SmartRA 学習キット
チュートリアル 5

ルネサス エレクトロニクス社　RA マイコン搭載
HSB シリーズマイコンボード　評価キット

-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、アクセスランプの点灯中に電源を切ったり、パソコンをリセットをしないでください。

製品の故障や、データ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. RA2L1_SCI

SCI, UART 等の呼び方をされる通信の規格で、日本語だと調歩同期通信などと言います。

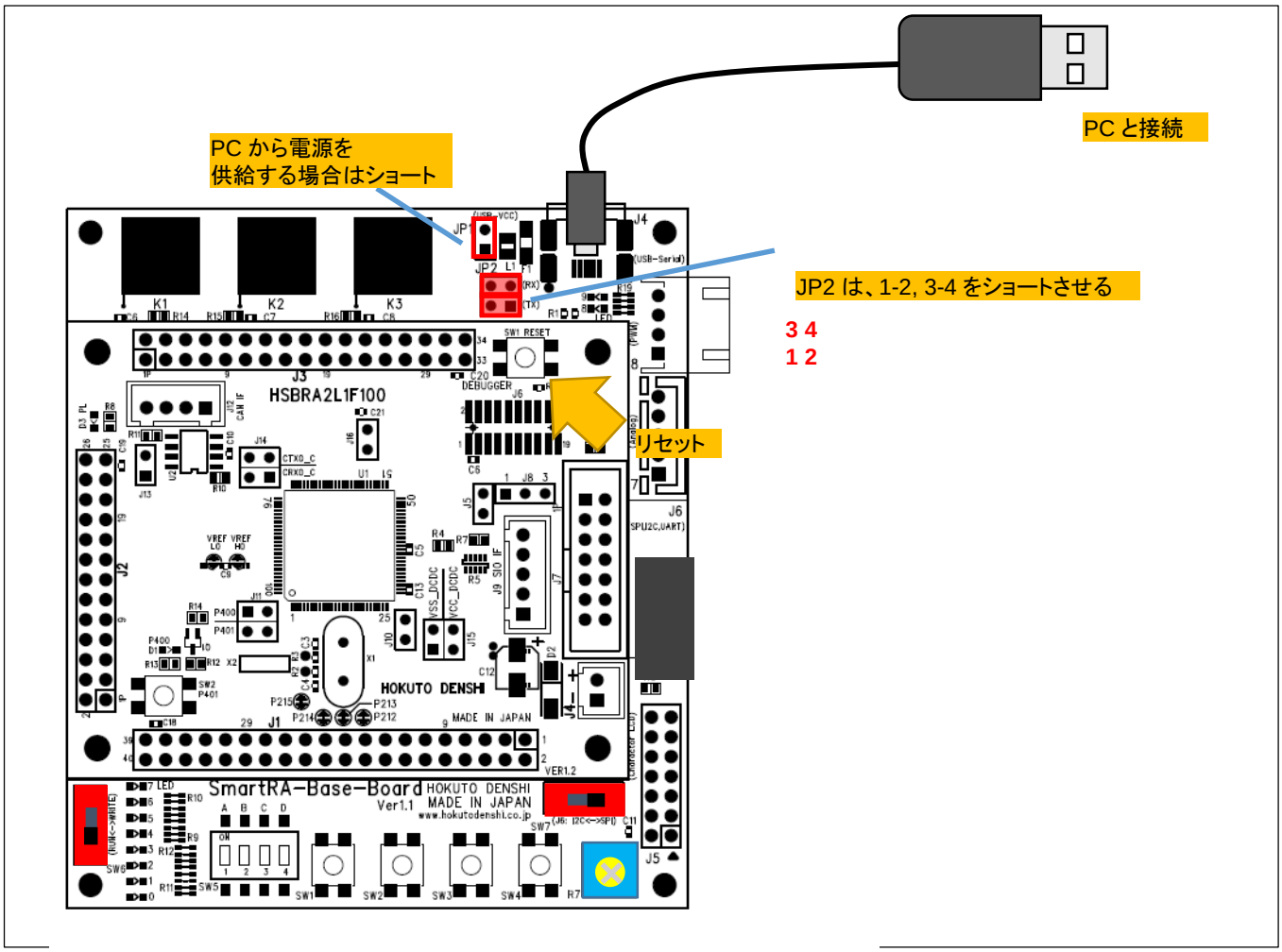
マイコンボード同士の通信や、マイコンボードと何か外部の機器とデータのやり取りを行う場合。PC とつないで、データのやり取りを行う場合などに使われます。

センサーモジュールのアクセスは、I2C や SPI が使われる事が多いですので、SCI(UART)が使われるのは、対 PC や GPS モジュールとの接続などです。

マイコンから何か情報を出力したいという時に、まず最初に思い浮かぶのが SCI(UART)で、非常に幅広く使われています。通信系のプログラムで最初に覚えるのが、SCI(UART)だと思います。

最近の PC には付いていない事が多いですが、シリアルポート(D-sub 9pin)も、プロトコル（通信手順）としては、SCI(UART)です。（但し、マイコンの SCI と PC のシリアルポートは電気的特性が合わないです…直接接続する事はできません）

1.1. プログラムの動作



SCI(UART)の信号は、ベースボード上の J4(USB-miniB)コネクタにつながっていますので、キット付属の USB ケーブルで PC とつないでください。ベースボード上に、USB-Serial 変換回路が搭載されており、SCI(UART)の信号は、USB 経由で PC と接続されます。

PC 側は、USB-Serial 変換回路のドライバが見つかれば、仮想 COM ポートとして認識します。

自動的にハードウェアの認識がされない場合は、ドライバをダウンロードしてインストール願います。

ドライバのダウンロードは、prolific Web

http://www.prolific.com.tw/

から、下記を辿って、ダウンロード願います。

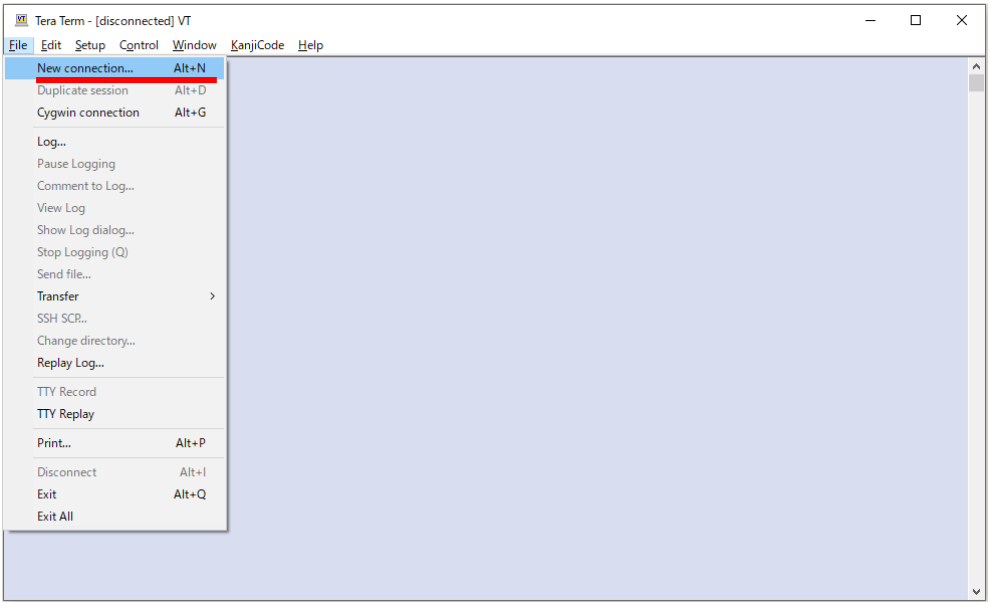
Products Application

SIO(Smart-IO)

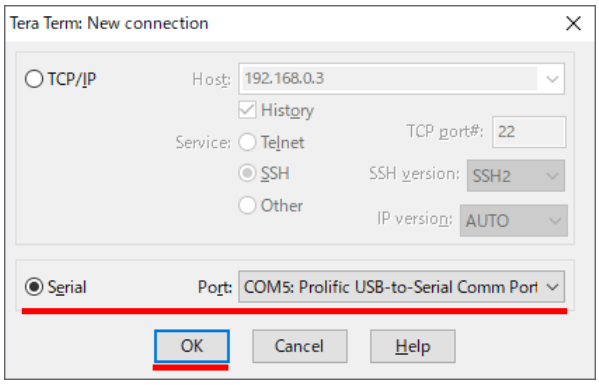
USB to UART/Serial/Printer

ベースボードには、USB シリアル変換 IC として、prolific 社製、PL2303HXD が搭載されています。

仮想 COM ポートは、TeratermPro 等のターミナルソフトで開いてください。



TeratermPro を使う場合、New Connection

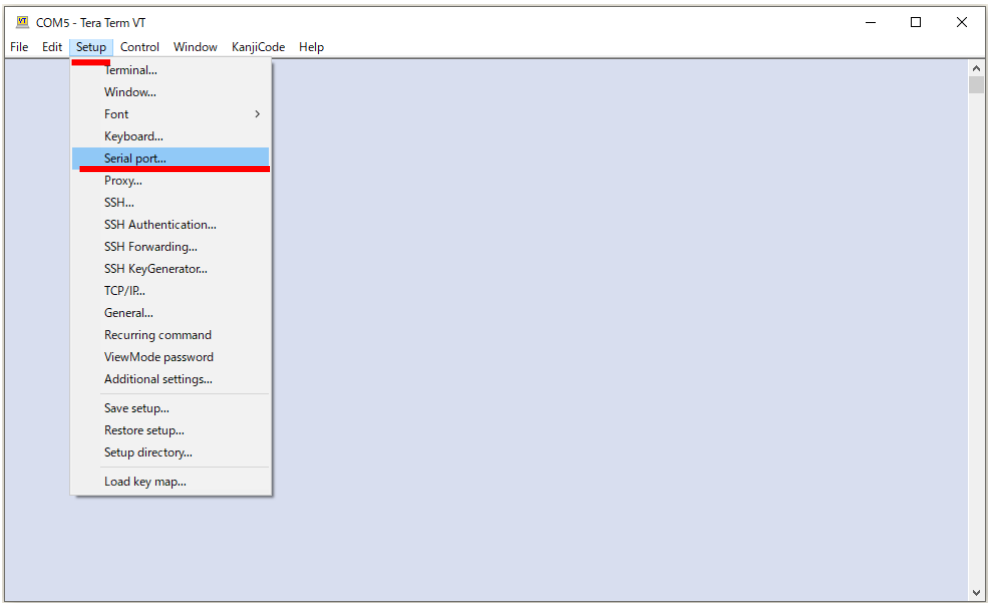


Serial を選択。Port として、Prolific USB-to-Serial Comm Port を選択してください。このハードコピーでは、COM5 となっていますが、番号は環境により変わります。

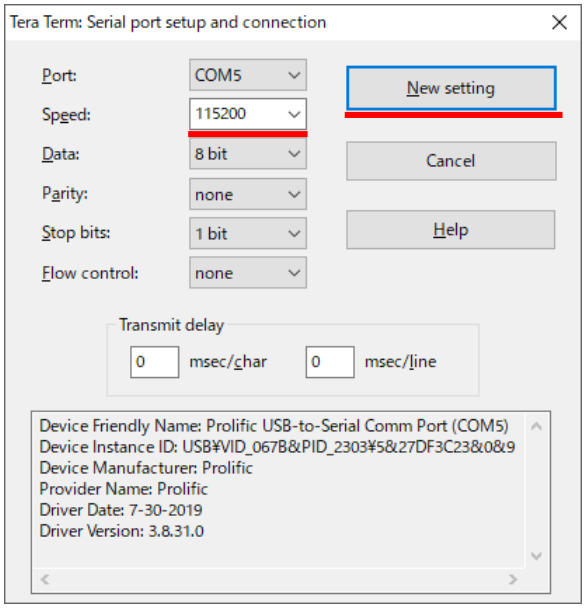
OK を押す。

Prolific USB-to-Serial Comm Port が選択肢にない場合は、ドライバがインストールされていないか、USB ケーブルの接続に問題がある事が考えられます。

※COM?の番号は、USB の差す場所によっても変わりますので、昨日までと番号が変わっているという事も考えられますので、注意願います。



Setup – Serial port を選択

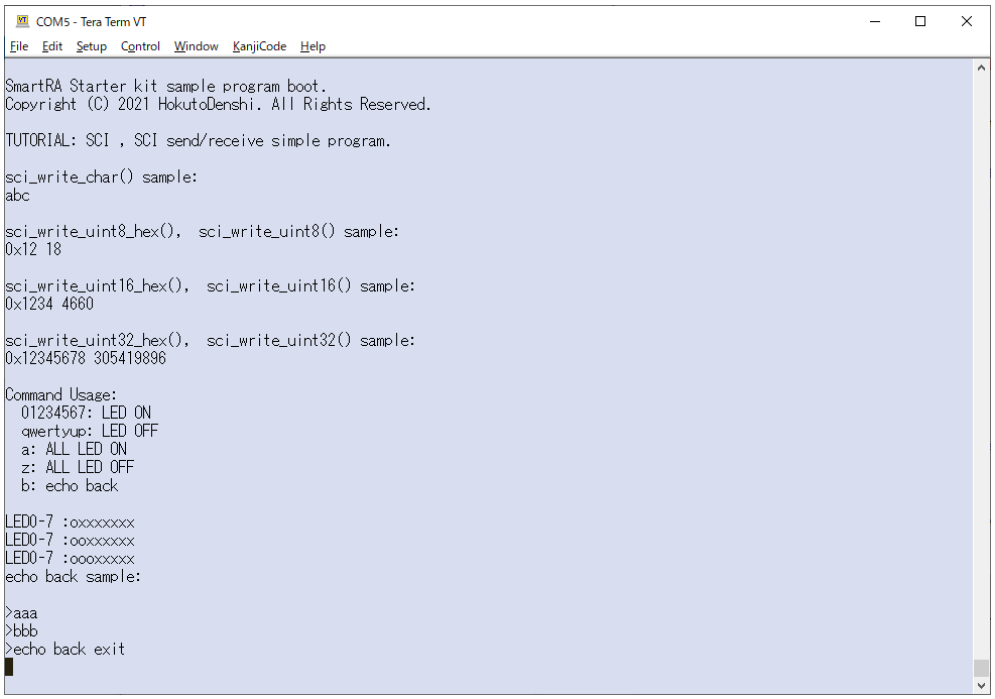


設定項目	設定値	備考
Port	COM?	USB ポートの位置等で変わります Prolific USB-to-Serial Comm Port として認識されている番号を選択 （シリアルポートを開く段階で選択済みの 場合は変更不要）
Speed	**115200**	
Data	8bit	デフォルト
Parity	none	デフォルト
Stop bits	1bit	デフォルト
Flow control	none	デフォルト

Speed を 115200 に設定してください。他の設定は変更不要です。この設定の「データ 8bit、パリティなし、ストップビット 1(8-N-1)」は、マイコン側と合わせる必要があります。（マイコン側で設定を変えた場合はこの設定を変更しなければなりません。8-N-1 の設定は、一番メジャーな設定です。

設定後、New setting（または New open）を押して設定画面を閉じてください。

端末の速度等設定後、マイコンボードのリセット(M-SW1)を押すと、端末に文字が表示されます。



・abc を表示

・0x12 を hex と 10 進表記で表示

・0x1234 を hex と 10 進表記で表示

・0x12345678 を hex と 10 進表記で表示

その後コマンド受付モードとなります。

文字化けや表示が出ない場合は、ポート番号(COM?)と速度設定が合っているかを確認してください。

文字化けした場合は、端末のリセット
(TeratermPro 上で)
Control - Reset terminal
Edit - Clear buffer
を実行し、再度マイコンボードのリセットボタンを押してください。

コマンドはキーボードから 1 文字入力で以下の動作となります。

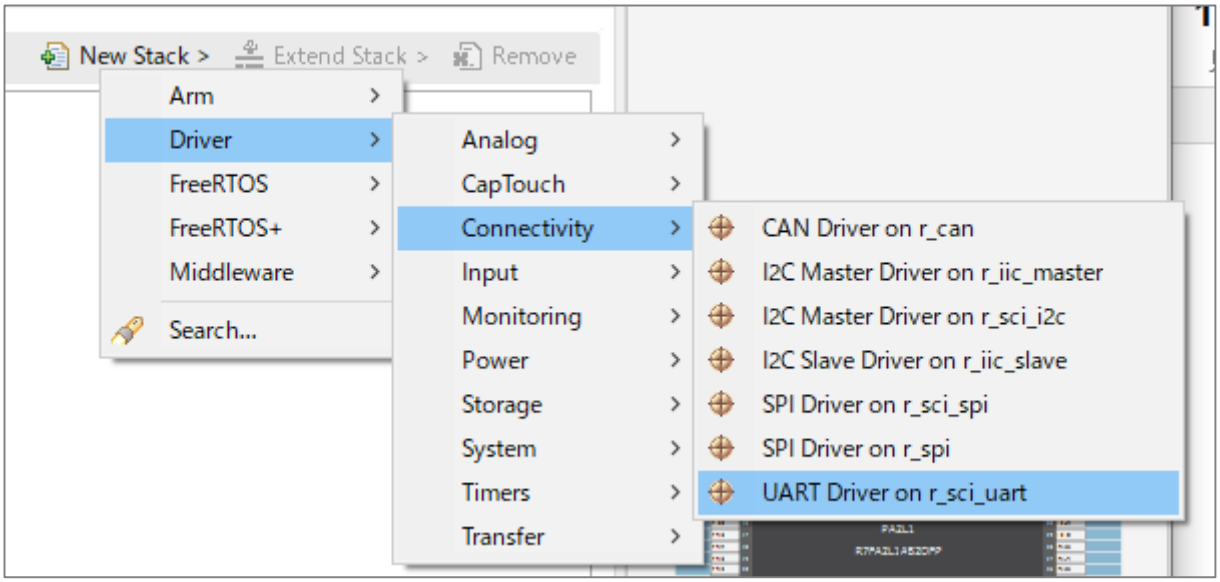
0~7: LEDn(n=0~7)を点灯させる
qwertyup: LEDn を消灯させる(q:LED1 を消灯, w:LED1 を消灯, …u:LED7 を消灯, p:LED0 を消灯)
(1~7, 0 の右下の qwertyup キーで LEDn の消灯です)
a: LED0~LED7 を点灯
z: LED0~LED7 を消灯
b: キーボードから入力した文字をエコーバックする(CNTL-C で終了)
※CNTL-C は端末によってはうまく認識しない可能性もあります

SCI の機能を使用すると、マイコンボードに指令を出したり、マイコンボードの動作やデバッグの情報を出力できるので、とても便利です。

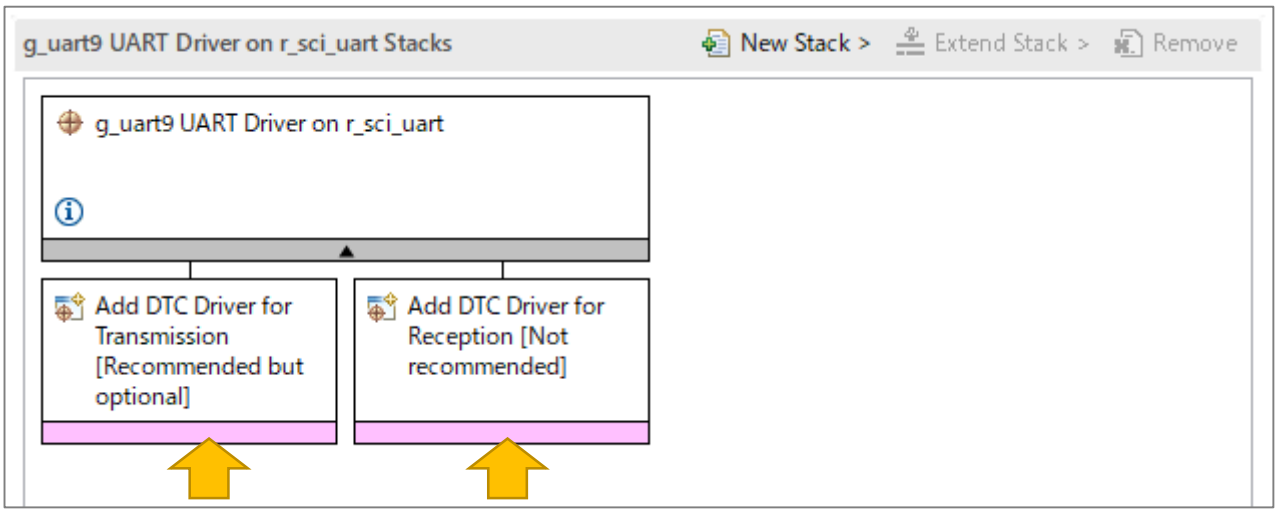
1.2. FSP の設定

FSP の設定は、

New Stack – Driver – Connectivity – UART Driver on r_sci_uart



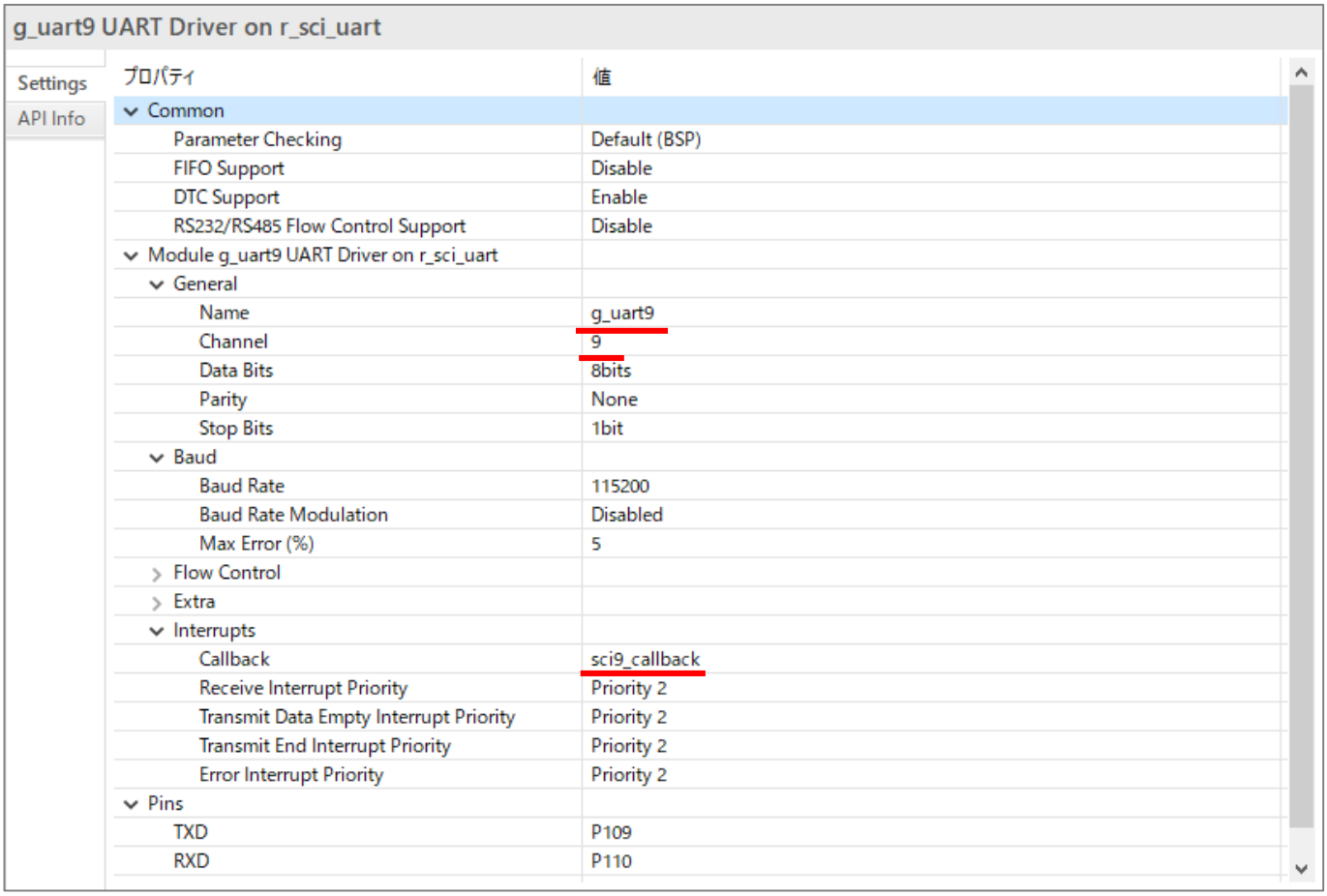
で追加します。



UART は、Stack にさらに DTC の Stack をぶら下げられるようになっています。

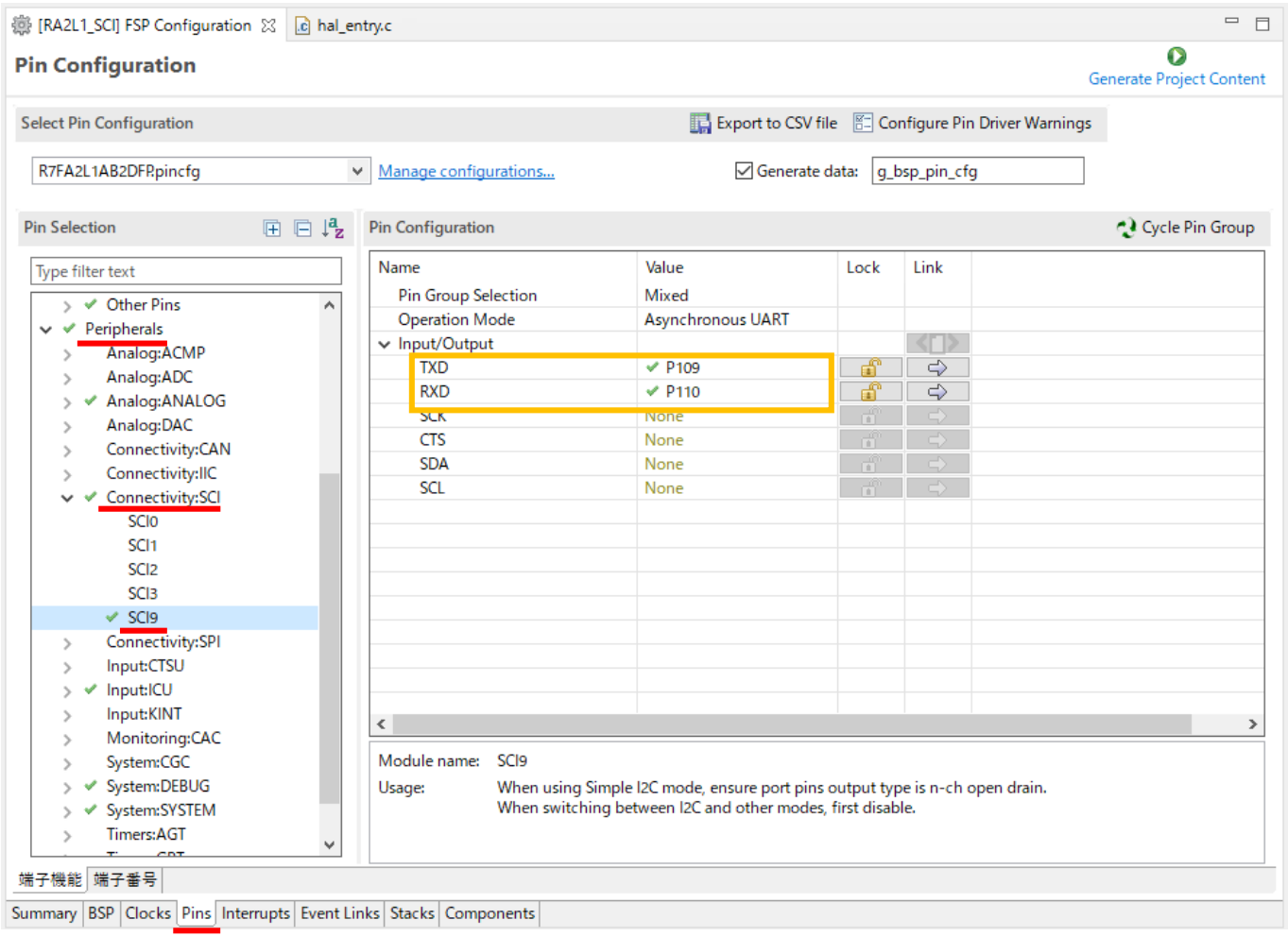
DTC は、Direct Transfer Controller で、CPU の介在なしにデータ転送を行うモジュールです。ここでは、使用しない事とします。(2 章で DTC を使った場合の CPU 効率を参考までに載せています)

UART Driver　Stack の設定は以下の様にしています



項目	設定値	備考
Name	g_uart9	今回使用するのが SCI9 なので名称を g_uart9 に設定しています
Channel	9	今回使用するのが SCI9 なので 9 に設定します
Data Bits	8	デフォルトのまま[PC のターミナルソフトと合わせる必要あり]
Parity	None	デフォルトのまま[PC のターミナルソフトと合わせる必要あり]
Stop Bits	1	デフォルトのまま[PC のターミナルソフトと合わせる必要あり]
BaudRate	115200	通信速度[PC のターミナルソフトと合わせる必要あり]
Callback	sci9_callback	値は任意です
Receive Interrupt Priority	Priority 2	設定値は任意です
Transmit Data Empty Interrupt Priority	Priority 2	設定値は任意です
Transmit End Interrupt Priority	Priority 2	設定値は任意です
Error Interrupt Priority	Priority 2	設定値は任意です

こちらは、マイコン側の設定ですが、ターミナルソフトで設定した、8-N-1 の設定はこのマイコン側の設定が基になっています。



SCI9 には、複数の端子が割り当てられています（複数の端子から選択が可能です）。

マイコン側は、TXD9(P109), RXD9(P110)を使用します。

FSP の Pin タブの Peripheral – Connectivity:SCI – SCI9 の

TXD　P109
RXD　P110

を選んでください。TXD は、

P203, P109, P602

の中から選択が可能です。ここで、P203 等別な端子を選ぶと、信号はそちらの端子に送られてしまいます。USB 経由で PC とつながっているのは、P109 端子ですので、正しく選択する事は重要です。

1.3. SCI(UART)の動作

SCI(UART)では、いくつかの動作モードがありますが、このサンプルプログラムでは、

・調歩同期（非同期、Asynchronous）
・フロー制御なし

というモード（一番一般的かと思われます）で使用しています。この場合使用する信号線は、TXD, RXD の 2 本です。

USB-Serial 変換部分の回路は図 1-1 の様になっています。

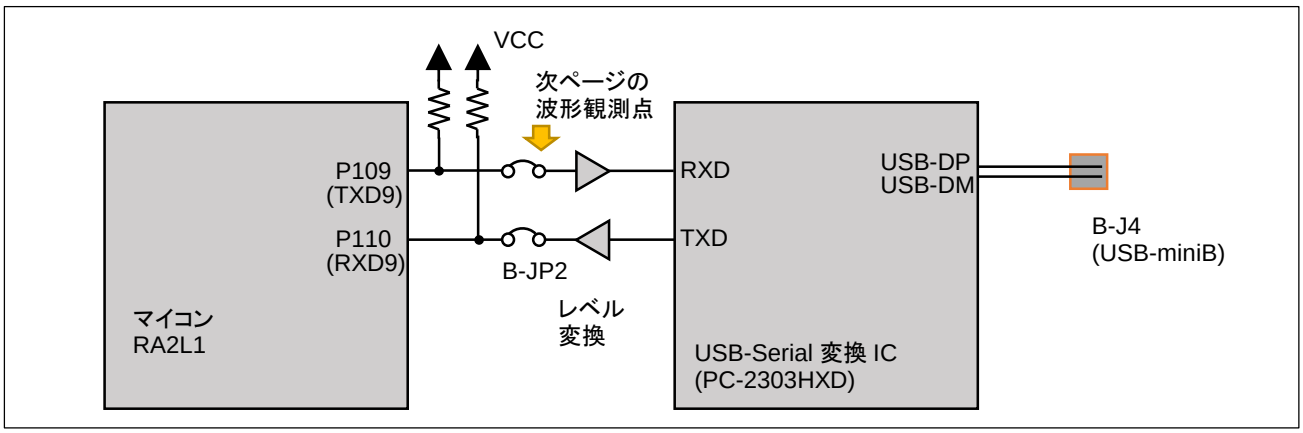


図 1-1 USB-Serial の回路

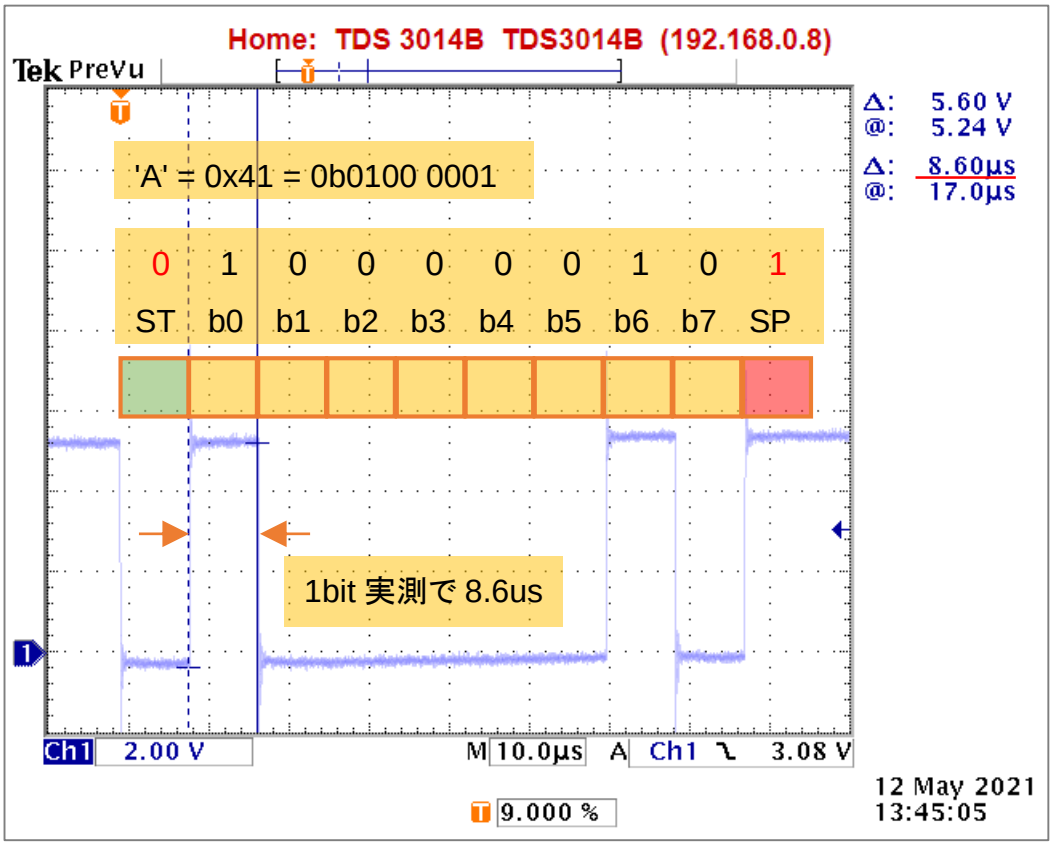
TXD は、出力端子でマイコンから外部（ここでは相手先は PC）に対してデータを送信します。RXD は受信端子で、PC からのデータ（サンプルプログラムでは、LED 制御のコマンド等）を受信します。

※双方向の通信が不要であれば、TXD, RXD のどちらか一方を使うこともできます

フロー制御は、データを受信できる準備が整っているかを通信相手に通知する機能、同様に通信相手がデータ受信の準備ができているかを知る機能ですが、サンプルプログラムでは使っていません。

1.4. SCI(UART)の信号波形例

マイコンからデータを出力した場合の、信号波形例を示します。観測点は、TXD9 の端子です。これは、マイコンから PC に'A'の文字を送った場合の波形です。



'A'は、ASCII コードで、0x41 なので、送信するデータとしては 0b 0100 0001 となります。データは、LSB ファースト（下位ビットから先に送る）方式となります。

SCI(UART)のプロトコル（通信手順）では、1 文字（キャラクタ）8bit のデータに、スタートビット(ST)とストップビット(SP)が追加されて、トータルで 10bit のデータとなります。

スタートビットは、1bit で、データは常に 0。ストップビットは、（サンプルプログラムの）設定では 1bit、データは常に 1 となります。

データを送っていない時は、TXD のレベルは H となります。

ビットレートは、115,200bps に設定しているので、計算上は 1bit あたり 8.68us となります。
※上の波形では、実測 8.60us となっていますが、スケールを拡大して実測すると 8.66us でした

SCI(UART)では、送信(TXD)と受信(RXD)は（端子としても、動作としても）独立していますので、送信中の受信動作もプロトコル上は問題ありません（データ送信中は、送信に専念するプログラムを作成した場合はその限りではありません。送受信が同時に行える（全二重通信ができる）かはプログラム次第です）。

1.5. 速度設定と誤差に関して

本サンプルプログラムでは、115,200bps に設定していますが、マイコンからは正確に 1 秒間に 115,200bit の信号が出力されるわけではありません。必ず、誤差が生じます。SCI(UART)に限った話ではないのですが、通信のクロックはマイコンの内部クロック（SCI では、PCLKB）を基準に動作します。信号波形が切り替わるタイミングは、PCLKB(=24MHz)を基準に、何カウントというタイミングとなるので、（余程運が良くないと）設定値ちょうどにはなりません。

※ビットレートを 9600 や 38400 といった数値ぴったりにするために、マイコン外付けの水晶振動子を 12.288MHz などの値（一見キリの悪い数値）を選ぶ事もあります

RA2L1(PCLKB=24MHz)で、115,200bps に設定した場合

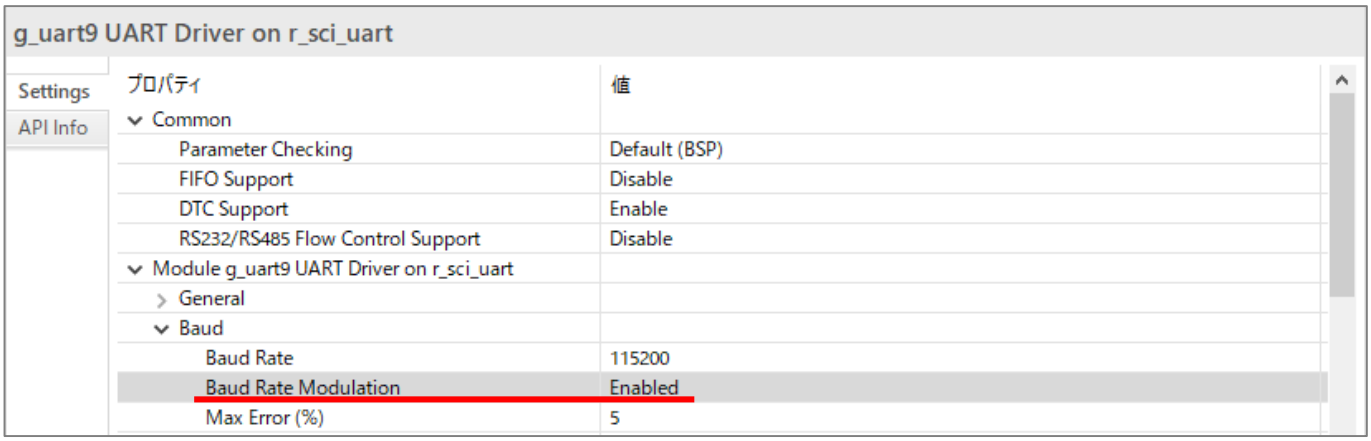
レジスタ設定値	SEMR.ABCSE	0
	SEMR.ABCS	0
	SMER.BGDM	1
	BRR	12
実際のビットレート[bps]		115384.6
誤差[%]		0.16

というような実信号レート、誤差となります。これは、PCLKB に誤差を含まない値なので、実際には PCLKB（=HOCO の分周）の誤差±1%が加算されます。

ビットレート設定で生じる誤差に関しては、クロック周波数(PCLKB)とビットレート（正確にはビットレートの逆数、1bit の時間）の関係で決まりますので、設定したビットレートで、どの程度の誤差が生じるのかは（本来は）意識する必要があります。（例えば、PCLKB=24MHz の場合は、60,000bps は誤差 0 となり、80,000bps は誤差-1.32%となります）

実ビットレートの計算式と、誤差はマイコンのハードウェアマニュアルに記載がありますので、そちらを参照してください。

なお、マイコンには「ビットモジュレーション機能」という、速度を微調整する機能があります。



上記を有効にした場合は、

レジスタ設定値	SEMR.BRME	1(*1)
	SEMR.ABCSE	0
	SEMR.ABCS	0
	SMER.BGDM	1
	BRR	10
	MDDR	216(*2)
実際のビットレート[bps]		115061.8
誤差[%]		-0.12

(*1)ビットモジュレーション有効化設定
(*2)補正値

使わない場合に比べて多少（0.16%→-0.12%）改善される様です。ビットモジュレーション機能を使う場合、補正値の計算が面倒ですが、FSP の設定のみで有効化できるのであれば、有効化して使用しても良いのかと思います。

※ちなみにですが、補正値を手計算で最適化すると

レジスタ設定値	SEMR.BRME	1
	SEMR.ABCSE	0
	SEMR.ABCS	0
	SMER.BGDM	1
	BRR	11
	MDDR	236
実際のビットレート[bps]		115234.4
誤差[%]		0.0298

FSP が算出するより、誤差が少ない値が見つかります。
（MDDR の値を探す場合は、単純なやり方としては、BRR を減じて、MDDR=255 から減らしていき、誤差 0 に一番近い点を探すというものかと思います。BRR=11 で最適解が見つかるのですが、何故か FSP ですと BRR=10 になっています。）

1.6. 本チュートリアルで使用している関数（抜粋）

sci_init

概要：SCI(SCI9)初期化関数
宣言：
　　void sci_init(void)
説明：
　・SCI9 の初期化
を行います

引数：
　　なし
戻り値：
　　なし

sci_write_char

概要：1 文字出力関数
宣言：
　　void sci_write_char(unsigned char c)

説明：
　・1 文字出力
を行います

引数：
　　c： 出力する文字コード
戻り値：
　　なし
使用例：
　sci_write_char('A');　　//A を出力する

sci_read_char

概要:受信関数
宣言:
　　unsigned char sci_read_char(void)

説明:
　・1 文字受信
を行います

引数:
　　なし
戻り値:
　　受信した文字コード
　　0xff は、受信データなしを表す
使用例:
　x = sci_read_char();　　//x に受信データを格納する

sci_write_str

概要:文字列出力関数
宣言:
　　void sci_write_str(char *str)

説明:
　・文字列出力
を行います

引数:
　　*str :　出力する文字列（¥0 終端）
戻り値:
　　なし
使用例:
　sci_write_str("Disp string¥n");　　//「Disp string[改行]」を出力する

sci_write_uint8_hex
sci_write_uint16_hex
sci_write_uint32_hex

概要: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:
　・hex（16 進数）での出力
を行います
8 は 8bit（16 進数で 2 桁）、16 は 16bit（16 進数で 4 桁）、32 は 32bit（16 進数で 8 桁）版です

引数:
　　c, s, l : 出力するデータ
戻り値:
　　なし
使用例:
　sci_write_uint8_hex(0x23);　　//「23」を出力する ※0x は付与されません

sci_write_uint8
sci_write_uint16
sci_write_uint32

概要:符号なし数値出力関数
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:
　・10 進数での出力
を行います
8 は 8bit、16 は 16bit、32 は 32bit の符号なし整数版です

引数:
　　c, s, l : 出力するデータ

戻り値:
　　なし
使用例:
　sci_write_uint8(123);　　//「123」を出力する

sci_write_int8
sci_write_int16
sci_write_int32

概要: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:
　・10 進数での出力
を行います
8 は 8bit、16 は 16bit、32 は 32bit の符号付き整数版です

引数:
　　c, s, l : 出力するデータ

戻り値:
　　なし
使用例:
　sci_write_uint8(-123);　　//「-123」を出力する

sci_write_flush

概要:出力バッファ出力完了待ち関数
宣言:
　　void sci_write_flush(void)

説明:
　・出力バッファに溜まっているデータの出力処理（出力完了までウェイト）
を行います

引数:

なし

戻り値:

なし

sci_read_buf_clear

概要:入力バッファクリア関数

宣言:

void sci_read_buf_clear(void)

説明:

・入力バッファに溜まっているデータのを空にする処理（溜まっているデータを読み出し済みとして処理）を行います

引数:

なし

戻り値:

なし

1.7. 入出力バッファに関して

本サンプルプログラムでは、出力バッファに 1024 文字分（1024 バイト）、入力バッファに 32 文字分のバッファを確保しています。

sci¥sci.h

```
/*----------------------------------------------------------------------
    定義
----------------------------------------------------------------------*/
#define SCI_SEND_BUF_SIZE   1024
#define SCI_RECV_BUF_SIZE   32
```

バッファの容量は調整可能です。RA2L1（本キットで採用されている、R7FA2L1AB2DFM）は、32kB の RAM を持っていますので、使用可能な RAM の範囲内で増やすことは可能です。

出力、入力バッファはリングバッファの構成となっています。

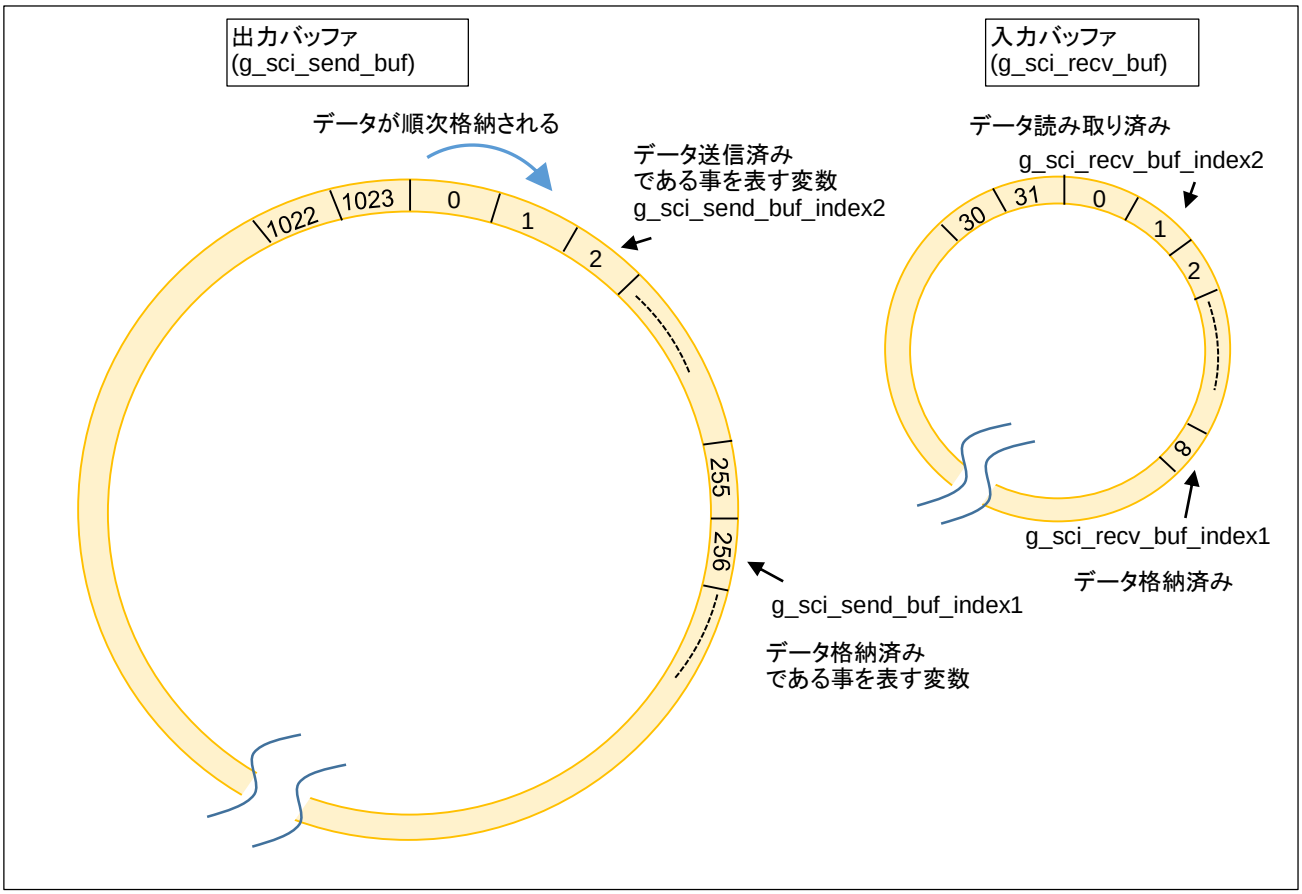


図 1-2 リングバッファ構成

リングバッファとは、バッファの最後（出力バッファだと 1023 バイト目 g_sci_send_buf[1023]）の次が 0 バイト目(g_sci_send_buf[0])となって、永遠に終わりが来ないデータ構造です。

出力の関数

sci_write_char()

を実行すると、データは g_sci_send_buf_index1 が指し示す次の箱（バッファ）に格納されます。（その際 g_sci_send_buf_index1 が 1 進みます）

g_sci_send_buf_index2 は、表示済み（マイコンからはデータ送信済み）を示すインデックスで、g_sci_send_buf_index1 == g_sci_send_buf_index2 となるまで（未表示のデータがなくなるまで）データを出力しつつ進んでいきます。

g_sci_send_buf_index1 が先行し、その後を g_sci_send_buf_index2 が追いかける形となります。

本サンプルプログラムでは、11,5200bps に設定しているので、1 文字（キャラクタ）を出力するのに、1/115200×10 = 86.8us 掛かります。（スタート・ストップビットがあるので、1 キャラクタ 10bit）

例えば、20 文字出力するのには、1.7ms（+オーバヘッド）の時間が掛かり、これはマイコンからすると結構長い時間です。文字出力が終わるまで、マイコンに送信処理のみを行わせる事は非効率なので、sci_write_char()等の文字表示関数では、バッファ（メモリ）へのデータコピーのみ行っておきます。実際のデータ送信は、1 キャラクタの送信が終わった時点で割り込み処理により、次に出力するデータを処理します。

※なお、送信バッファ(g_sci_send_buf)へのデータ格納(g_sci_send_buf_index1)が表示(g_sci_send_buf_index2)に追いついた場合（バッファがフルになった場合）は、本サンプルプログラムでは、格納データを捨てる（g_sci_send_buf には格納しない）動作となります。（表示される文字が歯抜けの状態となりますが、格納順と表示順の関係が入れ子（順番以外）になることはありません。）

受信側のバッファは、外部からデータが来ると、g_sci_recv_buf に格納していきます。データ格納のたびに、g_sci_recv_buf_index1 はインクリメント(+1)されます。sci_read_char()関数で 1 文字読み出しを行うと、g_sci_recv_buf_index2 がインクリメントされます。g_sci_recv_buf_index2 が g_sci_recv_buf_index1 に追いつくと、未読み出しのデータはない（バッファが空）という事となり、sci_read_char()の戻り値は 0xff（データなし）となります。

※データ格納(g_sci_recv_buf_index1)が、データ読み出し済み(g_sci_recv_buf_index2)に追いついた場合、外部から受信したデータは、未読み出しのデータに上書きされます。sci_read_char()では、古いデータは捨てられて、常に最新の 32 文字のデータにアクセス可能です。

※送信（新しいデータを捨てる）と受信（古いデータを捨てる）でバッファフル時の挙動が異なりますので、ご注意ください。

1.8. FSP で定義されている API 関数に関して

FSP で SCI のデータ送信と受信の API 関数が定義されています。本サンプルプログラムで作成した、

送信：sci_write_char()
受信：sci_read_char()

関数は、最終的には API 関数を使っているのですが、API 関数の使い勝手が悪いために、ユーザ定義関数を作ったという面もあります。

API 関数では、送信

```
R_SCI_UART_Write(&g_uart9_ctrl, &g_send_data, byte_count);
```

という関数です。第 2 引数に送信データ（の先頭アドレス）。第 3 引数に送信バイト数を指定します。API 関数で送信すると、実際の送信処理はバックグラウンドで行われるので CPU 効率が悪いという事はありません。問題は、この API 関数を複数回実行するケースです。

API 関数を使う場合、文字出力が完了（20 文字で 1.7ms）が終わるまで、次の関数呼び出しができません。その制約があると、実質的にバックグラウンド処理にはならないので、本サンプルプログラムでは、API 関数を裏で使いつつ、独自のリングバッファを設ける事としています。

また、API の受信関数は

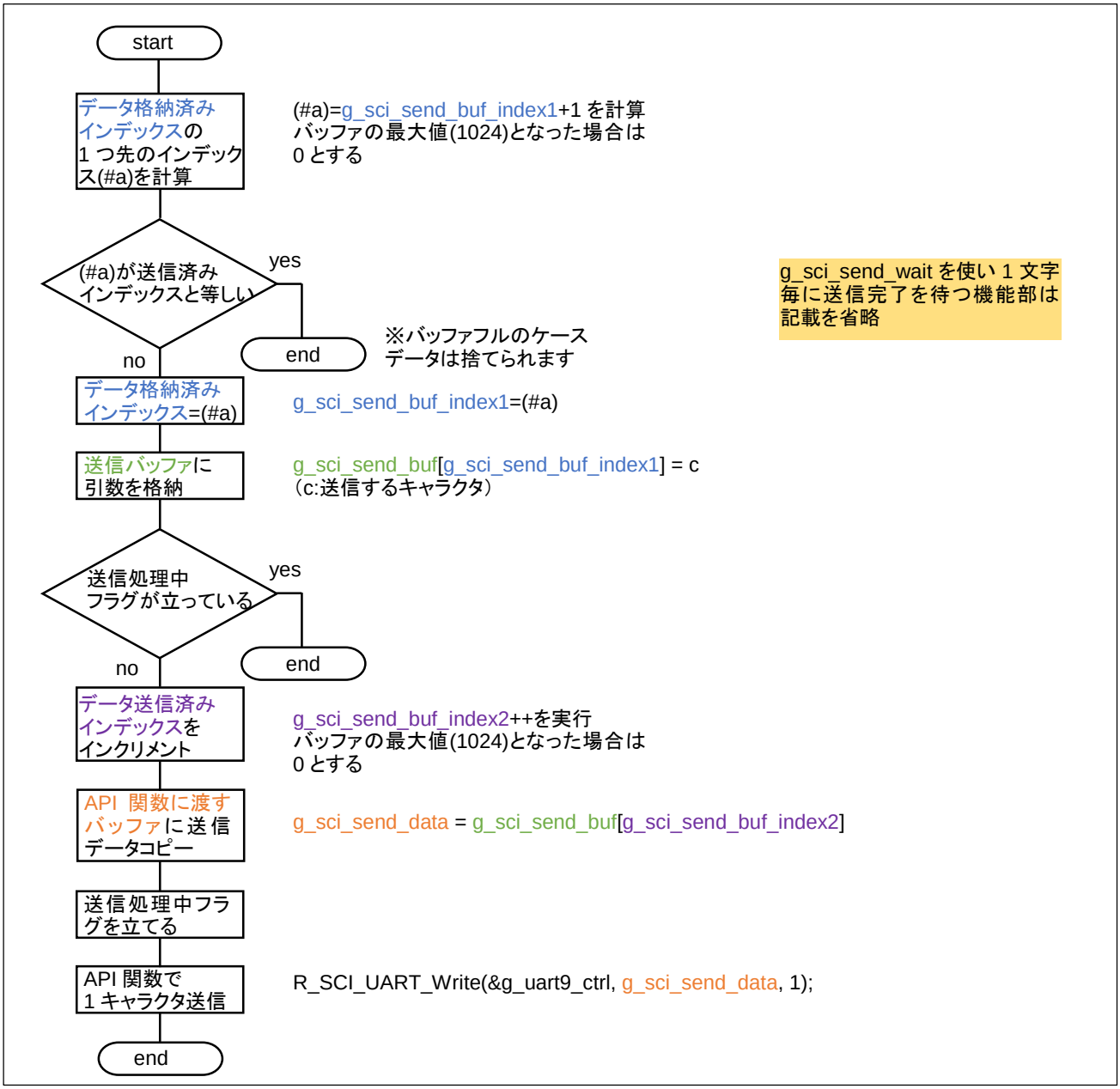
```
R_SCI_UART_Read(&g_uart9_ctrl, &g_recv_data, byte_count);
```

となっています。第 3 引数に受信バイト数を指定します。受信のケースでは、予めデータバイト数が判っている事は少ないと思います。この関数を使う限りでは、byte_count のデータを受信するまでは受信完了とならないので、本サンプルプログラムでは、API 関数を byte_count=1 で実行し、独自に設けたリングバッファに格納していく形としています。

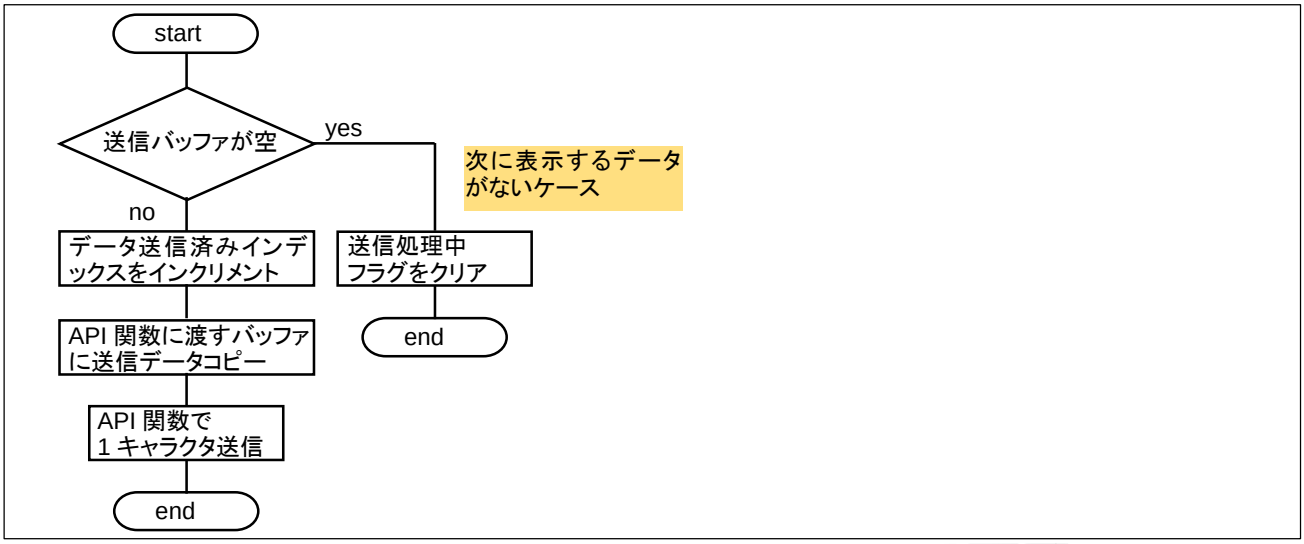
1.9. フローチャート

リングバッファの処理を含めたフローチャートを示します。

ー1 文字送信関数(sci_write_char)フローチャートー(¥sci¥sci.c)

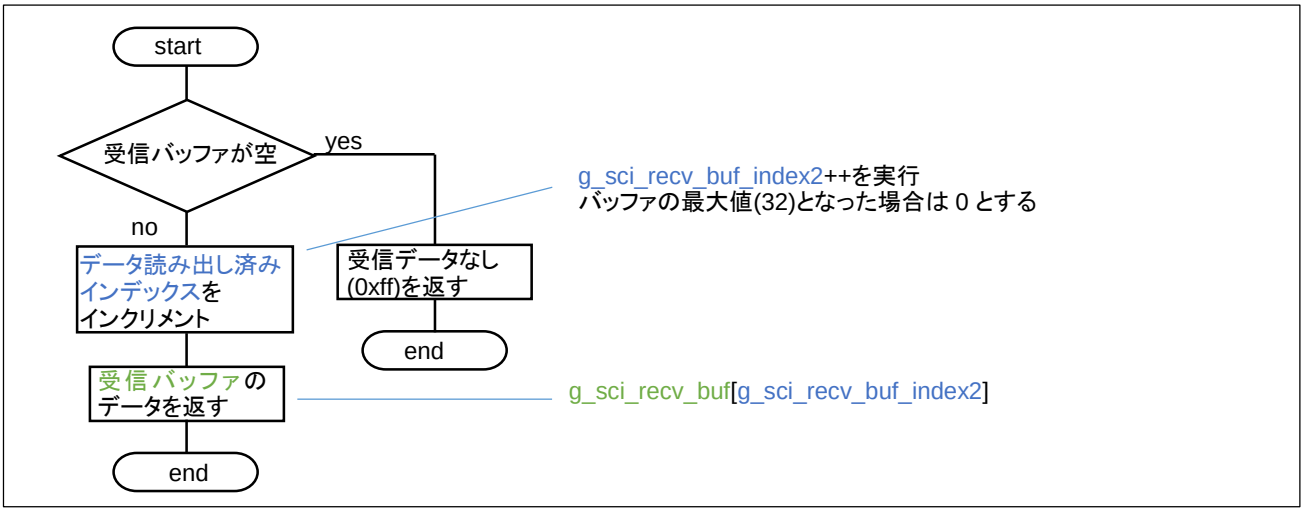


ーSCI9 割り込み関数 TX 送信完了処理部　フローチャートー(¥sci¥sci.c)

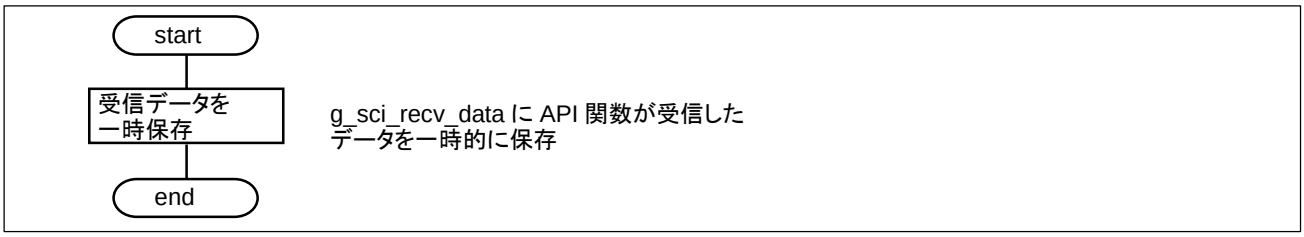


データの送信処理（API 関数を呼び出すところ）は、1 キャラクタのデータ送信完了時（TX 送信完了割り込み）に送信バッファにデータが残っている時は、「割り込み関数内で次のデータを送信」します。1 キャラクタのデータ送信完了時に送信バッファが空であれば、「次に sci_write_char()関数を呼び出した際」に、API 関数が呼び出されます。

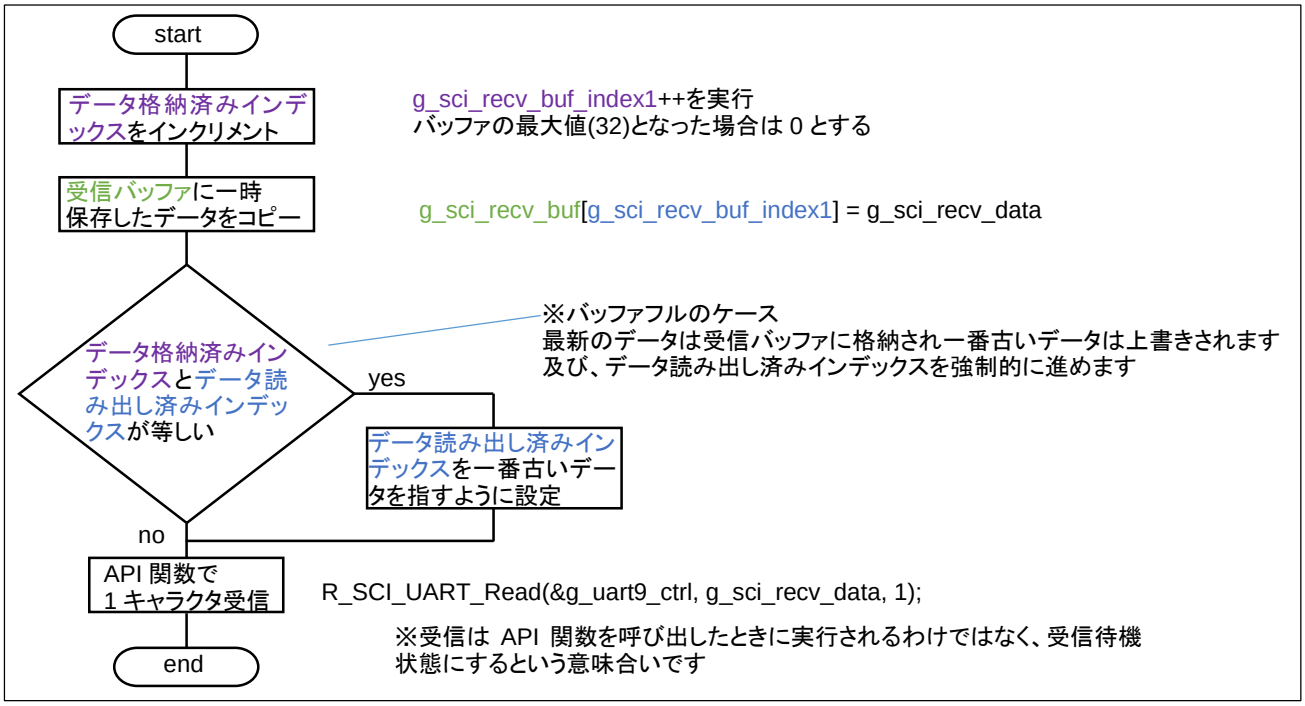
－1 文字受信関数(sci_read_char)フローチャート－(¥sci¥sci.c)



－SCI9 割り込み関数 RX 受信処理部　フローチャート－(¥sci¥sci.c)



－SCI9 割り込み関数 RX 受信完了処理部　フローチャート－(¥sci¥sci.c)



受信側は、サンプルプログラムでは、常に 1 キャラクタの受信を実行しているので、「SCI9 割り込み関数 RX 受信処理部」と「SCI9 割り込み関数 RX 受信完了処理部」は連続して呼び出される形となります。

1 キャラクタの受信が完了すると、次のデータ受信を指示します。RX 側は、PC のキーボードを叩いたタイミングでデータ受信となるので、マイコン側からすると、いつデータが来るかは判らないので、プログラムとしては、常にデータを待ち受けている状態としています。なお、sci の処理化処理(sci_init)内で最初の受信処理(R_SCI_UART_Read)を実行しています。

1.10.SCI の送受信処理の効率に関して

サンプルプログラムでは、ユーザにとって使い勝手が良い（であろうと考えて）、リングバッファ構成の関数を作成していますが、自作関数の中では API 関数を使っています。API 関数は、本来複数バイトのデータを処理できる仕様ですが、本サンプルプログラムでは、送信・受信とも 1 バイト単位での処理としています。API 関数は、複数バイトのデータ処理の方が効率が良い（CPU の処理時間を食わない）です。

送信バッファにデータを格納した際、ある程度まとめて API 関数に渡す等、効率を改善する余地はあると思いますので、効率が気になる方は、何か考えてみてください。なお、2 章では、効率に関しての実験結果を載せています。

－連続して大量のデータを出力する場合－

デバッグ用に SCI からデータを出力する場合で、プログラムの流れをできるだけ止めたくないという場合、本サンプルプログラムで採用しているデータのバッファリング（データ表示関数を実行したときにはバッファへのコピーのみ実行）は有効ですが、バッファのサイズは有限です。連続して出力処理を行った場合、本サンプルプログラムではバッファがあふれた際にはデータが捨てられます。

プログラムが一旦停止しても良いのでデータを出力したいという場合は、以下の方法があります。

(1)sci_write_flush 関数を使う

```
sci_write_str("...................");
sci_write_flush();    //SCI の送信バッファが空になるまで（データの出力が終わるまで、この行で）待つ
```

(2)g_sci_send_wait フラグを使う

```
g_sci_send_wait = FLAG_SET;
```

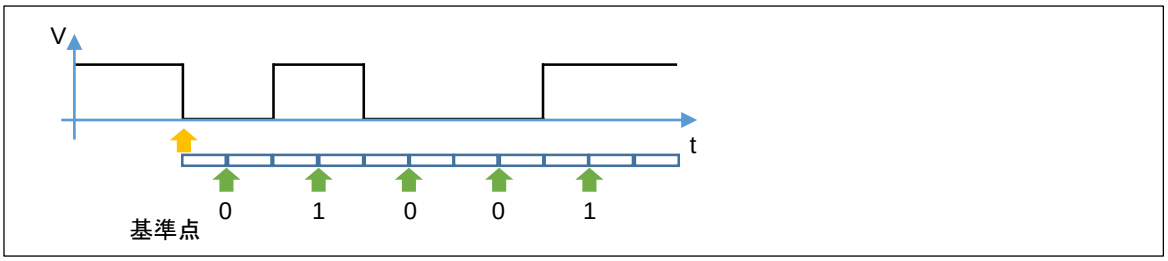
(2)は出力バッファが溢れない限りは、データ表示関数はバッファへのコピーのみを行い、出力バッファが溢れると、出力バッファに空きができるまで待ちます。出力バッファのサイズを超えるようなデータを連続で送信する場合に利用できる機能です。

コラム　クロック信号に関して

データ信号を送る場合、いくつかの方式が考えられます。

(1)信号のみ送信

SCI(UART)や、CAN もこのタイプです



データを受信する側では、基準となるエッジ（波形変化点）から、一定時間（初回は半サイクル、それ以降は 1 サイクル時間毎）データをサンプリング（L か H かを判断）する方式です。
※図では、波形変化点の真ん中に緑の矢印（サンプリングポイント）を描いていますが、CAN ですと波形変化点の真ん中（50%点）ではなく、真ん中より後ろの 75%程度の位置にサンプリングポイントを設定します

この方式の利点は、「信号線が 1 本でデータを送れるので効率が良い」という事となります。欠点としては、「データ送信のレートを送受信の双方で予め決めておく必要がある」「送信側と受信側の通信の基準クロック精度が要求される」「あまり長いデータパケットは送れない」という事でしょうか。

波形の変化点を決めているのは、送信側です。この変化点は、送信側の回路の基準クロックをベースにしています。RA2L1 で SCI を使う場合は、PCLKB が基準クロックとなります。PCLKB は、HOCO(48MHz)を 1/2 分周して生成した 24MHz です。HOCO には、±1%の誤差が含まれます。

SCI(UART)の場合、基準クロックの誤差に加え、設定速度 115,200bps(86.6us)の基準クロックの整数倍からのずれも、ビットレート誤差に加算されます。SCI（UART）の基準クロック PCLKB が 24MHz の場合、送信される信号は 24MHz をベースに分周されたタイミングでの波形生成になるので、ビットレートは 115,384.6bps となりこの時点で、0.16%の誤差を含みます。

RA2L1 で、115,200bps に設定した場合、最大 1.16%のビットレート設定誤差が生じる事となります。

受信側は、（送信側とは独立した）受信側のクロックで動作しているので、同様のビットレート誤差を持っています。

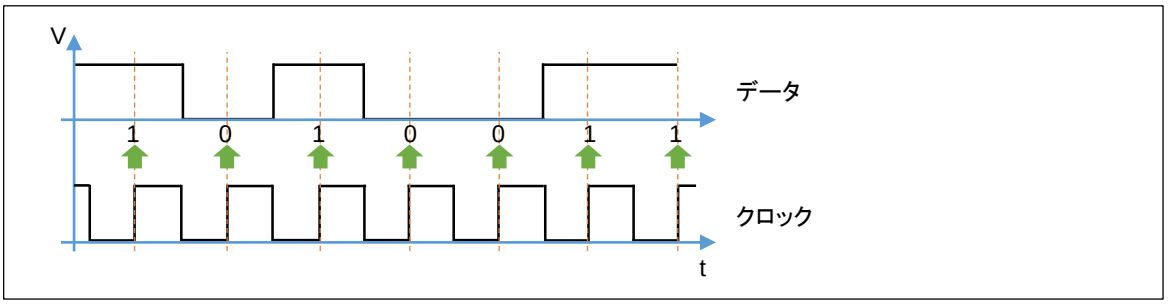
送信側と受信側の誤差の合算値が最悪のケースとなります（送信側と受信側で同じマイコンであれば、ビットレート設定に伴う 0.16%の誤差は相殺される等ありますが）。一般的には、ビットレート誤差は、3%以内が目安とも言われています。（このケースでは、1.16%なので、合格の範囲内でしょうか）

コラム　クロック信号に関して

また、信号のみ送信する系は、データパケット長が長いことも問題となります。最初の変化点（前の図では黄色矢印）の点から、受信側の基準クロックで、サンプリングポイント（緑の矢印）を刻んでゆくので、後ろ（時間が経過）に行く毎にずれ（送信側と受信側の誤差）が累積していきます。SCI(UART)では、1 パケットの長さは 10bit です（ストップビット 2、パリティあり等では 10bit より多少長くなる）。1 キャラクタ毎に、スタートビットがあり、受信側はスタートビットのデータをタイミングの起点に定める事ができますので、基準点に対して 10 回＋α のデータをサンプリングするだけですので、それほど困難ではないかと思います。

(2)クロック信号も送る

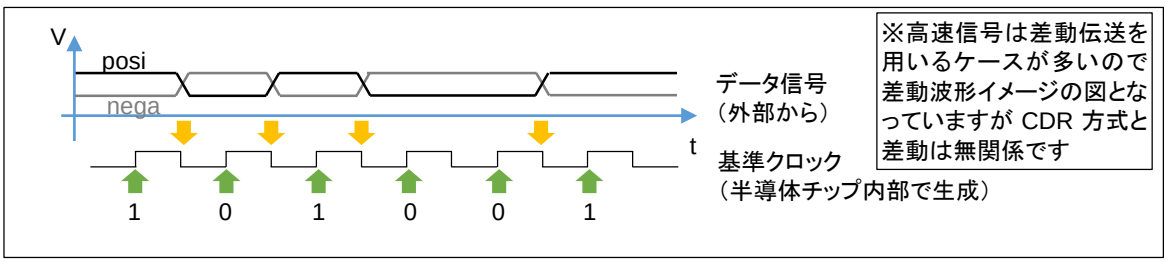
SPI や I2C（のマスタ側）や、メモリー（DDR2/3/4 等）など、チュートリアル 2 の LCD もこのタイプです



送信側がクロックとデータを両方送る場合、ビットレート設定誤差や基準クロックの誤差は、キャンセルされます。タイミングの絶対値がずれたとしても、データとクロックの相対的なタイミングには影響しないためです。受信側では、データ信号とクロック信号の 2 者のタイミング（関係）が重要です。この方式ですと、1 パケットのデータサイズの制約は付きません。

(3)データのみ送るが…

クロック信号をデータ信号と共に送る方式は、速度がそれ程速くない場合は問題ありませんが、非常に高速になってくると（数 Gbps~）クロックとデータの基板上の配線長の差分や半導体チップ内の回路のばらつき等で、クロックとデータのタイミングずれが問題になってきます。



そのような場合、データ信号からクロック信号を抽出する（クロックデータリカバリ，CDR）という手法が用いられます。受信側の半導体チップ内でクロック信号を生成し、そのクロック信号は、入力データの変化点を基準に校正を掛けて（クロックを速く、または遅くする方向に補正して）データ信号と同期が取れるようにする方式です。

コラム　クロック信号に関して

クロックデータリカバリを使う方式ですと、データの変化点がないと、受信側の基準クロックを送信側のビットレートに同期できませんので、0 データが長く続くと、受信側は何ビット 0 が来たかを判断するのが難しくなります。（そういう意味では、最初のデータだけを送る方式と同じ欠点があります。）

クロックデータリカバリを使う、PCI-Express では、Gen1(2.5Gbps),Gen2(5Gbps)では、8B-10B 変換（8bit のデータを 10bit に伸長する）により、5bit 以上同一符号（0 または 1）が続かない事を保証していました。Gen3(8Gbps)以降は、128B-130B 変換で最悪 128bit 同一符号となるケースがあり得ます。

CAN のデータフォーマットでビットスタッフィング（5bit 同一符号が続くと、ビット反転を挿入する）を行っているのも、同じ理由です。（データ信号に変化点が入る事で、タイミングの同期を取っている）。

※CAN は、(1)の信号しか送らないグループとして前述しましたが、(3)という訳でもなく、時々基準点をリセットする(1')でしょうか

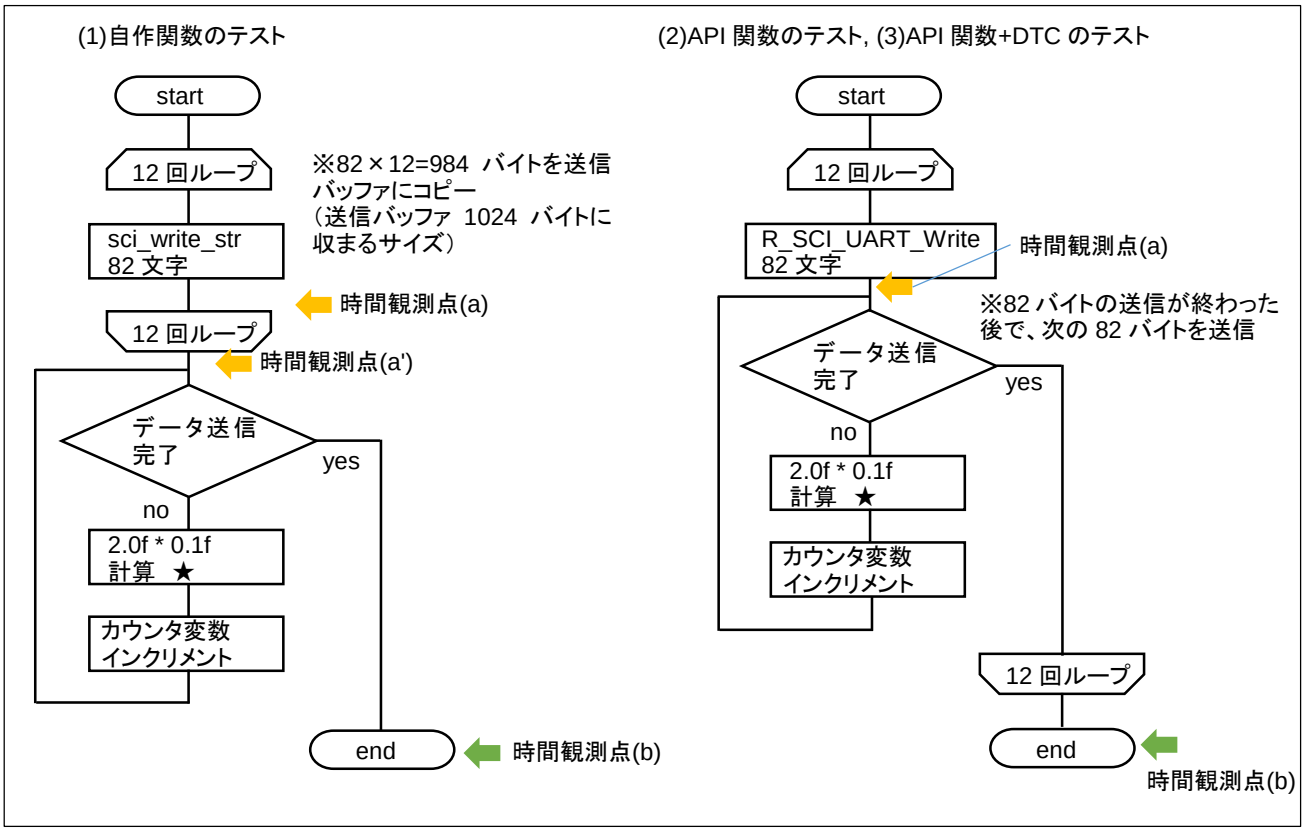
2. API 関数と DTC を使った場合の効率に関して[参考]

CPU の効率に関して単純に比較を行ってみました。

(1)自作関数(sci_write_str)
(2)API 関数
(3)API 関数+DTC 有効化

の 3 種類です。

2.1. テストプログラムフローチャート



テストでは、82 バイト(80+CR+LF)を 12 回送り（合計 984 バイト）のデータ出力が完了するまでの時間を計測しています。

自作関数では、984 バイトは、送信バッファの容量（1024 バイト）内なので、82 バイトを送信バッファにコピーする処理を 12 回連続で実行し、その後、★の処理（浮動小数点の乗算）をデータ送信が完了するまで実行します。

API 関数では、82 バイトの送信後、★の処理を実行しつつ、1 セット（82 バイト）の送信完了まで待ち、トータルで 12 セット行います。API 関数では、DTC を使わない場合と DTC を使った場合も比較しています。

見るべきポイントは、全体の処理完了までの時間と、SCI でのデータ（文字）出力中に、他の処理★（ここでは演算）にまわす余力がどの程度あるのかです。

時間の観測は、マイコンのタイマ機能を用い、100us 刻みで行っています。

2.2. 実行結果

○実行時間　[コンパイラの最適化オフ(-O0)]

		(1)	(2)	(3)
(a)	82 文字のデータを関数(*1)に与えて処理が終わるまでの時間	300us	0(*2)	0(*2)
(a')	(a)×12 回（984 文字）を出力バッファにコピーするのに要する時間	4.9ms		
(c)	984 文字の表示処理が終了するまでの時間	102.2ms	85.4ms	85.6ms

(*1)自作関数では、sci_write_str(), API 関数では R_SCI_UART_WRITE()
(*2)時間計測の分解能が 100us なので、100us 未満

(a)の時間は、自作関数では 1 文字ずつ出力バッファにコピーする処理となっているので、結構時間が掛かっています。sci_write_str は内部では sci_write_char を使って 1 文字ずつ処理しています。
※sci_write_str から、sci_write_char を呼び出すことなく、複数バイトまとめて直に出力バッファにコピーする等、改善の余地があると思います

(c)の時間ですが、1 文字の表示に 86.8us 掛かるので、984 文字で、85.4ms となります。(2)(3)は、データ出力を連続で行っていますので、ほぼ理論値が出ています。それに対し、(1)は出力バッファから文字表示するところでも、1 文字単位での処理となっているので、20%程度速度が落ちています。この部分でも、改善の余地があると思われます。

次に、文字出力中にどの程度他の処理が行えるかを見てみたいと思います。

○計算処理余力　[コンパイラの最適化オフ(-O0)]

	(1)	(2)	(3)
全体の処理が終わるまでに実行された★の演算回数	15150	16928	18159
1ms あたりの演算回数	148.5	198.2	212.1

文字出力中に、浮動小数点数の演算（2.0×0.1）が何回実行できたかを拾ったのが上表となります。この数値が大きい方が、文字出力の負荷が小さくマイコンが他の処理に時間を掛けられるという事です。実行時間だけを見ると、(2)と(3)では、DTC を使わない(2)の方が微妙に速いという結果でしたが、この結果を見ると、DTC は有効に働いている様に見えます。

〇実行時間　[コンパイラの最適化デフォルト(-O2)]

		(1)	(2)	(3)
(a)	82 文字のデータを関数に与えて 処理が終わるまでの時間	100us	0	0
(a')	(a)×12 回(984 文字)を出力バッファに コピーするのに要する時間	2.3ms		
(c)	984 文字の表示処理が終了するまでの時間	93.7ms	85.4ms	85.5ms

〇計算処理余力　[コンパイラの最適化デフォルト(-O2)]

	(1)	(2)	(3)
全体の処理が終わるまでに実行された★の 演算回数	18848	19236	19828
1ms あたりの演算回数	201.2	225.2	231.9

コンパイラの最適化を有効化(デフォルト)とした場合は、上記となります。全般的に速度と計算処理余力は向上していますが、傾向は変わりません。

この様にベンチマークを取ってみると、改善点が見えてきたかと思います。

API 関数は、効率良く作られていると思うのですが、連続して呼び出せないのが非常に不便です。

自作関数は、
・送信バッファ(リングバッファ)にコピーする処理
・文字出力する処理
の 2 点で 1 文字ずつ処理している点に関して、改善の余地があると思われます。

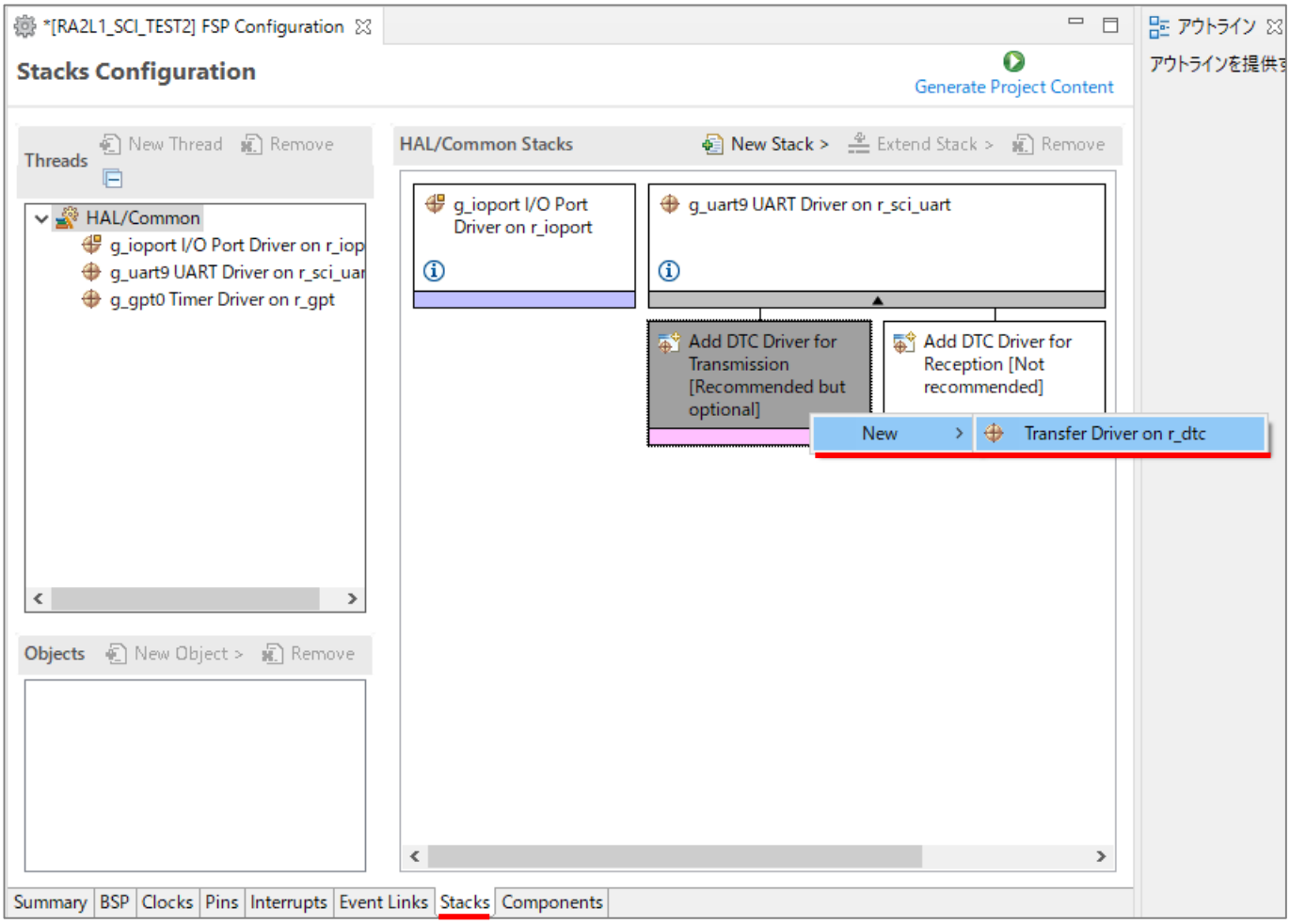
自作関数も、最後に文字出力する部分は API 関数を使っています。現状、API 関数を使いつつ、高速化、高効率化する余地があると思います。更なる性能向上を求めるのであれば、API 関数を使わず直接マイコンのハードウェア(レジスタ)を制御するという事となると思います。

(8bit マイコンの時代には、レジスタを直接制御する手法が当たり前でしたが、32bit マイコンではマイコン自体の演算性能が昔に比べて極端に上がっています。自作関数を作る際、性能最優先ではなく、移植性が確保できる API 関数を使うという選択肢もあると思います。)

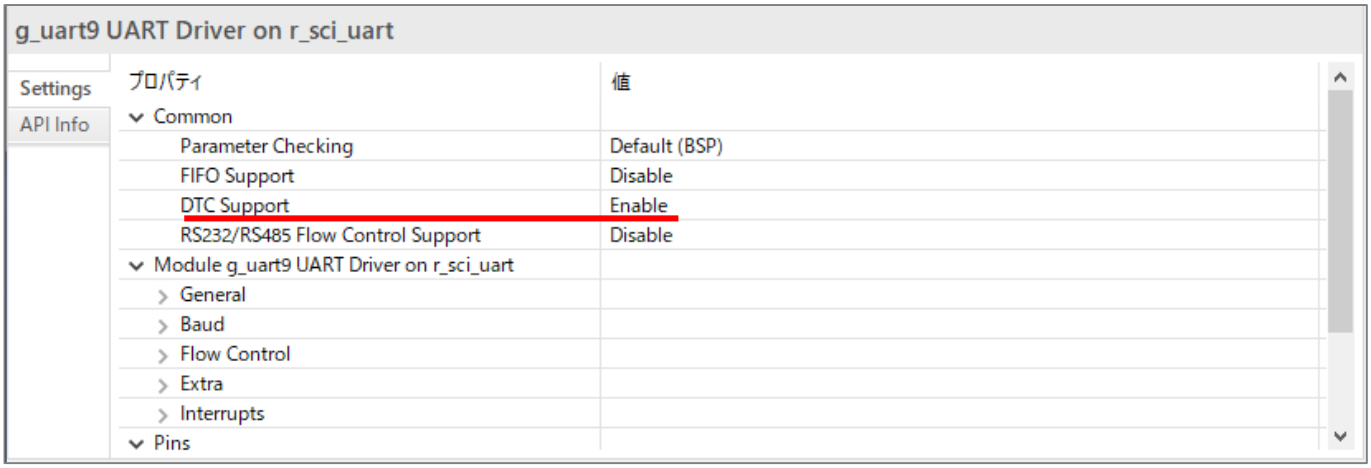
このチュートリアルでは、次のステップである速度的なチューニングまでは行いませんが、自作の関数やルーチンを作成した際は、速度やメモリ使用量等のベンチマークを行う事で、弱点や改善点が見えてくるものと思います。

2.3. DTC の有効化設定

SCI(UART)と DTC を連動して使用する場合は、下記の設定が必要です。



FSP Configutation.xml の Stacks タブの r_sci_uart の下の
ADD DTC Driver for Transmission
の箱を「左クリック」 ※右クリックではありません
して、
New – Transfer Driver on r_dtc
を追加。



DTC Support を「Enable」に変更。

※FSP の設定を変更した際は、「Generate Project Content」ボタンを押して、ソースファイルに反映させることを忘れないでください

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	2021.5.18	—	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:http://www.hokutodenshi.co.jp

ルネサス エレクトロニクス RA マイコン搭載
HSB シリーズマイコンボード 評価キット

SmartRA 学習キット
チュートリアル 5

株式会社 北斗電子

 2021 年 5 月 18 日改訂 REV.1.0.0.0 (210518)