 側が）レスポンス送信を行わない場合は、レスポンス受信を行います。

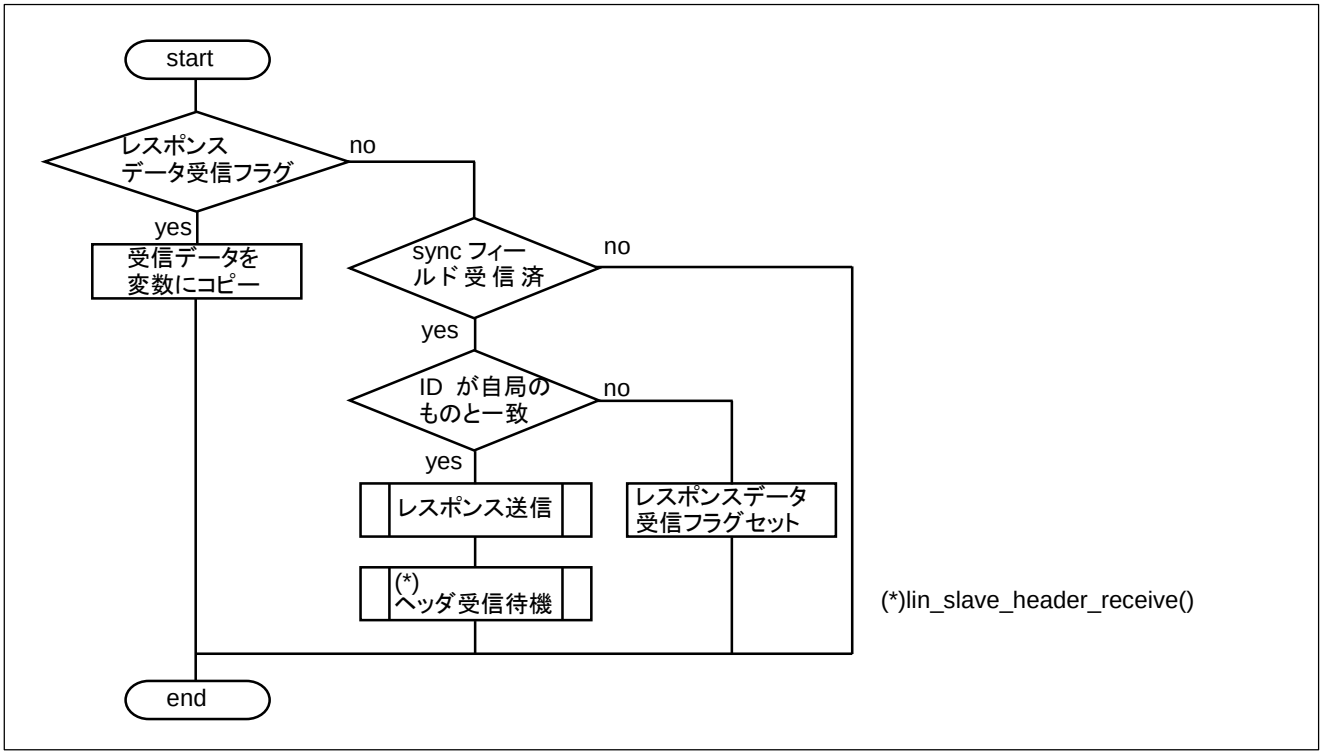
・RX231, RA2L1, RL78/G11 の場合

これらのマイコンでは、SCI(UART)の機能を使用して LIN データを処理しているので、多少処理が煩雑となります。



データ受信の際に、1 バイト毎に受信割り込みが発生します。

－SCI 受信割り込み（概略）－

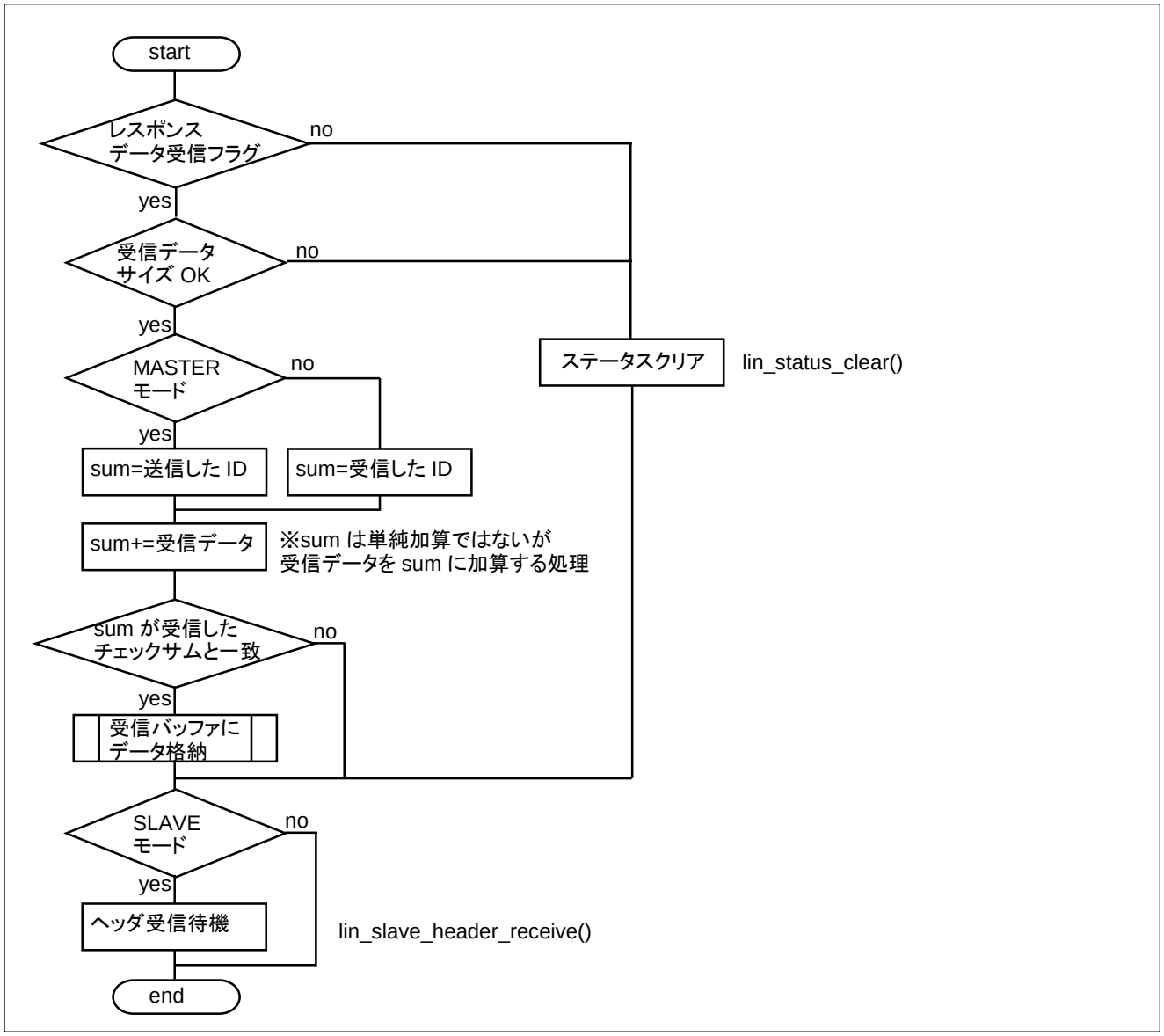


2 回目の受信割り込み（ID 受信）のタイミングで、ID が自局のもの（lin_linit()で設定した ID）と一致した場合は、レスポンス送信を行い、再びヘッダの受信待機に入ります。

不一致の場合、レスポンスデータ受信フラグを立てて、3 回目以降の受信割り込みの際に、受信したデータを変数にコピーします。

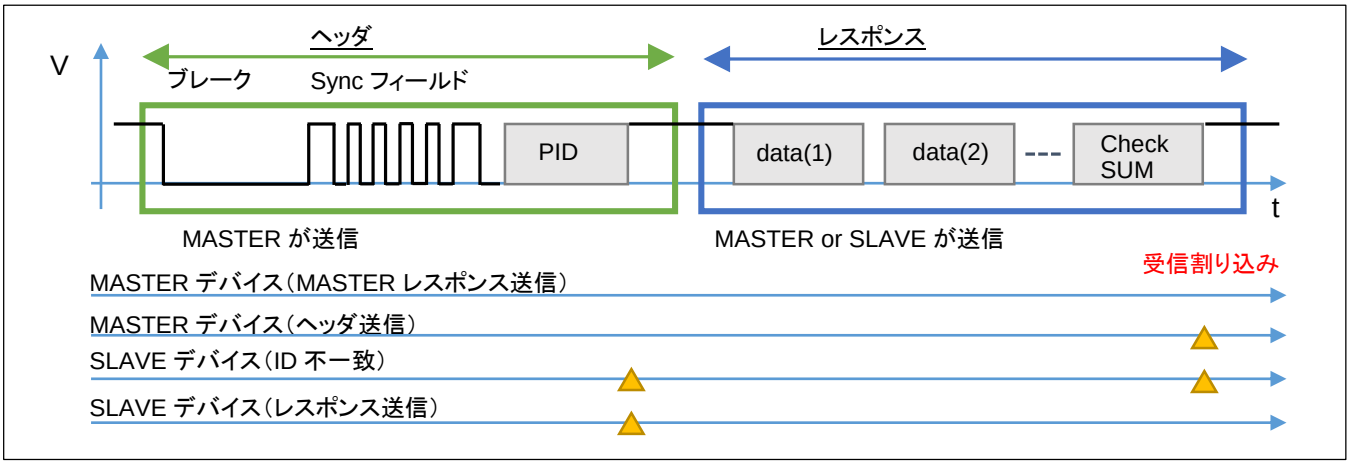
LIN の場合は、CAN とは異なり、受信する側はデータバイト数が判らないので、ヘッダ受信のタイミングでタイマを起動する事としています。（マイコン種毎に細かな部分は多少異なりますが、チェックサム含め最長 9 バイトデータの受信完了後のタイミングで割り込みが掛るタイマ）

ータイマ割り込み（概略）ー



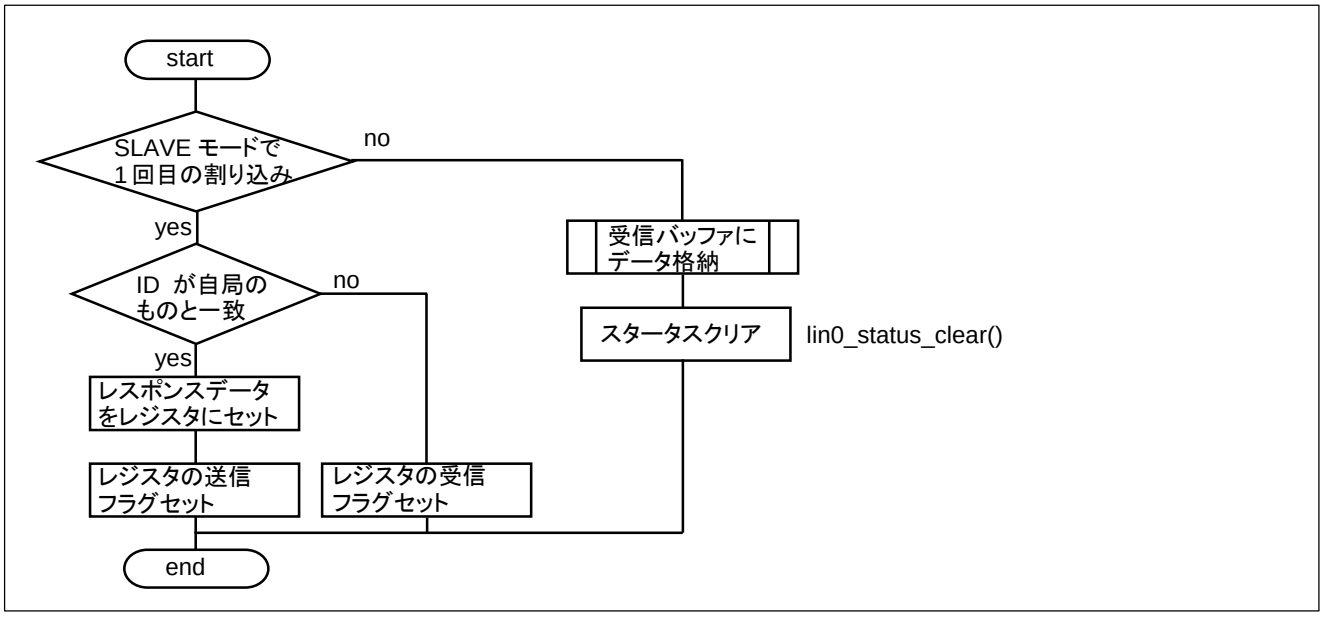
データ（最大 8 バイト）とチェックサムの受信が終わったタイミングで、変数にコピーしておいたデータを確認（データのサイズが妥当か、チェックサムが合っているか）し、問題が無ければ受信バッファにデータを格納します。受信バッファは、g_lin_recv_buf[]（RL78/F15 の場合は、g_lin0_recv_buf[]）で、この部分の処理は CAN と同様、リングバッファ構成の受信バッファにデータを保存する処理となります。

・RL78/F15 の場合



RL78/F15 の場合は、受信割り込みは、MASTER 側で SLAVE レスポンス送信の場合は、レスポンス受信のタイミングで 1 回。SLAVE 側はヘッダ受信後のタイミングと、自局が送信しない場合レスポンス受信後のタイミングの 2 回となります。

－LIN 受信割り込み（概略）－



RX231, RA2L1, RL78/G11 の場合は、チェックサムの計算、比較はユーザプログラムで行っていますが、RL78/F15 の場合は、チェックサム値が合わない場合は受信（完了）割り込みに飛んできません。（エラーという事で、ステータス割り込みに飛びます。）RL78/F15 では、LIN のモジュール（ハードウェア）で、チェックサムの比較が行われます。

LIN のプログラムを作成する上では、LIN モジュールを内蔵している RL78/F15 がユーザ側で行う処理が少なく、プログラムは非常に楽になります。

レスポンス受信で、保存されたデータの読み出しを行う関数は

lin_read_data(&msg);

で、リングバッファ(g_lin_recv_buf[])に保存されているデータを LIN のメッセージ構造体(msg)に取得可能です。

```
int ret;
lin_message msg;

while(1)
{
   ret = lin_read_data(&msg);        ※RL78/F15 では、lin0_read_data(&msg);となります
   if (ret != 2)
   {
      //LIN 受信データあり
      （受信データの処理）
   }
}
```

（受信データの処理）部分で、
msg.id　受信した LIN-ID
msg.data[8]　受信したデータ
msg.sum　受信したチェックサム
msg.size　受信したデータサイズ
を使用して、各種処理を行う事が可能です。

なお、レスポンス受信でデータがバッファに格納されるのは、
・MASTER モードで、SLAVE レスポンス送信のケース（応答する SLAVE が存在した場合）
・SLAVE モードで、MASTER レスポンス送信を受信した場合
・SLAVE モードで、ヘッダに含まれる ID が自分自身の ID でない場合で、他の SLAVE がレスポンスを返した場合
です。

・MASTER モードで MASTER レスポンス送信の場合
・SLAVE モードで自分自身がレスポンスデータを送信した場合
は、データがバッファに保存される事はありません。

4.8. 関数

lin_init
lin0_init [RL78/F15](ch0)
概要:初期化関数

宣言:
int lin_init(unsigned char mode, unsigned char id)
int lin0_init(unsigned char mode, unsigned char id)

説明:
・LIN の初期化
を行います

引数:
mode: MASTER/SLAVE 区分
LIN_MASTER(0x00) MASTER として初期化
LIN_SLAVE(0x01) SLAVE として初期化
id: LIN-ID を指定(0x00~0x3F)

戻り値:
0: 正常終了
1: 引数エラー

lin_response_data_set
lin0_response_data_set [RL78/F15](ch0)
概要:データ設定関数

宣言:
int lin_response_data_set(unsigned char *data, unsigned char size)
int lin0_response_data_set(unsigned char *data, unsigned char size)

説明:
・SLAVE 設定時のレスポンスデータの設定
を行います

引数:
*data: レスポンスデータ
size: データバイト数を指定(1~8)

戻り値:

0: 正常終了

1: 引数エラー

※SLAVE デバイスは、MASTER からデータ送信要求（SLAVE デバイスの LIN-ID をヘッダ送信）があった場合に、本関数で設定したデータをレスポンスデータとして送信します

lin_master_header_send
lin0_master_header_send [RL78/F15](ch0)

概要:ヘッダ送信関数

宣言:

int lin_master_header_send(unsigned char id, unsigned char size)

int lin0_master_header_send(unsigned char id, unsigned char size)

説明:

・MASTER からヘッダ送信（レスポンスは他者が返すことを期待）

を行います

引数:

id: ヘッダに含める LIN-ID

size: レスポンスとして返ってくる（事を期待する）データバイト数を指定(1~8)

戻り値:

0: 正常終了

1: 引数エラー

※RX231, RA2L1 向けの関数では、size を引数として指定しますが未使用です（関数の互換性のため、持たせているものです）

lin_master_header_response_send
lin0_master_header_response_send [RL78/F15](ch0)

概要:ヘッダ＋レスポンス送信関数

宣言:

int lin_master_header_response_send(unsigned char id, unsigned char *data, unsigned char size)

int lin0_master_header_response_send(unsigned char id, unsigned char *data, unsigned char size)

説明:

・MASTER からヘッダ＋レスポンス送信

を行います

引数:

*data: レスポンス部分で送信するデータ

size: 送信するデータバイト数を指定(1~8)

lin_slave_header_receive
lin0_slave_header_receive [RL78/F15](ch0)

概要:受信待機関数

宣言:

int lin_slave_header_receive(void)

int lin0_slave_header_receive(void)

説明:

・ヘッダの受信待機（SLAVE 側で実行）

を行います

引数:

なし

戻り値:

0: 正常終了

lin_status_clear
lin0_status_clear [RL78/F15](ch0)

概要:スタータスクリア関数

宣言:

lin_status_clear(void)

lin0_status_clear(void)

説明:

・ステータスのクリア

を行います

引数:

なし

戻り値:

0: 正常終了

lin_read_data
lin0_read_data [RL78/F15](ch0)

概要:受信バッファ読み出し関数

宣言:

int lin_read_data(lin_message *msg)

int lin0_read_data(lin_message *msg)

説明:

・リングバッファに保存されているデータの読み出し

を行います

引数:

msg->id: 受信した ID(unsigned char)

msg->data: 受信したデータ(最大 unsigned char [8])

msg->sum: 受信したチェックサム(unsigned char)

msg->size: 受信したデータサイズ(unsigned char)(1~8)

戻り値:

0: 正常終了

1: データの読み出し前に上書きされたデータあり(読み出しは完了，残っているデータで一番古いものを返す)

2: 保存されているデータなし(全て読み出し済み)

lin_read_data_size
lin_read_data_size [RL78/F15](ch0)

概要:受信バッファ未読数返却関数

宣言:

int lin_read_data_size(void)

int lin0_read_data_size(void)

説明:

・リングバッファに保存されているデータの数の読み出し

を行います

引数:

なし

戻り値:

0: 未読み出しのデータなし

>0: 受信バッファに溜まっているメッセージ数

lin_read_buf_clear
lin0_read_buf_clear [RL78/F15](ch0)

概要:受信バッファクリア関数

宣言:

void lin_read_buf_clear(void)

void lin0_read_buf_clear(void)

説明:

・リングバッファに保存されているデータのクリア

を行います(保存されているデータを全て読み出し済みと設定します)

引数:

なし

戻り値:

なし

5. UART 通信

UART 通信は、コード生成の機能を使用しています。基本的な送受信の関数や割り込み処理は、コード生成に任せて、ユーザプログラムでバッファリングやデータ変換等を行っています。

通信速度は、115,200bps に設定、8 ビット、パリティなし、ストップビット 1 の設定です。（速度設定等は、コード生成機能の GUI で設定。）

デフォルトでは、HSB_CAN_MULTI_4 のボードが UART 通信担当となり、UART 通信で PC とやり取りを行います。

5.1. 使用 ch

マイコン種	UART 通信に使用しているモジュール	備考
RX231	SCI1	スマートコンフィグレータで出力したコードを使用
RL78/F15	UART0	コード生成機能で出力したコードを使用
RA2L1	SCI9	FSP で出力したコードを使用
RL78/G11 (HSB_LIN_COMM)	UART1	コード生成機能で出力したコードを使用 ※デバッグモード時のみ UART を使用

5.2. ソースコード構成

sci フォルダ内

ファイル名	説明
sci.c	SCI(UART)の通信処理
sci.h	↑ヘッダファイル
readme.txt	SCI の初期設定等の説明

sci.c, sci.h は、マイコン種に拠らず共通のコード（機種依存の部分は条件コンパイルで分ける）事としています。sci.h に CPU タイプ等の設定を行う必要があります。

5.3. sci.h の設定

```
/*-----------------------------------------------------------------------
マイコン種別（いずれか 1 つを選択）(*1')
-----------------------------------------------------------------------*/

#ifndef MCU_TYPE_DEFINED//他で定義済みの場合は、ここでの設定をスキップ

//下記 4 種類の内 1 つを有効化

#define MCU_TYPE_RX
//#define MCU_TYPE_RL78          CPU タイプの選択
//#define MCU_TYPE_RA
//#define MCU_TYPE_RH850

#endif

/*-----------------------------------------------------------------------
RL78 の場合  スマートコンフィグレータ，コード生成のどちらか （どちらかを選択）(*2)
-----------------------------------------------------------------------*/

#if defined(MCU_TYPE_RL78)
//#define RL78_SMART_CONFIGURATOR  //スマートコンフィグレータ（RL78/G23 以降）
#define RL78_CODE_GENERATION  //コード生成
#endif                                  RL78 で F15 や G11 はコード生成を使用して
                                        いるので、コード生成側を選ぶ

/*-----------------------------------------------------------------------
      RA の場合  SCI, SCI_B のどちらか （どちらかを選択）(*2')
-----------------------------------------------------------------------*/

#if defined(MCU_TYPE_RA)
#define RA_SCI  //通常の SCI          RA で、RA2L1 は RA_SCI 側を選ぶ
//#define RA_SCI_B  //SCI_B
#endif
```

ソースコード CD に含まれるプロジェクトでは設定済みなので、変更の必要はありませんが、上記で CPU タイプや、コード生成区分（RL78 の場合のみ）、SCI のモジュール区分（RA の場合のみ）を選択します。

```
#define SCI_SEND_BUF_SIZE  1024  //送信バッファ：1024 x 2 ページ  = 2048bytes を送信バッファとして使用
#define SCI_RECV_BUF_SIZE  512  //受信バッファ：512bytes
/*
※RL78 の G10 等 RAM の小さいマイコンでは、デフォルト(2kB)は、RAM 容量をオーバしていますので
SCI_SEND_BUF_SIZE  の値を調整してください
*/

#define SCI_USE_FLOAT  //浮動小数点の関数を使用する
/*
RL78 の G10 等 ROM の小さいマイコンでは必要に応じてコメントアウトしてください
（RL78 では、コメントアウトする事で 4kB 程度 ROM 消費量が減ります）
```

また、メモリ(RAM)の使用量を設定する箇所がありますので、メモリの空きがない場合は削る等の変更を行ってください。SCI_SEND_BUF_SIZE 1024, SCI_RECV_BUF_SIZE 512 の場合は、2.5kB の RAM を消費します。

・RL78/G11 の場合

```
#define SCI_SEND_BUF_SIZE  256  //送信バッファ:1024 x 2 ページ  = 2048bytes を送信バッファとして使用
#define SCI_RECV_BUF_SIZE  16  //受信バッファ:16bytes(キーボードからのコマンド入力を想定)
/*
※RL78 の G10 等 RAM の小さいマイコンでは、デフォルト(2kB)は、RAM 容量をオーバしていますので
SCI_SEND_BUF_SIZE  の値を調整してください
*/

//#define SCI_USE_FLOAT  //浮動小数点の関数を使用する
/*
RL78 の G10 等 ROM の小さいマイコンでは必要に応じてコメントアウトしてください
(RL78 では、コメントアウトする事で 4kB 程度 ROM 消費量が減ります)
```

RL78/G11 は RAM 容量が小さいので、バッファ容量を減らしています。また、浮動小数点関連の関数は未使用とすることにより、ROM 使用量を削減可能です。

※RL78/G11 は、通常動作では、SCI の機能を使用していません(デバッグモード時のみ、送信側の機能を使用しています)

5.4. コード生成ツールでの設定

RX231 では、「スマートコンフィグレータ」
RL78/F15 では「コード生成ツール」
RA2L1 では、「FSP」
のツールを使用して、UART の通信に関するプログラムコード(API 関数)を出力します。

ツールを使用したコード生成のフローに関しては、「取扱説明書　開発環境編」で説明します。

5.5. UART 初期設定

```
sci_start();
```

初期設定の関数は、sci_start()です。UART の送受信前に、本関数を実行してください。

5.6. UART 送信

・文字列の表示

```
sci_write_str("ABCDE¥n");
```

UART を使用して、文字送信を行う場合のコードです。HSB_CAN_MULTI_4 の J11 が UART の通信ポートですので、接続先の PC 上で端末を開いた場合、画面上に
ABCDE
と表示されます。(端末は、通信速度 115,200bps に設定してください)

・hex コードの表示

```
sci_write_uint16hex(0x123A);
```

123A

数値を、16 進数で表示します。

・数値の表示

```
sci_write_uint16(0xABCD);
sci_write_int16(0xABCD);
```

43981

-21525

数値を、符号なし 16bit，符号付き 16bit で表示します

・小数点を含む数値の表示

```
char buf[20];
float a = 1.23456;
float2str(a, 2, &buf[0]);
sci_write_str(buf);
```

1.23

小数点以下 2 位まで表示を行います。

5.7. UART 受信

```
unsigned short ret;
unsigned char c;

ret = sci_read_char(&c);
if (ret != SCI_RECEIVE_DATA_EMPTY)
{
   switch(c)
   {
      case '1':
```

sci_read_char()は、1 文字受信を行う関数です。CAN や LIN と同様に受信は割り込みルーチンで、受信バッファにデータをコピーする処理が実行されます。sci_read_char()は、受信バッファ（リングバッファ）のデータを読み出す関数となります。

5.8. 関数

sci_start

概要:SCI 初期化関数

宣言:

void sci_start(void)

説明:

SCI の動作開始を行います

引数:

なし

戻り値:

なし

sci_stop

概要:SCI 停止関数

宣言:

void sci_stop(void)

説明:

SCI の動作停止を行います

引数:

なし

戻り値:

なし

sci_write_char

概要:SCI 1 文字出力関数

宣言:

void sci_write_char(unsigned char c)

説明:

SCI に 1 文字出力行います

引数:

c: 表示する文字の文字コード(1 を表示する場合は、0x31 または'1')

戻り値:

なし

sci_write_str

概要:SCI 文字列出力関数

宣言:

void sci_write_str(const char *str)

説明:

SCI に文字列を出力します

引数:

*str: 文字列(¥0 終端)

戻り値:

なし

説明:

文字列に、"¥n"(0x0a)が含まれる場合は、"¥r¥n"(0x0d,0x0a)に変換されます

sci_write_uint8_hex
sci_write_uint16_hex
sci_write_uint32_hex

概要:SCI hex 出力関数

宣言:

void sci_write_uint8_hex(unsigned char c)
void sci_write_uint16_hex(unsigned short s)
void sci_write_uint32_hex(unsigned long l)

説明:

SCI に hex コードを出力します

引数:

c: hex コード(8bit)
s: hex コード(16bit)
l: hex コード(32bit)

戻り値:

なし

使用例:

```
sci_write_str("0x");
sci_write_uint8_hex( 0x5a );
```

端末に、"0x5a"が表示されます。

sci_write_uint8
sci_write_uint16
sci_write_uint32

概要:SCI 数値出力関数

宣言:

void sci_write_uint8(unsigned char c)
void sci_write_uint16(unsigned short s)
void sci_write_uint32(unsigned long l)

説明:

SCI に符号なしで数値を出力します

引数:

c: 表示させる数値(8bit)

s: 表示させる数値(16bit)

l: 表示させる数値(32bit)

戻り値:

なし

使用例:

unsigned short s = 0x8000;

sci_write_uint16(s);

端末に、"32768"が表示されます。

sci_write_int8

sci_write_int16

sci_write_int32

概要:SCI 数値出力関数

宣言:

void sci_write_int8(char c)

void sci_write_int16(short s)

void sci_write_int32(long l)

説明:

SCI に(負の数値の場合)符号付きで数値を出力します

引数:

c: 表示させる数値(8bit)

s: 表示させる数値(16bit)

l: 表示させる数値(32bit)

戻り値:

なし

使用例:

short s = 0x8000;

sci_write_int16(s);

端末に、"-32768"が表示されます

floart2str
float2str_eformat
double2str
double2str_eformat

概要:浮動小数点数ー文字列変換関数

宣言:

void float2str(float value, int num, char *str)

void float2str_eformat(float value, int num, char *str)

void double2str(double value, int num, char *str)

void double2str_eformat(double value, int num, char *str)

説明:

浮動小数点数を文字列に変換します

（_eformat は 1.23e-3 の様に e 形式に変換を行う関数です）

引数:

value: 表示させる数値(float/double 型)

num: 小数点以下の桁数

*str: 文字列変換後の先頭ポインタ

戻り値:

なし

使用例:

float f = 1.23456f;

char buf[20];

float2str(f, 2, buf);

sci_write_str(buf);

端末に、"1.23"が表示されます。

sci_write_flush

概要:SCI バッファ出力関数

宣言:

void sci_write_flush(void)

説明:

SCI の出力バッファに溜まっているデータを全て出力します

引数:

なし

戻り値:

なし

説明:

SCI の動作を停止させる場合等

sci_write_flush();

sci_stop();

本関数を実行後に停止させると、バッファに蓄えられているデータを全て出力した後に SCI の処理を停止する動作となります

sci_read_char

概要:SCI 1 文字読み出し関数

宣言:

unsigned short sci_read_char(unsigned char *c);

説明:

SCI から 1 文字読み出しを行います

引数:

c: 読み出し結果(ポインタ)

戻り値:

0: 受信データがあり、*c に読み出した文字(ポインタ)を格納

0xffff(SCI_RECEIVE_DATA_EMPTY): バッファに溜まっている受信データがない

sci_read_str

概要:SCI 文字列読み出し関数

宣言:

unsigned short sci_read_str(char *str, unsigned short size);

説明:

SCI から文字列を読み出します

引数:

str: 読み出し結果(ポインタ)

size: 読み出すバイト数

戻り値:

0: 受信データが size バイト以上あり、*str に読み出した文字の先頭ポインタを格納

0xffff(SCI_RECEIVE_DATA_EMPTY): バッファに size バイトの受信データが溜まっていない

sci_read_data_size

概要:受信バッファ未読数返却関数

宣言:

unsigned short sci_read_data_size(void)

説明:

・リングバッファに保存されているデータの数の読み出し
を行います

引数:

なし

戻り値:

0: 未読み出しのデータなし

>0: 受信バッファに溜まっているメッセージ数

sci_read_buf_clear

概要:受信バッファクリア関数

宣言:

void sci_read_buf_clear(void)

説明:

・リングバッファに保存されているデータのクリア
を行います(保存されているデータを全て読み出し済みと設定します)

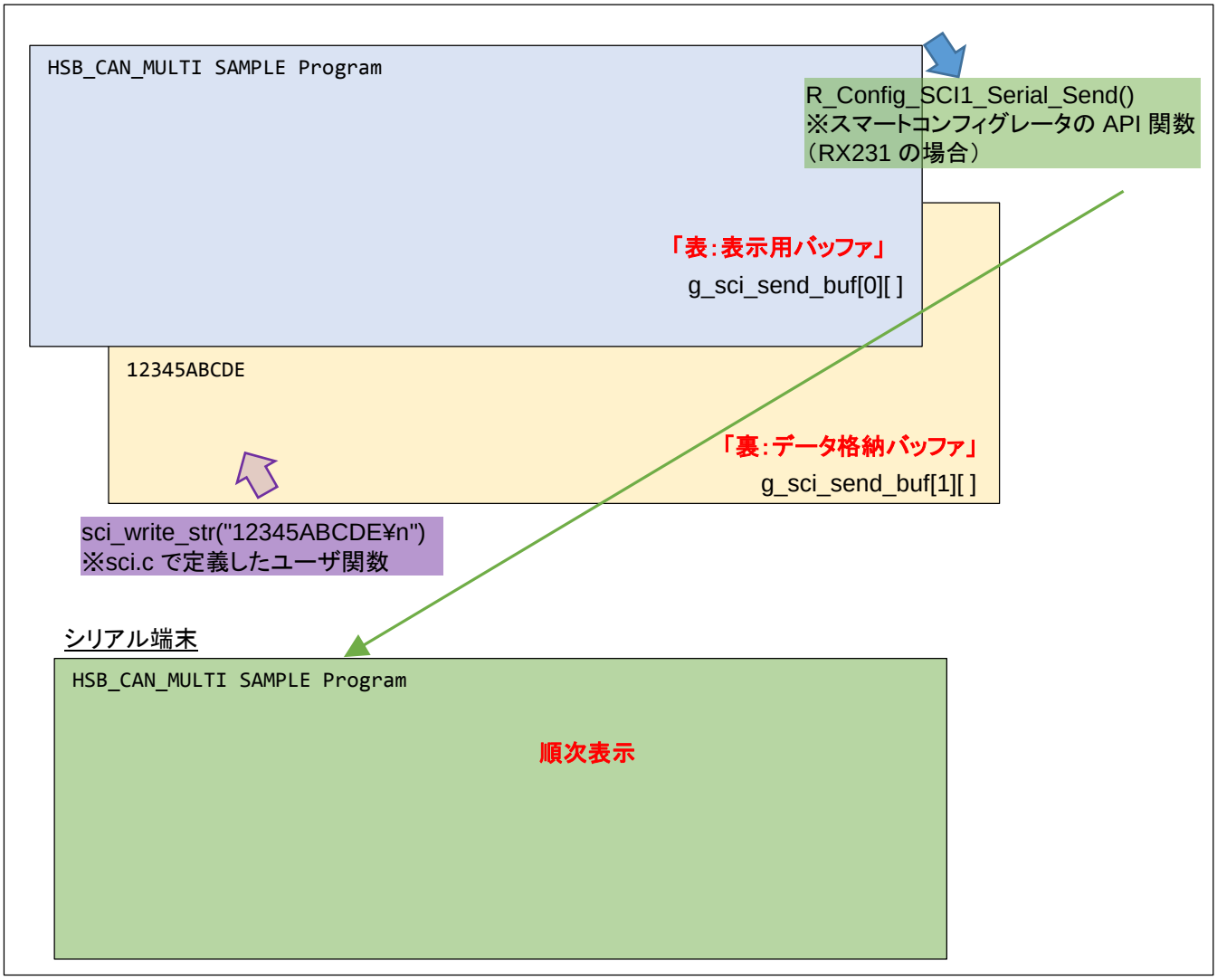
引数:

なし

戻り値:

なし

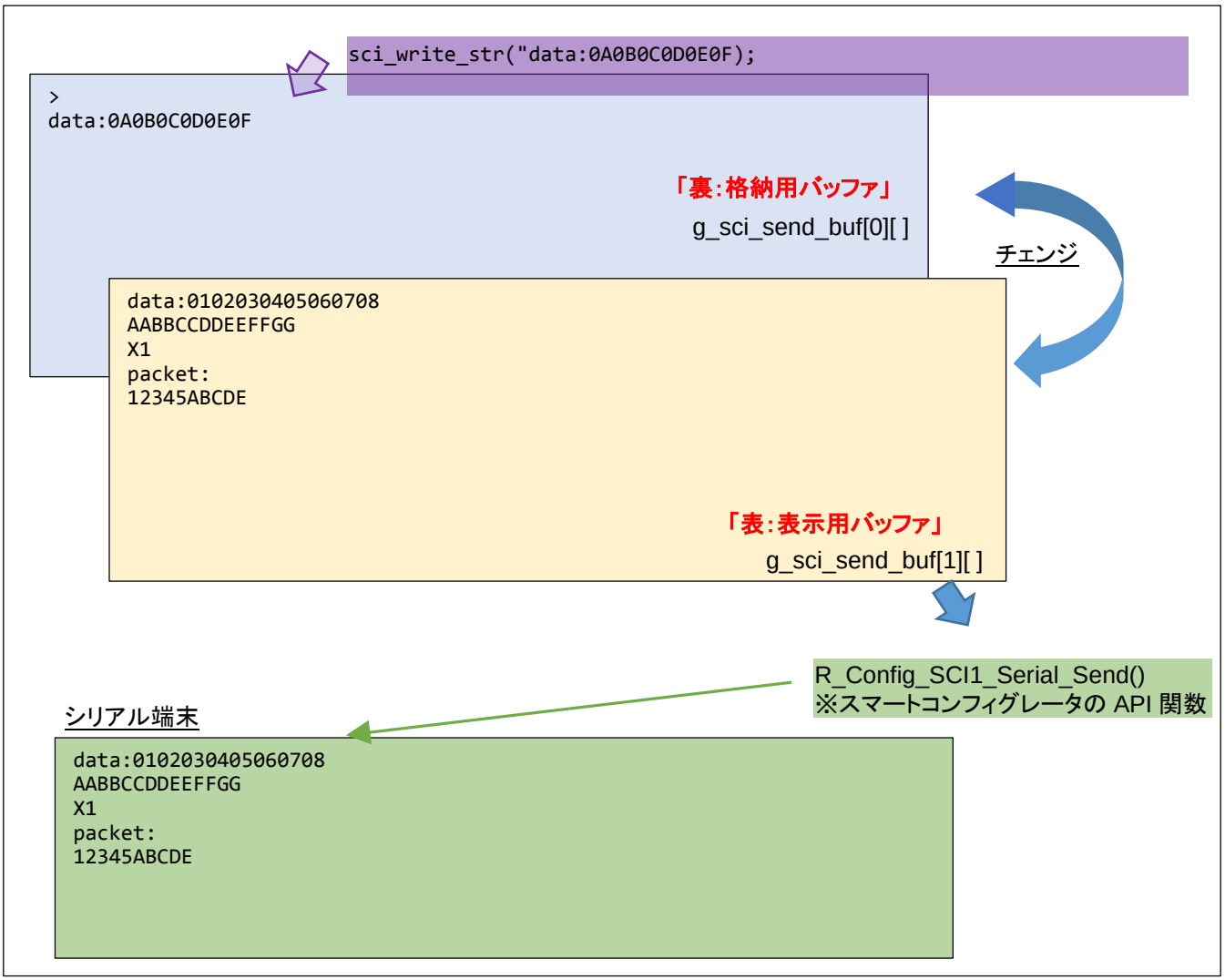
5.9. 出力バッファに関して



文字出力用のバッファは 2 面（2 つ）あり、表示用のバッファは API 関数に渡され、シリアル端末に順次データが送られます。シリアルは、115,200bps 設定なので、1 文字を送信するのに、87us（1/115,200×10）掛かります。これは、マイコンの処理に比べて非常に遅いので、文字出力が終わるまで待っていると効率が悪いためバックグラウンドで送信処理を行わせています。

※本プログラムの設定では、スタートビット=1，データ（1 文字）=8bit，ストップビット=1 で合計 10bit で 1 文字送信します

表示用バッファのデータを全て出力した時点で、今度はデータ格納バッファのデータを API 関数に渡して、バッファの表（表示用）と裏（データ格納用）を入れ替えます。



出力バッファのサイズは、sch.h 内で

```
#define SCI_SEND_BUF_SIZE  1024  //送信バッファ：1024 x 2 ページ  = 2048bytes を送信バッファとして使用
```

定義されており、デフォルトでは、1024 バイト×2 面です。

このバッファが溢れた場合、sci_write_xxx()関数で出力したデータは捨てられるという動作となります。

```
g_sci_send_nowait_flag = FLAG_SET;
```

データを捨てるか、データ出力を待つ（バッファが格納可能になるのを待つ）かは選択可能で、上記変数を FLAG_SET に変更すれば、データ出力を待つ設定となります。

この場合、UART から出力されるデータに抜けはなくなりますが、sci_write_xxx()関数の実行が終わるまで「待つ」という動作になりますので、このフラグ変数の設定は注意が必要です。

6. ボード搭載機能

6.1. HSB_CAN_MULTI_1

HSB_CAN_MULTI_1 には、ボード固有の機能として、モータ制御機能と、モータの回転数を取得するエンコーダセンサあります。

6.1.1. モータ制御

モータ制御の回路に関しては、「取扱説明書」内に記載があります。プログラムでモータを回転させる際、モータドライブ回路に接続されている、2 端子(IN1, IN2)を PWM 制御する必要があります。

6.1.1.1. ソースコード構成

motor_drive フォルダ内

ファイル名	説明
motor_drive.c	モータ駆動
motor_drive.h	↑ヘッダファイル

6.1.1.2. 使用タイマ

モータ駆動では、PWM 波形を生成するのにタイマ機能を使用しています。

マイコン種	モータ駆動に 使用している モジュール	備考
RX231	MTU2 MTU0	
RL78/F15	TAU0/ch0 TAU1/ch0	
RA2L1	GPT3	GTIOC3A/GTIOC3B の 2 端子を使用，タイマは 1 種

IN1	IN2	OUT1	OUT2	動作
0	0	Hi-Z	Hi-Z	停止
PWM	**1**	**PWM**	**H**	**正転**
1	**PWM**	**H**	**PWM**	**逆転**
1	1	L	L	ブレーキ

モータを回転させる場合は、IN1, IN2 の片方の端子を H 固定とし、もう一方の端子を PWM 駆動とします。PWM の L 期間の割合が多い程、モータは高速に回転します。（L 固定とした場合、duty=100%です）

例えば、RX231 では、正転とする場合、MTU0 を H 出力として、MTU2 を PWM で駆動する事でモータを回転させる事ができます。逆転の場合は、MTU2 を H 出力として、MTU0 を PWM 駆動とします。

6.1.1.3. 関数

motor_start

概要:モータ初期化関数

宣言:

void motor_start(void)

説明:

モータの動作開始を行います

引数:

なし

戻り値:

なし

motor_duty

概要:モータ duty 変更関数

宣言:

void motor_duty(short duty)

説明:

モータの duty 変更を行います

引数:

duty: duty 比を設定（-100~100 の範囲が有効）

戻り値:

なし

```
motor_start();

motor_duty(-50); //逆回転の方向に 50%の duty で駆動する
```

最初に、motor_start()を実行してください。その後、motor_duty()でモータの回転数を変えられます。
duty は-100~100[%]が有効で、0 でモータ停止となります。

6.1.2. モータエンコーダ

モータエンコーダは、光がスリットの空いている円盤で一定時間(10ms)で何回遮られたかをカウントして、回転数、回転方向を取得します。

6.1.2.1. ソースコード構成

motor_encoder フォルダ内

ファイル名	説明
motor_encoder.c	モータエンコーダ
motor_encoder.h	↑ヘッダファイル

6.1.2.2. 使用機能

モータエンコーダでは、タイマ機能（もしくは割り込み機能）を使用しています。

マイコン種	モータエンコーダに使用しているモジュール	備考
RX231	MTU1 CMT2	エンコーダセンサのカウントに使用 一定時間(10ms)のカウントに使用
RL78/F15	INTC3 INTC11 TAU1/ch2	エンコーダセンサのカウントに使用 エンコーダセンサのカウントに使用 一定時間(10ms)のカウントに使用
RA2L1	GPT2 GPT4	エンコーダセンサのカウントに使用 一定時間(10ms)のカウントに使用

RX231, RA2L1 では、エンコーダセンサのカウントはタイマのハードウェアで行う事ができます。それに対し、RL78/F15 はハードウェアエンコーダ機能が無いため、端子の割り込みでエンコーダセンサのカウントを数えています。（多数回の割り込み処理が実行されるため、CPU リソース的にはスマートやり方とは言えません）

6.1.2.3. 関数

motor_enoder_start

概要:モータエンコーダ開始関数

宣言:

void motor_encoder_start(void)

説明:

モータエンコーダの動作開始を行います

引数:

なし

戻り値:

なし

motor_enoder_get

概要:モータエンコーダカウント指示関数

宣言:

void motor_encoder_get(void)

説明:

モータエンコーダのカウントを実行します、本関数呼び出し後 10ms でカウントは停止します

引数:

なし

戻り値:

なし

motor_enoder_rpm

概要:モータエンコーダ回転数取得関数

宣言:

int motor_encoder_rpm(short *rpm)

説明:

モータエンコーダのカウント数を回転数[rpm]に変換して返します

引数:

*rpm: 回転数

戻り値:

0: 回転数の取得 OK

1: エンコーダ値の取得未（motor_encoder_get 実行後 10ms 経過していない）

```
short rpm;
motor_encoder_start(); //最初に 1 回実行する
motor_encoder_get();
//…10ms 以上期間を開ける
motor_encoder_rpm(&rpm);
```

motor_encoder_get()実行後、10ms 以上経過した後で、motor_encoder_rpm()を実行してください。

6.1.3. スイッチの読み取り

本ボードには、プッシュスイッチ(SW2, SW3)があります。

任意の用途で使用可能なスイッチですが、デモプログラムではモータの回転数を増減させるスイッチとして働きます。

6.1.3.1. ソースコード構成

timer フォルダ内

ファイル名	説明
timer.c	定期処理

スイッチの読み取りは、timer.c 内で行っています。

6.1.3.2. 使用機能

スイッチはでは、タイマ機能で一定間隔（1ms 毎）に読み取りを行っています。

マイコン種	スイッチ読み取りに使用しているモジュール	備考
RX231	CMT1	
RL78/F15	TAU0/ch2	
RA2L1	GPT5	

SW2 は、duty を増やす方向、SW3 は duty をマイナス方向に増やす様に作用します。現在の duty が-50%（逆回転で 50%）のとき、SW2 を 2 回押すと duty は 0（回転は停止）となります。

6.2. HSB_CAN_MULTI_2

HSB_CAN_MULTI_2 には、ボード固有の機能として、LCD 制御と、マトリックスキーがあります。

6.2.1. キャラクタ LCD

LCD との接続に関しては、「取扱説明書」内に記載があります。LCD とマイコンは、6 端子(E, RS, D7-D4)で接続されており、6 端子を汎用 I/O ポートで制御しています。

6.2.1.1. ソースコード構成

lcd_1602 フォルダ内

ファイル名	説明
lcd_1602.c	LCD 駆動
lcd_1602.h	↑ヘッダファイル
lcd_1602_board_def_CPU_CARD_*XXXX*.h	XXXX には CPU 名が入ります

lcd_1602.c 内には CPU タイプに依存しないコード。CPU 毎に異なる部分は、lcd_1602_board_def_CPU_CARD_XXXX.h 側に記載してあります。

6.2.1.2. 関数

lcd_init

概要:LCD 初期化関数

宣言:

void lcd_init(void)

説明:

・LCD の初期化を行います

引数:

なし

戻り値:

なし

lcd_cmd

概要:LCD コマンド送信関数

宣言:

void lcd_cmd(unsigned char c)

説明:

・LCD にコマンドを送信します

引数:

c: 送信コード(8bit)

戻り値:

なし

※コマンドに関しては、「取扱説明書」の 6 章に LCD の資料が掲載されていますので参照してください
0x01:表示クリア
0x02:カーソルをホーム位置に移動
等のコマンドがあります

lcd_hs1
lcd_hs2

概要:1 行目にカーソルを移動させる関数(lcd_hs1), 2 行目にカーソルを移動させる関数(lcd_hs2)

宣言:

void lcd_hs1(void)

void lcd_hs2(void)

説明:

・LCD にカーソル移動のコマンドを送信します

引数:

なし

戻り値:

なし

lcd_pos

概要:指定した位置にカーソルを移動させる関数

宣言:

lcd_pos (unsigned char row, unsigned char column)

説明:

・LCD にカーソル移動のコマンドを送信します

引数:

row: 行の指定(1~2)

column: 桁の指定(0~63)

戻り値:

なし

	LCD 表示エリア									表示エリア外 （スクロール機能で表示可能）				
1 行目	0x0	0x1	0x2	0x3	0x4	0x5	…	0xE	0xF	0x10	0x11	0x12	…	0x3F
2 行目	0x40	0x41	0x42	0x43	0x44	0x45	…	0x4E	0x4F	0x50	0x51	0x52	…	0x7F

LCD のアドレスは、0x0~0x7F となり 16 桁の範囲が表示可能です。

lcd_clear

概要:LCD 画面クリア関数

宣言:

void lcd_clear(void)

説明:

・LCD の画面をクリアします

引数:

なし

戻り値:

なし

lcd_write_char

概要:LCD キャラクタ送信関数

宣言:

void lcd_write_char(unsigned char c)

説明:

・LCD にキャラクタ(1 文字)を送信します

引数:

c: キャラクタコード

戻り値:

なし

使用例:

lcd_write_char('A');

現在のカーソル位置に、A を表示させる。

lcd_write_uint8_hex
lcd_write_uint16_hex
lcd_write_uint32_hex

概要:LCD 数値表示(16 進)関数
宣言:
　　void lcd_write_uint8_hex(unsigned char c)
　　void lcd_write_uint16_hex(unsigned short s)
　　void lcd_write_uint32_hex(unsigned long l)

説明:
　・LCD に 16 進数で数値を表示します

引数:
　　c, s, l: 表示する数値

戻り値:
　　なし

使用例:
lcd_write_uint16_hex(0x1234);
　現在のカーソル位置に、1234 を表示させる。

lcd_write_uint8
lcd_write_uint16
lcd_write_uint32

概要:LCD 数値表示関数
宣言:
　　void lcd_write_uint8(unsigned char c)
　　void lcd_write_uint16(unsigned short s)
　　void lcd_write_uint32(unsigned long l)

説明:
　・LCD に数値を表示します(10 進数、符号なし)

引数:
　　c, s, l: 表示する数値

戻り値:

なし

使用例:

lcd_write_uint16(0x8000);

現在のカーソル位置に、32768 を表示させる。

lcd_write_int8
lcd_write_int16
lcd_write_int32

概要:LCD 数値表示関数

宣言:

void lcd_write_int8(char c)

void lcd_write_int16(short s)

void lcd_write_int32(long l)

説明:

・LCD に数値を表示します（10 進数、符号あり）

引数:

c, s, l: 表示する数値

戻り値:

なし

使用例:

lcd_write_int16(0x8000);

現在のカーソル位置に、-32768 を表示させる。

lcd_write_str

概要:LCD 文字列表示関数

宣言:

lcd_write_str(const char *str)

説明:

・LCD に文字列を送信します

引数:

*str: 表示する文字列

戻り値:

なし

使用例:

lcd_write_str("LCD SAMPLE");

現在のカーソル位置に、LCD SAMPLE を表示させる。

```
lcd_init();

lcd_hs1();
lcd_write_str("HSB_CAN_MULTI_2)";
```

lcd_init()は最初に 1 回実行。(LCD との接続が外れて挿しなおした場合等は再度実行してください。)

LCD の 1 行目に"HSB_CAN_MULTI_2"を表示する例です。

6.2.1. マトリックススイッチ

マトリックススイッチは、3 行、4 列の 12 個のスイッチを 7 本の信号線(4 本が出力, 3 本が入力)で読み取りを行っています。

6.2.1.1. ソースコード構成

key_matrix フォルダ内

ファイル名	説明
key_matrix.c	マトリックススイッチ読み取り
key_matrix.h	↑ヘッダファイル
key_matrix_board_def_CPU_CARD_XXXX.h	XXXX には CPU 名が入ります

6.2.1.2. 使用タイマ

マトリックススイッチの読み取りにタイマ機能(1ms 毎)を使用しています。

マイコン種	使用している モジュール	備考
RX231	CMT1	
RL78/F15	TAU0/ch2	
RA2L1	GPT5	

6.2.1.3. 関数

key_matrix_init

概要：マトリックススイッチ初期化関数

宣言：

void key_matrix_init(void)

説明：

・マトリックススイッチの初期化を行います

引数：

なし

戻り値：

なし

key_matrix_scan

概要：マトリックススイッチスキャン関数

宣言：

void key_matrix_scan(void)

説明：

・マトリックススイッチの読み取りを行います

引数：

なし

戻り値：

なし

説明：

1 回の実行で、1 列（初回、キー7,4,1）の読み出しを行います。4 回実行する事で、全キーの読み出しが行えます。タイマ等で定期的に、本関数を実行してください。

```
key_matrix_init();  //最初に 1 回実行

key_matrix_scan(); //1 列目（キー7,4,1 の読み取り）
key_matrix_scan(); //2 列目（キー8,5,2 の読み取り）
key_matrix_scan(); //3 列目（キー9,6,3 の読み取り）
key_matrix_scan(); //4 列目（キーCL,EN,0 の読み取り）

key_matrix_scan(); //1 列目（キー7,4,1 の読み取り）
…
```

key_matrix_scan()関数は、1 回の実行で 1 列（3 つのキー）の読み出しを行います。4 回の実行で全キーの読み出しとなります。

本プログラムでは、1ms 毎に key_matrix_scan()を実行しています。

※読み取り周期を非常に長く取る場合は、連続して 4 回実行する様にしてください

6.2.1.4. 使用変数

```
typedef struct{
   unsigned char key[10];
   unsigned char en;
   unsigned char cl;
} key_matrix;
```

キーの読み出し結果は、key_matrix 構造体を使用して保持されます。

本プログラムでは、2 つのグローバル変数が用意してあり

```
key_matrix g_key_matrix;
unsigned short g_key_matrix_pressed;
```

g_key_matrix は読み出し結果を保持する構造体変数です。

0 のキーが押されている場合は、
g_key_matrix.key[0] = 0
となります。

```
#define KEY_MATRIX_KEY_PUSH  0
#define KEY_MATRIX_KEY_RELEASE  1
```

押されている場合は 0，押されていない場合は 1 です。各キーは 4ms 毎に読み取られて、g_key_matrix 変数に反映されます。（1ms 毎に 3 つのキーのデータが更新、各キーのスキャン頻度は 4ms 毎）

g_key_matrix_pressed は、履歴が残る変数です。

bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
KEY					CL	EN	9	8	7	6	5	4	3	2	1	0
data(0b)	0	0	0	0	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1

キーが押された場合、当該ビットが立つ(1)となります。(※g_key_matrix.key の 0/1 と逆です)

また、一度立ったビットはユーザプログラムでクリアしない限り、1 のままです。

```
g_key_matrix_pressed = 0;

…

if ((g_key_matrix_pressed & 0x0001) != 0)
{
   //g_key_matrix_pressed をクリアしてから、0 のキーが押されている
```

6.3. HSB_CAN_MULTI_3

HSB_CAN_MULTI_3 には、ボード固有の機能として、フォトダイオード(明るさセンサ)、SPI フラッシュメモリ、I2C 温度センサがあります。

6.3.1. フォトダイオード

フォトダイオードは明るさセンサとして使用しています。センサ部の明るさにより、出力値が異なる動作となります。センサの出力はアナログ電位となるので、マイコンの A/D 変換機能を使用して、明るさの数値を得る形です。

6.3.1.1. ソースコード構成

adc フォルダ内

ファイル名	説明
adc.c	フォトダイオードの A/D 値取得
adc.h	↑ヘッダファイル

6.3.1.2. 使用機能

A/D 変換機能を使用しています。

マイコン種	使用している モジュール	備考
RX231	S12ADE	12bit 分解能
RL78/F15	A/D コンバータ	10bit 分解能
RA2L1	ADC12	12bit 分解能

6.3.1.3. 関数

adc_val

概要:A/D 変換値取得関数

宣言:

int adc_val(unsigned short *val)

説明:

・A/D 変換結果の取得

引数:

*val: 変換結果

戻り値:

0: 正常終了

1: エラー（A/D 変換未完了　または　未実行）

adc_kick

概要:A/D 変換実行関数

宣言:

int adc_kick(void)

説明:

・A/D 変換の実行

引数:

なし

戻り値:

0: 正常終了

1: エラー（A/D 変換実行中）

```
unsigned short val;

adc_kick();

//一定時間経過後
adc_val(&val);
```

adc_kick()で A/D 変換が実行されます。A/D 変換が終了すると、割り込みにより変換結果を保存しておく動作となります。adc_val()で、保存されている値を取得できます。

2 つの関数は、タイマ等で定期的に

```
adc_val(&val); //前回の実行結果を取得
adc_kick();      //次の A/D 変換を実行
```

ペアで使用する事を想定した作りになっています。

フォトダイオードは、センサーに光が当たっている（明るい）場合、A/D 変換結果は大きな数値（RX231, RA2L1 では、0xFFF に近い数値、RL78/F15 では 0x3FF に近い数値）となり、暗い場合小さな数値となります。

6.3.2. SPI フラッシュメモリ

ボード上に搭載されているフラッシュメモリは、SPI インタフェースでアクセスするタイプで、2Mbit(256kB)の容量を持っています。

6.3.2.1. ソースコード構成

spi フォルダ内

ファイル名	説明
spi.c	SPI フラッシュメモリアクセス
spi.h	↑ヘッダファイル

6.3.2.2. 使用機能

フラッシュメモリのアクセスには、SPI 通信機能を使用しています。

マイコン種	使用している モジュール	備考
RX231	RSPI0	
RL78/F15	CSI11	
RA2L1	SPI0	

6.3.2.3. 関数

spi_flash_read_status_register

概要:ステータスレジスタリード関数

宣言:

int spi_flash_read_status_register(unsigned char *reg)

説明:

・ステータスレジスタの読み出し

引数:

*reg: ステータスレジスタ値

戻り値:

0: 正常終了

2: SPI 通信エラー

spi_flash_write_status_register

概要:ステータスレジスタライト関数

宣言:

int spi_flash_write_status_register(unsigned char reg)

説明:

・ステータスレジスタの書き込み

引数:

reg: ステータスレジスタ値

戻り値:

0: 正常終了

2: SPI 通信エラー

3: 書き込みイネーブルコマンドエラー

spi_flash_busy_wait

概要:BUSY 完了待ち関数

宣言:

int spi_flash_busy_wait(void)

説明:

・フラッシュメモリが BUSY の間待つ

引数:

なし

戻り値:

0: 正常終了

3: ステータスコマンドエラー

4 タイムアウト

spi_flash_write_enable

概要:書き込みイネーブル関数

宣言:

int spi_flash_write_enable(void)

説明:

・書き込みプロテクトの解除

引数:

なし

戻り値:

0: 正常終了

2: SPI 通信エラー

3: ステータスコマンドエラー

5 書き込みフラグが書き込み可に変化しないエラー

spi_flash_write_enable_status_register

概要:ステータスレジスタ書き込みイネーブル関数

宣言:

int spi_flash_write_enable_status_register(void)

説明:

・ステータスレジスタ書き込みプロテクトの解除

引数:

なし

戻り値:

0: 正常終了

2: SPI 通信エラー

spi_flash_write_disable

概要:書き込みディスエーブル関数

宣言:

int spi_flash_write_disable(void)

説明:

・書き込みプロテクト有効化

引数:

なし

戻り値:

0: 正常終了

2: SPI 通信エラー

3: ステータスコマンドエラー

5 書き込みフラグが書き込み不可に変化しないエラー

spi_flash_read_data

概要:データ読み出し関数

宣言:

int spi_flash_read_data(unsigned long addr, unsigned short size, unsigned short *data)

説明:

・データの読み出し

引数:

addr: フラッシュメモリアドレス（24bit アドレス）

size: 読み出しサイズ（最大 256）

*data: 読み出しデータ

戻り値:

0: 正常終了

1: 引数エラー

2: SPI 通信エラー

spi_flash_write_data

概要:データ書き込み関数

宣言:

int spi_flash_write_data(unsigned long addr, unsigned short size, unsigned char *data)

説明:

・データの書き込み

引数:

addr: フラッシュメモリアドレス（24bit アドレス）

size: 書き込みサイズ（最大 256）

*data: 書き込みしデータ

戻り値:

0: 正常終了

1: 引数エラー

2: SPI 通信エラー

3: 書き込みイネーブルコマンドエラー

補足:

256 バイト(ページ)をまたぐ addr, size の組み合わせ指定不可(次ページの書き込みは実行されない)

addr=0xXXXX00 size=256 の組み合わせの場合、0xXXXX00~0xXXXXFF の 256 バイトの書き込みを行う

addr=0xXXXX80 size=256 の組み合わせの場合、0xXXXX00~0xXXXXFF の 128 バイトの書き込みを行う

注意:

書き込みが完了する前に関数からリターンするので、連続して書き込みを行う場合、

spi_flash_busy_wait()　　//前の処理が終わるまで待つ

spi_flash_write_data(....)

とする必要があります

書き込みは、ブランク領域のみ行う事が可能です(データの上書き不可)

spi_flash_sector_erase

概要:セクタ消去関数

宣言:

int spi_flash_sector_erase(unsigned long addr)

説明:

・セクタ(4kB 単位)の消去

引数:

addr: フラッシュメモリアドレス(24bit アドレス)

戻り値:

0: 正常終了

2: SPI 通信エラー

3: 書き込みイネーブルコマンドエラー

補足:

アドレスは、0xXXX000(下 3 桁ゼロ)のアドレスを指定してください。(それ以外のアドレスを指定した場合は、下 3 桁を 000 に切り捨てます。

書き込みプロテクトが掛っている場合は、本関数の実行により解除されます。

注意:

消去が完了する前に関数からリターンします。連続して別な処理を行う際は、spi_flash_busy_wait()で処理の完了を確認後に、次の処理を実行してください。

spi_flash_block_erase

概要:ブロック消去関数

宣言:

int spi_flash_block_erase(unsigned long addr)

説明:

・ブロック(64kB 単位)の消去

引数:

addr: フラッシュメモリアドレス(24bit アドレス)

戻り値:

0: 正常終了

2: SPI 通信エラー

3: 書き込みイネーブルコマンドエラー

補足:

アドレスは、0xXX0000(下 4 桁ゼロ)のアドレスを指定してください。(それ以外のアドレスを指定した場合は、下 4 桁を 0000 に切り捨てます。

書き込みプロテクトが掛っている場合は、本関数の実行により解除されます。

注意:

消去が完了する前に関数からリターンします。連続して別な処理を行う際は、spi_flash_busy_wait()で処理の完了を確認後に、次の処理を実行してください。

spi_flash_chip_erase

概要:チップ消去関数

宣言:

int spi_flash_chip_erase(void)

説明:

・チップ(全体)の消去

引数:

なし

戻り値:

0: 正常終了

2: SPI 通信エラー

3: ステータスレジスタ書き込みコマンド，書き込みイネーブルコマンドエラー

補足:

ステータスレジスタの初期化も実行されます。

書き込みプロテクトが掛っている場合は、本関数の実行により解除されます。

注意:

消去が完了する前に関数からリターンします。連続して別な処理を行う際は、spi_flash_busy_wait()で処理の完了を確認後に、次の処理を実行してください。

スペック上の最大実行時間は、約 2 秒です（実測では、325ms@3.3V）。

6.3.3. I2C 温度センサ

ボード上搭載されている、温度センサが搭載されており、I2C インタフェースでアクセスが可能です。

6.3.3.1. ソースコード構成

i2c フォルダ内

ファイル名	説明
i2c.c	I2C 温度センサアクセス
i2c.h	↑ヘッダファイル

6.3.3.2. 使用機能

温度センサのアクセスには、I2C 通信機能を使用しています。

マイコン種	使用している モジュール	備考
RX231	RIIC0	
RL78/F15	IICA0	
RA2L1	IIC1	

HSB_CAN_MULTI_3（マイコン側）が MASTER, I2C 温度センサが SLAVE となります。（SLAVE アドレスは 0x4A）

6.3.3.3. 関数

i2c_write

概要:I2C データ送信関数

宣言:

int i2c_write(unsigned char *data, unsigned short size)

説明:

・データの送信

引数:

*data: 送信データ

size: 送信サイズ

戻り値:

0: 正常終了

1: API 関数戻り値エラー

2: タイムアウト

i2c_read

概要:I2C データ受信関数

宣言:

int i2c_read(unsigned char *data, unsigned short size)

説明:

・データの受信

引数:

*data: 受信データ

size: 受信サイズ

戻り値:

0: 正常終了

1: API 関数戻り値エラー

2: タイムアウト

tmp101_pointer_reg_write

概要:温度センサレジスタライト関数

宣言:

int tmp101_pointer_reg_write(unsigned char bit2)

説明:

・温度センサ(TMP101)のレジスタ設定

引数:

bit2: レジスタ設定値

戻り値:

0: 正常終了

1: API 関数戻り値エラー

2: タイムアウト

補足:

TMP101 のレジスタの b1~b0(2bit)に、引数で指定した値を設定します。

tmp101_temp_reg_read

概要:温度センサレジスタリード関数

宣言:

int tmp101_temp_reg_read(float *temp)

説明:

・温度センサ(TMP101)の温度レジスタの読み出し

・摂氏温度への変換

引数:

*temp: 温度値[℃]

戻り値:

0: 正常終了

1: API 関数戻り値エラー

2: タイムアウト

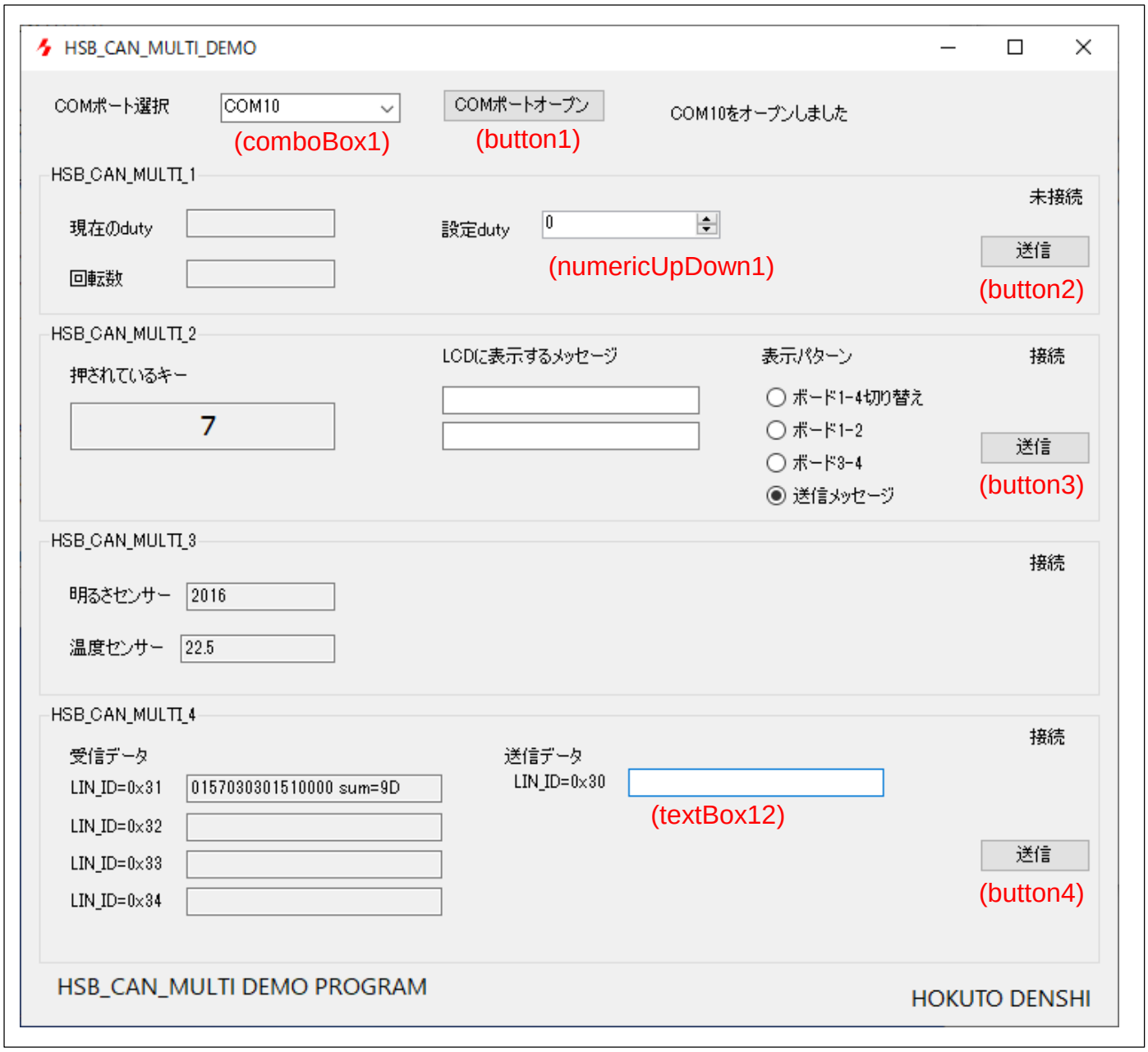
6.4. HSB_CAN_MULTI_4

HSB_CAN_MULTI_3 には、ボード固有の機能として、USB-Serial 変換(UART)、LIN 通信があります。

UART は、5 章。LIN 通信は 4 章を参照してください。

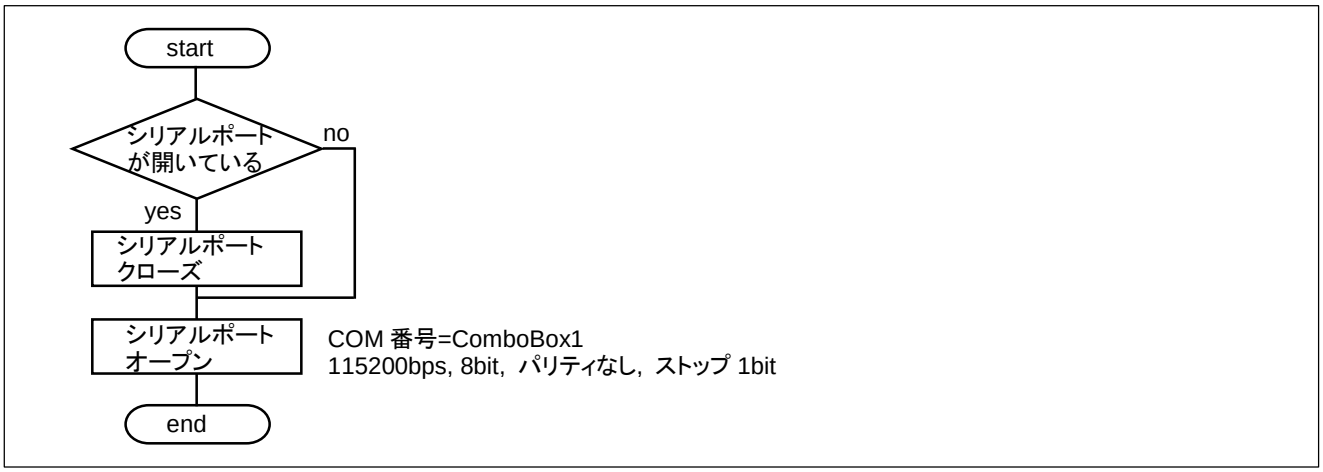
7. PC 向けモニタプログラム

7.1. HSB_CAN_MULTI_DEMO



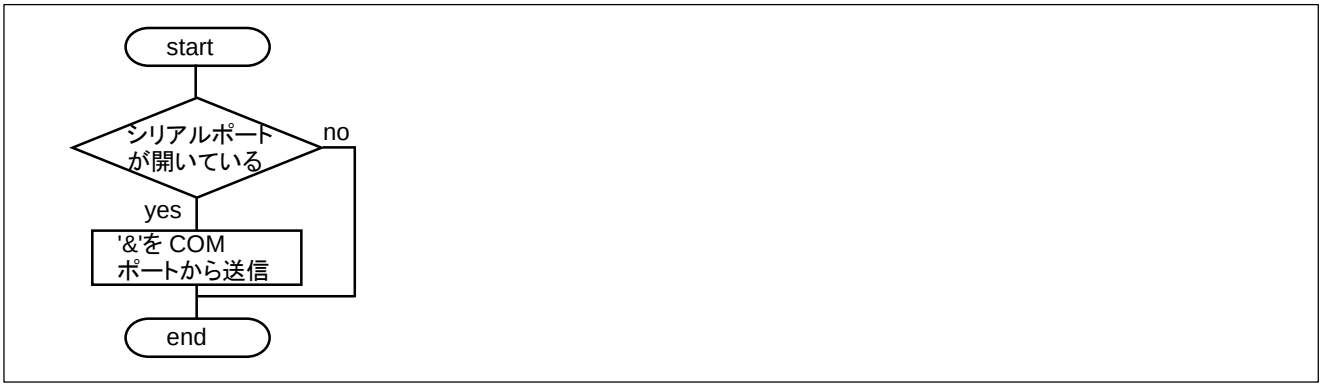
PC 向けのプログラムは、VisualC#向けのプロジェクトとなっています。

7.1.1. COM ポートオープン(button1_Click())



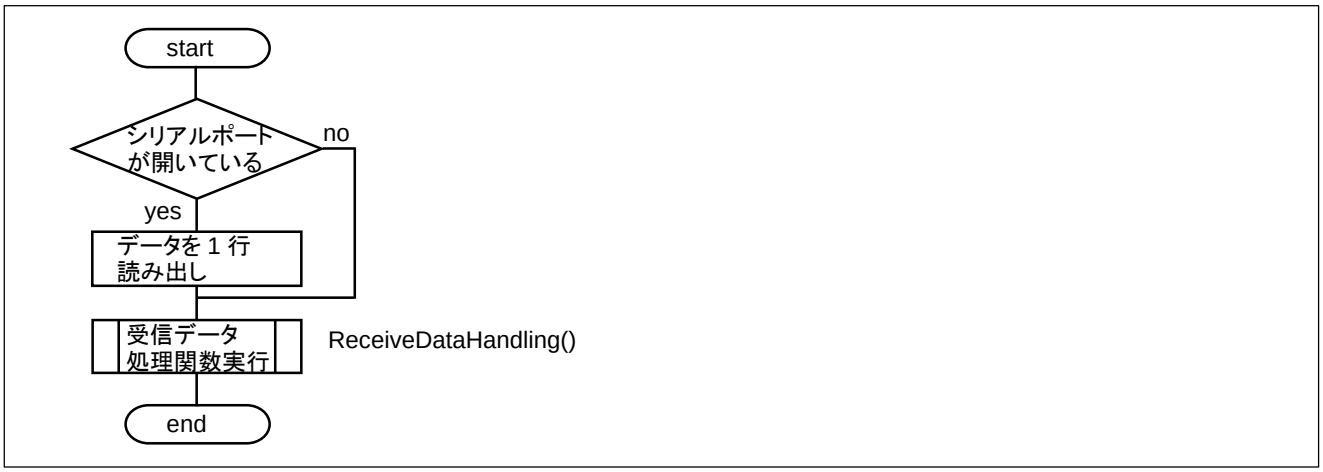
プログラムでは最初に、ComboBox1 で COM ポート番号を選択し、「COM ポートオープン」(button1)を押して、COM ポートを開く処理を行います。HSB_CAN_MULTI_4 側では、115,200bps, 8bit, パリティなしでデータの送受信を行う様になっていますので、PC 側のアプリケーションでも通信条件を同じとしています。

7.1.2. タイマ処理(timer1_Tick())



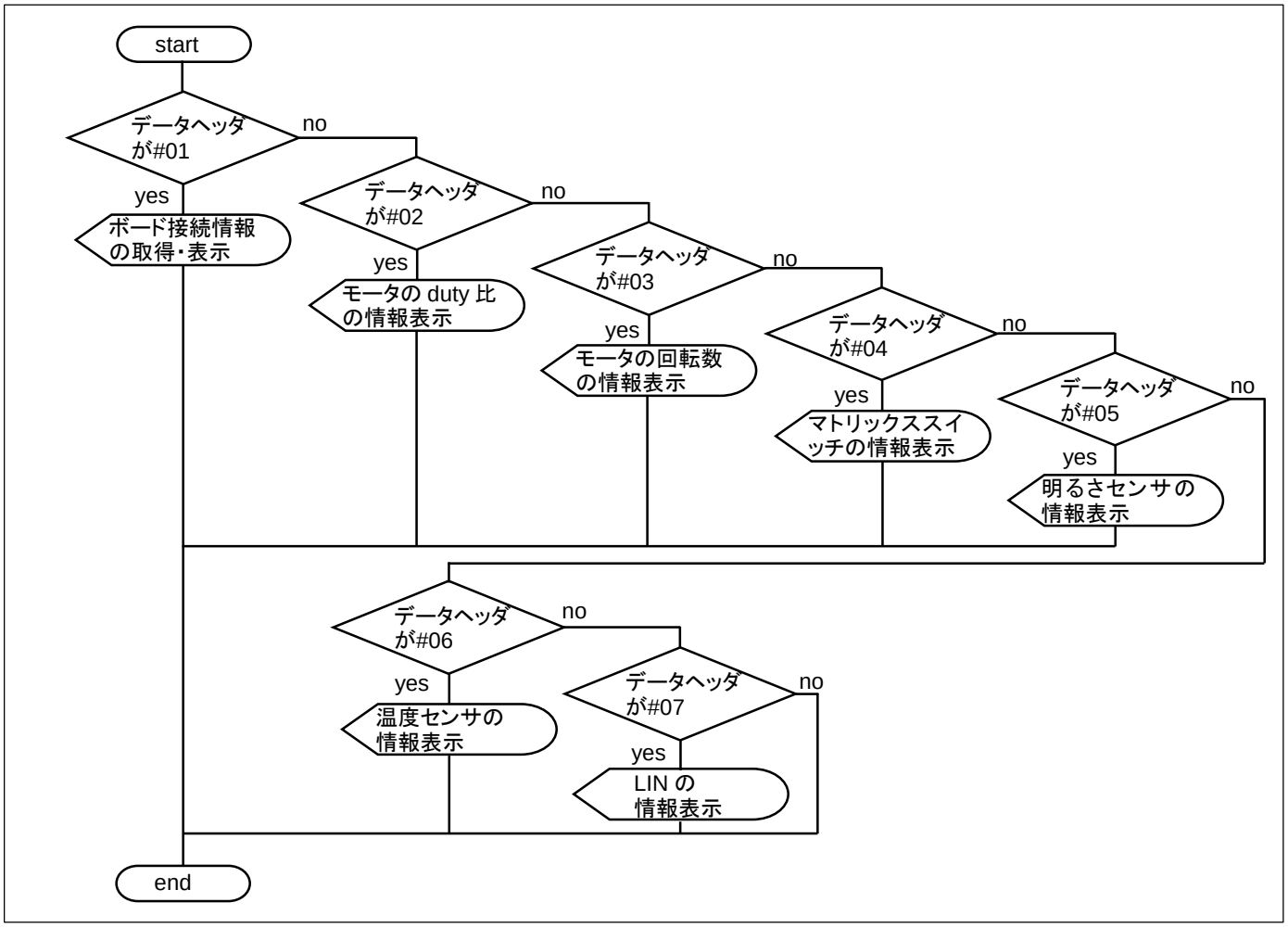
HSB_CAN_MULTI_4 側では、&(データ取得コマンド)に応答して、データを送信しますので、PC 側のアプリケーションでは定期的(500ms に 1 回)、'&'を COM ポートから送信します。

7.1.3. COM データ受信(OnReceived())



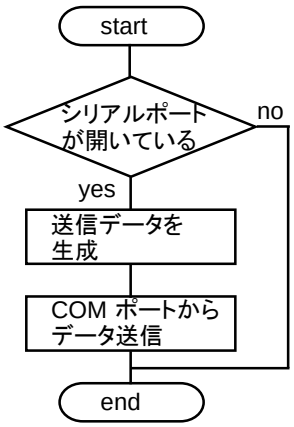
データ受信を行った際には、1 行単位(¥r¥n まで)で読み出しを行い、受信データの処理関数に渡します。

7.1.1. 受信データ処理(ReceiveDataHandling())



タイマで'&'を送信すると、HSB_CAN_MULTI_4 側から、#01, #02, ...で始まる各種データが送られてきます。これらのデータを、アプリケーション上の textBox に情報を表示させる処理を行っています。

7.1.2. 送信ボタン(button2_Click())

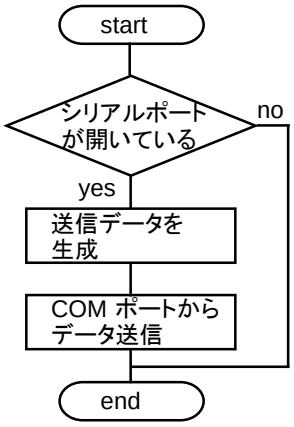


$00000201000132000000000000¥r¥n

$: CAN 経由でのコマンド送信
00000201: CAN-ID
00: RTR=0x00（データフレーム）
01: コマンド
32: 設定 duty (numericUpDown1 ダイアログ)設定値（duty50%の時の値）
000000000000: フィルデータ
¥r¥n: 改行コード

送信ボタン(button2, button3)は、PC から HSB_CAN_MULTI_4 に対し、$コマンド（CAN でメッセージを送信するコマンド）を送ります。button2 は、数値入力のダイアログ(numericUpDown1)の値を埋め込んだデータを生成して、COM ポートから送信します。
（HSB_CAN_MULTI_4 は、受け取ったデータを基に、CAN バスにパケットを転送します。）

7.1.3. 送信ボタン(button4_Click())

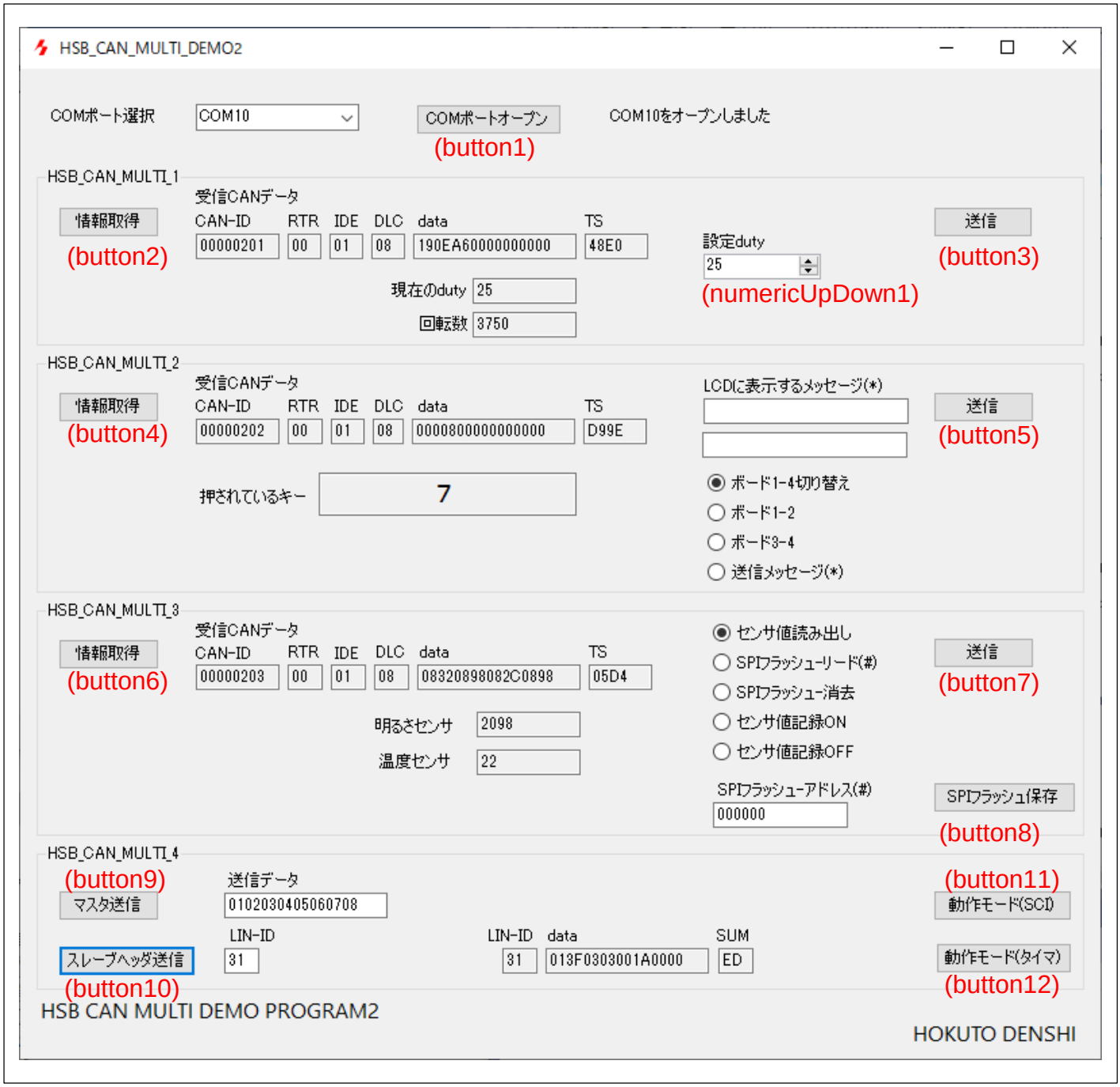


%0102030405060708¥r¥n

%: HSB_CAN_MULTI_4 に対する LIN データ送信
0102030405060708: LIN データ(textBox12 に設定したデータ)
¥r¥n: 改行コード

送信ボタン(button4)は、PC から HSB_CAN_MULTI_4 に対し、%コマンド（LIN でメッセージを送信するコマンド）を送ります。button4 は、データ入力のダイアログ(textBox1)の値を埋め込んだデータを生成して、COM ポートから送信します。
（HSB_CAN_MULTI_4 は、受け取ったデータを基に、LIN バスに MASTER レスポンス送信を行います。）

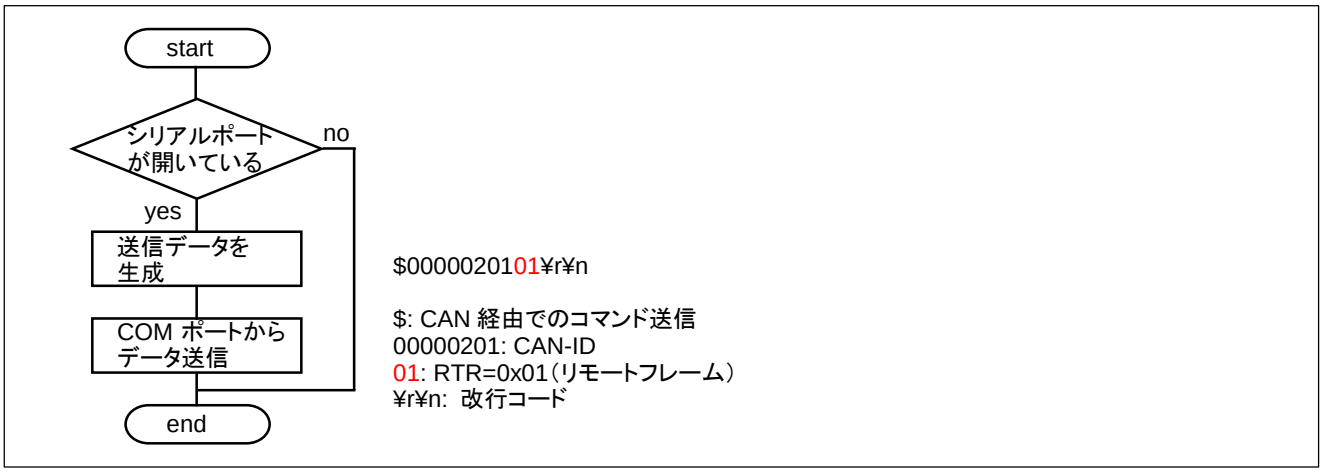
7.2. HSB_CAN_MULTI_DEMO2



もう一種類のアプリケーションの動作です。

COM ポートを開く動作(button1)は、HSB_CAN_MULTI_DEMO と同じです。

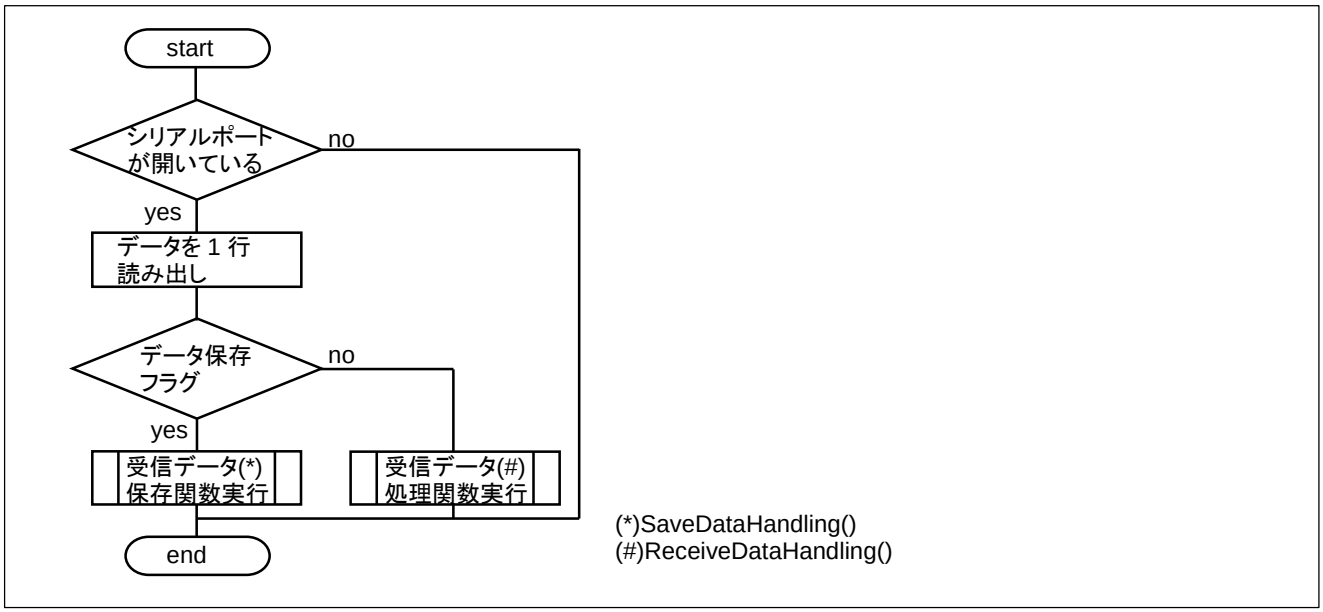
7.2.1. 情報取得ボタン(button2_Click())



情報取得ボタン(button2, 4, 6)は、CAN バスにリモートフレーム要求を送信するボタンです。button4 は CAN-ID が 0x00000202, button6 は CAN-ID が 0x00000203 に変わります。
（HSB_CAN_MULTI_4 は、受け取ったデータを基に、CAN バスにパケットを転送します。）

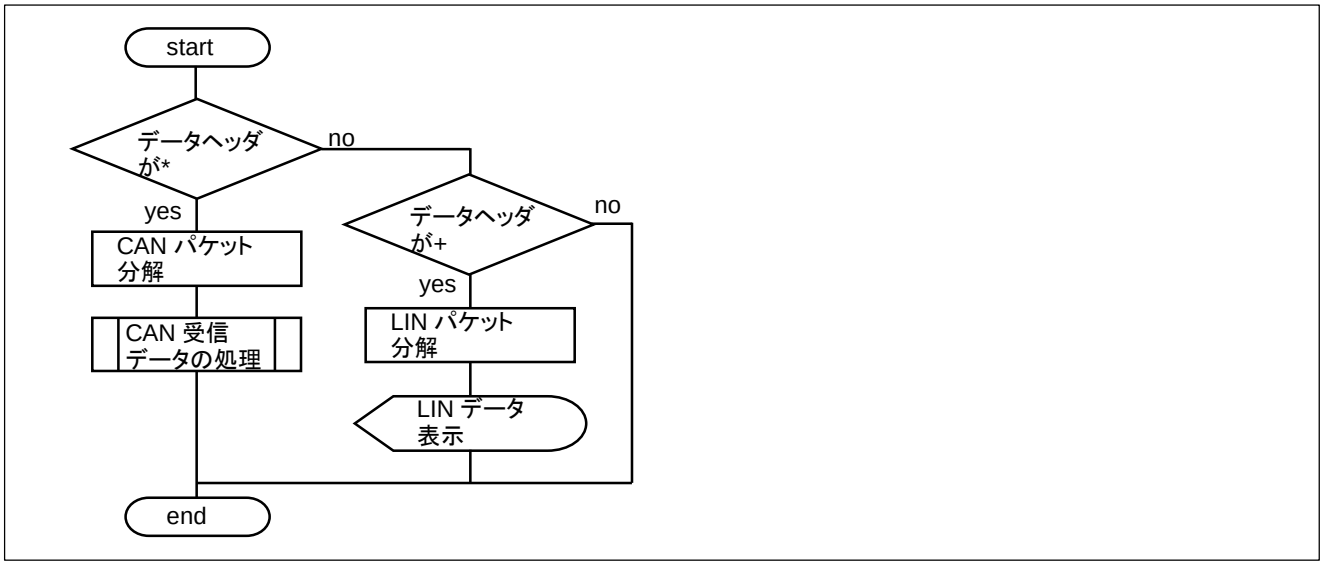
HSB_CAN_MULTI_1~3 のボードは、リモートフレームを受け取ると、CAN のデータフレームを送出します。そのデータフレームは HSB_CAN_MULTI_4 が UART で PC に転送する動作となります。

7.2.2. COM データ受信(OnReceived())



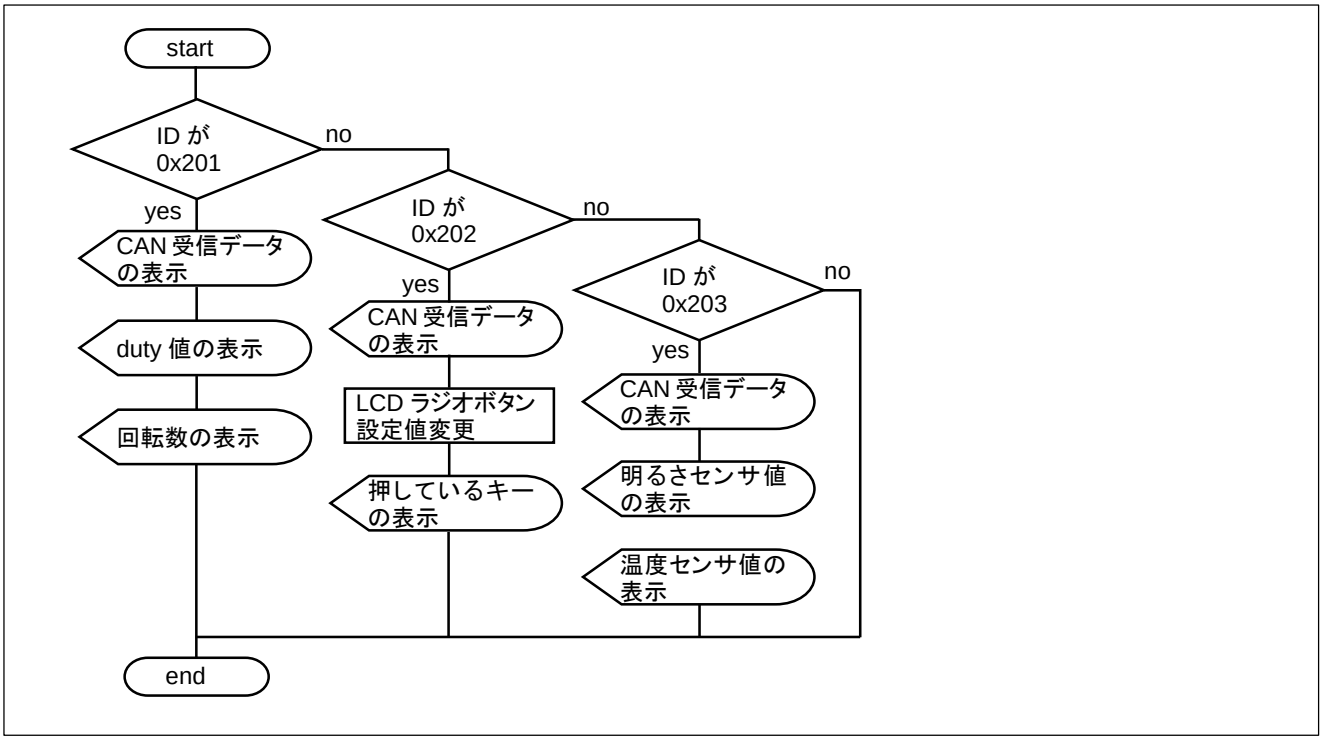
データ受信を行った際には、1 行単位（¥r¥n まで）で読み出しを行い、受信データの処理関数に渡します。本アプリケーションでは、受信したデータを CSV ファイルに保存する処理もあり、そちらの場合は SaveDataHandling()関数。ファイル保存以外のデータは、ReceiveDataHandling()関数で処理します。

7.2.3. 受信データ処理(ReceiveDataHandling())



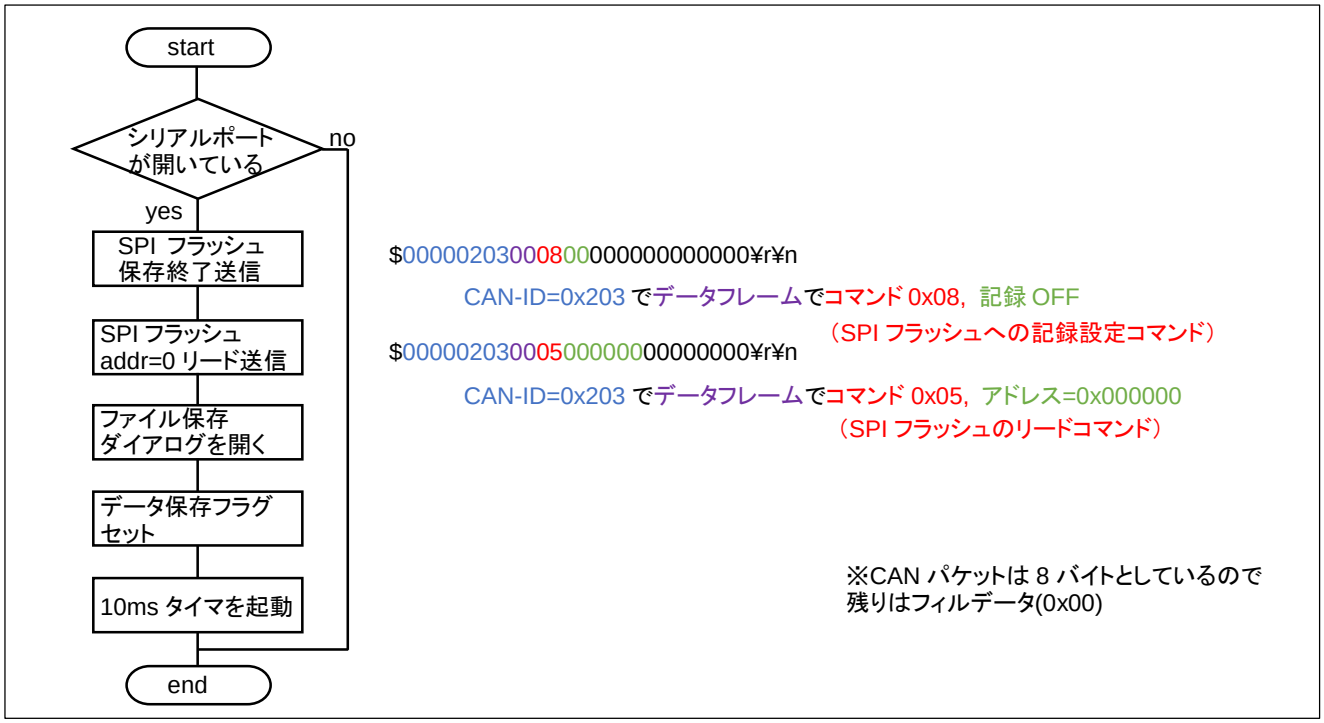
本アプリケーションでは、CAN のデータパケット（ヘッダ*）と LIN のデータパケット（ヘッダ+）を取り扱います。

－CAN 受信データの処理－



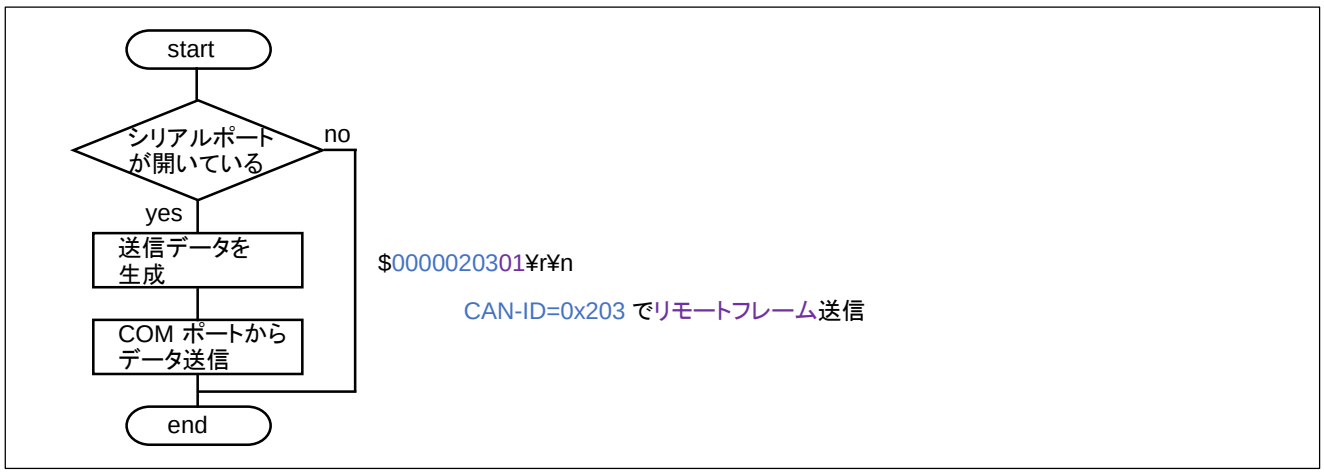
CAN データの表示は、HSB_CAN_MULTI_DEMO と同様です。CAN の ID により、表示内容を選ぶ点が異なります。

7.2.4. SPI フラッシュ保存処理(button8_Click())



SPI フラッシュ保存のボタンを押すと、CAN バス経由で HSB_CAN_MULTI_3 ボード(CAN-ID=0x203)に対しては、SPI フラッシュへの記録を OFF にするコマンド、SPI フラッシュのリードコマンド（アドレス 0 番地から）を送信します。その後、ファイル保存のダイアログボックスを開いて、データ保存用のファイルをオープンします。

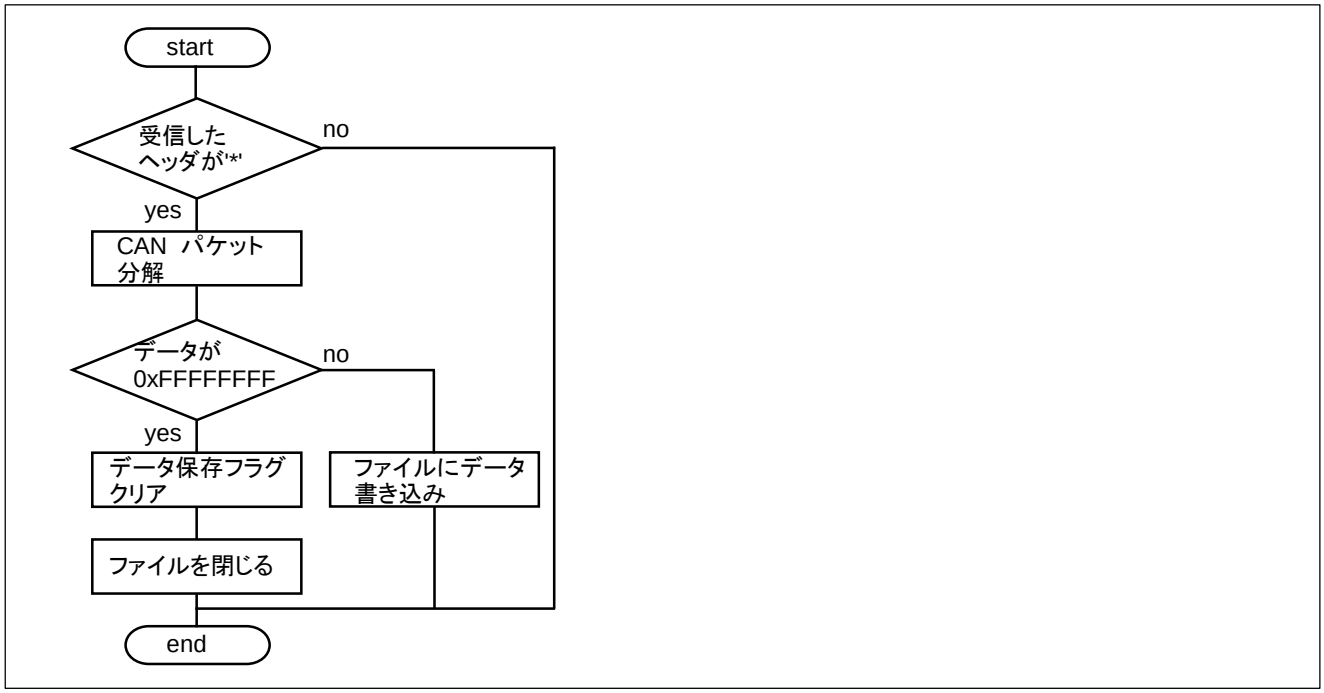
－10ms タイマー－



10ms タイマは、HSB_CAN_MULTI_3 ボードに対して、リモートフレームを送信します。
（HSB_CAN_MULTI_3 ボードは、リモートフレームを受け取ると、現時点のモードが SPI フラッシュのリードになっているので、SPI フラッシュのデータを返送します。）

－データ保存処理　SaveDataHandling()－

　データ保存フラグがセットされている場合で、COM ポート経由でデータ受信した際は、こちらの関数に飛んできます。



・SPI フラッシュ内のデータ

addr	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x01	0x23	0x08	0x66	0x01	0x24	0x08	0x66	0x01	0x25	0x08	0x66	0x01	0x26	0x08	0x66
0x10	0x01	0x27	0x08	0x98	0x01	0x28	0x08	0x98	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF

　SPI フラッシュの記録 ON にした場合、1 秒毎に

明るさセンサ（フォトダイオード）のデータ（2 バイト、0x0~0xFFF の範囲のデータ）

温度センサのデータ（2 バイトデータ、0x0866=2150→21.5℃）

が、HSB_CAN_MULTI_3 の SPI フラッシュメモリ上に記録されていきます。

　CAN 経由のデータ読み出しの場合、1 回の読み出しで 8 バイト(addr=0x0~0x7)のデータを読み出します。

　読み出したデータが 0xFFFF, 0xFFFF の場合、センサの情報が記録されたデータではないので、データ保存の処理を終了します。（上記例では 0x17 までに、6 組のデータが記録されたデータ。0x18~は、SPI フラッシュの消去後の値 0xFF）

－保存したファイル例－

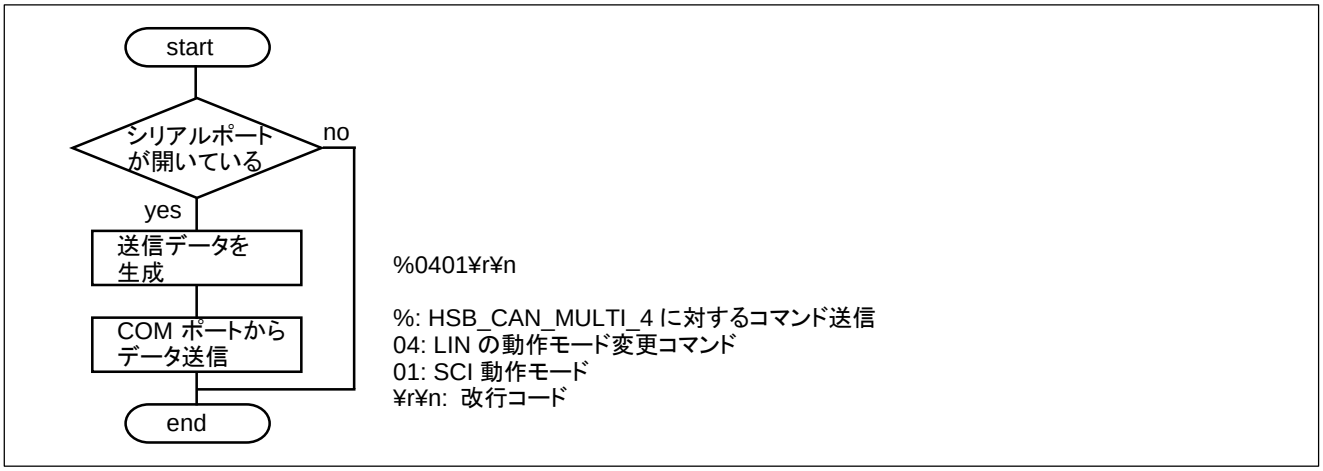
```
serial,bright,temp
0,291,21.5
1,292,21.5
2,293,21.5
3,294,21.5
4,295,22
5,296,22
```

ファイルは、CSV 形式となり、
シリアル番号(0~)，明るさセンサの値（10 進数に変換），温度センサの値（℃に変換）
の、1 行 2 データ（4 バイトのデータが 1 行）で保存されます。

データの読み出し、ファイルへの保存は、10ms のタイマのタイミングとなります（10ms 毎に 8 バイト、2 組のデータを受信）。

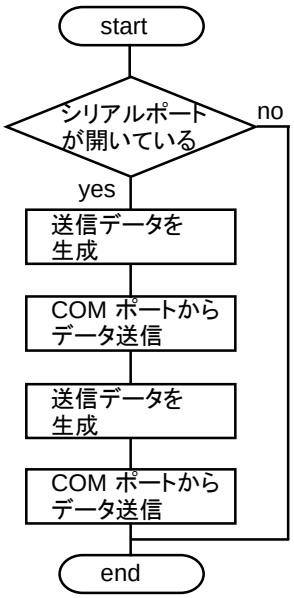
（1 時間分 3600 点記録されたデータをファイルに保存するのに、18 秒程度掛かります。）

7.2.5. 動作モード(SCI)ボタン(button11_Click())



動作モード(SCI)ボタンは、HSB_CAN_MULTI_4 ボードに対する指示コマンド(%)を送信します。
（HSB_CAN_MULTI_4 が上記コマンドを受け取ると、LIN の動作モードがタイマモードから、本アプリケーションで制御される様に切り替わります。）

7.2.6. マスタ送信ボタン(button9_Click())



送信データボックスに設定した値
%0101020304050607 08¥r¥n

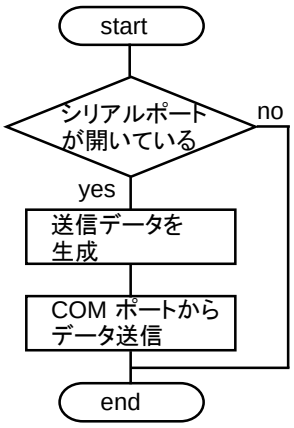
%: HSB_CAN_MULTI_4 に対するコマンド送信
01: HSB_CAN_MULTI_4 の送信データ設定コマンド
0102030405060708: LIN 送信データ
¥r¥n: 改行コード

%03¥r¥n

%: HSB_CAN_MULTI_4 に対するコマンド送信
01: HSB_CAN_MULTI_4 から MASTER レスポンス送信を行うコマンド
¥r¥n: 改行コード

（動作モードを SCI に変更後）マスタ送信ボタンを押すと、送信データのボックスに設定した値を、HSB_CAN_MULTI_4 ボードに送信します。続いて、MASTER ヘッダ/レスポンス送信のコマンドを送信します。（HSB_CAN_MULTI_4 が上記コマンドを受け取ると、LIN の MASTER ヘッダ/レスポンス送信を行います。）

7.2.7. スレーブヘッダ送信ボタン(button10_Click())



LIN-ID ボックスに設定した値
%0431¥r¥n

%: HSB_CAN_MULTI_4 に対するコマンド送信
01: HSB_CAN_MULTI_4 から MASTER ヘッダ送信を行うコマンド
31: ヘッダに含まれる LIN-ID
¥r¥n: 改行コード

（動作モードを SCI に変更後）スレーブヘッダ送信ボタンを押すと、LIN-ID のボックスに設定した値を、HSB_CAN_MULTI_4 ボードに送信します。
（HSB_CAN_MULTI_4 が上記コマンドを受け取ると、LIN の MASTER ヘッダ送信を行います。）

取扱説明書改定記録

バージョン	発行日	ページ	改定内容
REV.1.0.0.0	2023.6.6	—	初版発行

お問合せ窓口

最新情報については弊社ホームページをご活用ください。

ご不明点は弊社サポート窓口までお問合せください。

株式会社 北斗電子

〒060-0042 札幌市中央区大通西 16 丁目 3 番地 7

TEL 011-640-8800　FAX 011-640-8801

e-mail：support@hokutodenshi.co.jp (サポート用)、order@hokutodenshi.co.jp (ご注文用)

URL:https://www.hokutodenshi.co.jp

ルネサス エレクトロニクス社　RX231, RL78/F15, RA2L1 搭載
HSB シリーズ応用キット

CAN マルチネットワークボード取扱説明書
ソフトウェア編

株式会社 北斗電子
 2023 年 6 月 6 日改訂 REV.1.0.0.0 (230606)