RL78 の UART への受信リングバッファ機能追加

(1) 変更内容の概要

今回は，以前ちらっと書いたように CS+のコード生成機能で生成されたコードに受信用の 16 バイトのリングバッファを追加することにします。INTSR0 割り込み処理の生成されたコードの中で修正を加えても問題ないところは r_uart0_callback_softwareoverrun 関数のユーザ・コード部分になります。その場所を以下に示します。

```
static void r_uart0_callback_softwareoverrun(uint16_t rx_data)
{
    /* Start user code. Do not edit comment generated here */
    /* End user code. Do not edit comment generated here */
}
```

ここにプログラムを追加する

この関数は，R_UART0_Receive 関数呼び出しで，受信バッファが設定される前に受信完了割り込みが発生したときの処理になります。本当は割り込み処理本体部に記述するのが望ましいのですが，そうするとコード生成を行うと消されてしまいます。余分なコードが残りますが，機能的にはここに記述しても問題なさそうです。

リングバッファ機能はコード生成のオリジナルの R_UART0_Receive 関数とは排他的に使用する必要があります。そのため，リングバッファを使う場合には，絶対に R_UART0_Receive 関数は使用しないようにしてください。

リングバッファの状態確認用と，リングバッファからのデータ読み出し用の関数は別途用意します。

(2) リングバッファの概要

ここで追加するリングバッファは 16 バイトのサイズのものです。ここで，データを制御するために，3 つの変数を使用します。これらを外部変数として，Global variables and functions 部のユーザ・コード部分に以下のように宣言しておきます。

```
/* Start user code for global. Do not edit comment generated here */
extern volatile uint8_t g_tx_ready_flag;                /* 送信完了／許可フラグ      */
extern volatile uint8_t g_rx_buff[16];                  /* 受信用リングバッファ      */
extern volatile uint8_t g_rx_rdpt;                      /* データの読み出しポインタ  */
extern volatile uint8_t g_rx_dtno;                      /* 格納されているデータ数    */
extern volatile uint8_t g_rx_status;                    /* 受信ステータスフラグ      */
/* End user code. Do not edit comment generated here */
```

リングバッファの制御にはリングバッファからデータを読み出すためのポインタ（リードポインタ）と格納されているデータ数の 2 つの変数を使用します。この組み合わせのときには，受信データの格納ポインタはリードポインタ＋データ数となります。

(3) リングバッファへの格納処理

リングバッファへの受信データの格納処理を以下に示します。

```
    /* 受信用リングバッファ制御 */

    uint8_t  setptr;

    if ( g_rx_dtno < 16 )
    {                               /* バッファに空きあり                    */
        setptr = ((g_rx_rdpt + g_rx_dtno) & 0x0F);  /* 書き込みポインタ      */
        g_rx_buff[setptr] = rx_data;    /* 受信データをバッファに格納        */
        g_rx_dtno++;
    }
    else
    {                               /* バッファがフル時の処理                */
        g_rx_status = 0x80;             /* オーバフロー・フラグをセット      */
    }
```

最初にデータ数を確認し，バッファがいっぱいになっているかをチェックします。バッファに余裕があれば，書き込み用のポインタを求めます。バッファを 16 バイトにしたのは，ポインタを求めるときの余分な判断処理をなくし，簡単な演算だけで済ませるためです。求めたポインタを使って，バッファに受信データを格納し，データ数を＋1 します。

既に，バッファがいっぱいなら，バッファのオーバフロー・フラグをセットします。

(4) リングバッファの管理処理

リングバッファを管理するために，UART0.c でいくつかの関数を定義しています。

関数名	機能概要
init_bf	リングバッファを空状態にする
chk_status	リングバッファの状態を読み出す。
get_data	リングバッファから 1 データを読み出す。
get_blk	リングバッファから指定した数のデータを読み出す。

(a) init_bf 関数

INTSR0 割り込みをマスクして，ポインタ（g_rx_rdpt），カウンタ（g_rx_dtno），ステータス（g_rx_status）を 0x00 にクリアします。最後に INTSR0 割り込みのマスクを解除して終了します。

(b) chk_status 関数

リングバッファの状態（ビット 7 にオーバフロー，ビット 4～0 にデータ数）を戻します。実行すると，ステータスをクリアします。戻り値が 0x00 ならリングバッファは空ということになります。

(c) get_data 関数

リングバッファから 1 バイトのデータを読み出します。リングバッファが空だった場合には 0x00 が，データがあれば，一番古いデータが戻されます。

通常は，chk_status 関数でデータの有無を確認して，データがあれば get_data 関数で読み出すことになります。

(d) get_blk 関数

リングバッファから複数のデータを読み出します。読み出したデータは第 1 引数で指定したバッファに転送されます。第 2 引数のデータ数とリングバッファのデータ数の少ない方のデータ数だけでのデータが転送されます。戻り値としては，指定されたデータ数のうちで転送できなかった数となり，指定したデータ数が全て転送されると 0x00 となります。

chk_status 関数の戻り値を第 2 引数に設定すると，リングバッファのすべてのデータを読み出すことができます。

(5) リングバッファの管理処理関数の使い方例

簡単な評価のための例を main 関数に示します。この例は，UART0 の TXD0 端子と RXD0 端子を接続した状態で動作させます。

(a) 初期化部

main 関数が最初に呼び出す R_MAIN_UserInit 関数は，以下のようになります。

```
void R_MAIN_UserInit(void)
{
    /* Start user code. Do not edit comment generated here */
    R_UART0_Start();            /* UART0 動作起動             */
    init_bf();                  /* リングバッファを初期化    */
    g_tx_ready_flag = 0x01;     /* 送信完了フラグをセット    */
    EI();
    /* End user code. Do not edit comment generated here */
}
```

UART0 のハードウェアを動作可能に設定して，リングバッファの初期化を行っています。

(b) main 関数での準備

main 関数で通信を行うための変数は以下のように定義しています。

```
/***********************************************************************************
Global variables and functions
***********************************************************************************/
/* Start user code for global. Do not edit comment generated here */
volatile uint8_t g_rx_ready_flag;       /* 受信完了フラグ       */
volatile uint8_t static g_rx_buff2[16]; /* 受信データ・バッファ */
volatile uint8_t g_tx_ready_flag;       /* 送信可能フラグ       */
volatile uint8_t static g_tx_buff[16];  /* 送信データ・バッファ */
volatile uint8_t static buff_work[20];
```

このなかで，送信データ・バッファは定義しただけでまだ使っていません。送信データは以下のように const 宣言しています。

```
static uint8_t const test_data[2][16] =
{
    {0x40,0x41,0x42,0x43,0x44,0x45,0x46,0x47,
     0x48,0x49,0x4A,0x4B,0x4C,0x4D,0x4E,0x4F},
    {0x50,0x51,0x52,0x53,0x54,0x55,0x56,0x57,
     0x58,0x59,0x5A,0x5B,0x5C,0x5D,0x5E,0x5F}
};
```

また，ローカル変数として以下の宣言をしています。

```
    /* Start user code. Do not edit comment generated here */
    {

        uint8_t const *pt_txdata;
        uint8_t work;
```

(c) main 関数での送受信の使用方法

最初の const のテーブル test_data の先頭から 16 バイトを送信します。

g_tx_ready_flag フラグで送信が完了していることを確認してからフラグをクリアし，コード生成された R_UART0_Send 関数を使って送信します。これが送信の通常の方法です。こうすることで，送信中に他の処理を並行して実行できます。

今回は，リングバッファの状態を確認したかったので，この後に送信完了待ちを行い，chk_status 関数でリングバッファの状態をモニタしています。

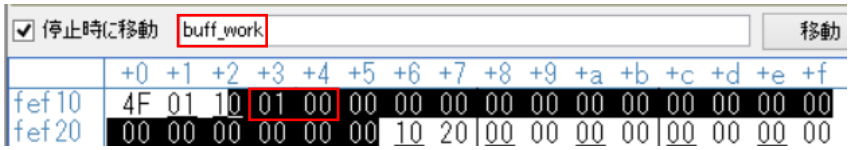
```
        while( 0x00 == g_tx_ready_flag )        /* 送信完了待ち              */
        {
            NOP();
        }
        g_tx_ready_flag = 0x00;                 /* 送信完了フラグをクリア    */
        pt_txdata = &test_data[0][0];           /* 送信ポインタを設定        */
        R_UART0_Send( (uint8_t *)pt_txdata, 0x10 );   /* データ送信処理開始 */
```

リングバッファに 16 バイトのデータが入ったことを確認して，リングバッファから get_blk 関数を用いて 17 バイトの読み出し要求を行います。当然，1 バイト不足するので，16 バイトのデータが転送され，戻り値には 0x01 が入ることになります。

```
        work = get_blk( (uint8_t *)g_rx_buff2, 0x11 ); /* リングバッファから読み出し*/
        buff_work[1] = work;

        NOP();
        work = chk_status();                    /* リングバッファ状態確認    */
        buff_work[2] = work;
```

ここまでを E1 で実行して，buff_work の内容を確認したのが次ページの図になります。



buff_work[1]が 01 は，転送できなかったデータが 1 バイトであることを示しています。buff_work[2]が 00 は，リングバッファのデータが空であることを示しています。

以下に示すのは，chk_status 関数を用いたリングバッファへのデータ受信待ちとそれに続く 1 バイトのデータ読み出しです。この while 文と get_data 関数の組み合わせが，リングバッファからの 1 バイトのデータ読み出しの基本的な処理の例です。

```
while ( 0x00 == chk_status() );
NOP();

work = get_data();
buff_work[6] = work;
NOP();
```

以下に示すのが，chk_status 関数を用いたリングバッファ内のデータ数を確認してそのデータを全て読み出す処理の例です。

```
work = chk_status();                    /* リングバッファ状態確認    */
buff_work[7] = work;
work = get_blk( (uint8_t *)g_rx_buff2, work ); /* リングバッファから読み出し*/
```

全体の処理はプロジェクトの中の r_main.c ファイルを参照してください。