Gertbot Advanced

Rev-1.1, 25-Jan.-2015

The Gertbot is a motor controller board for the Raspberry-Pi. For details how to operate it read the Gertbot manual and/or the Gertbot GUI. This manual describes how to unlock some of the more challenging features of the board.

Contents

1 Introduction 1
2 Cascade. 2
3 Reset 3
4 Use 3V3 power 3
5 More current for DCC and DC motors. 4
6 More current for Stepper motors. 6
7 Going to DC 10 Amp. 6
8 Going to Stepper 10 Amp. (Not) 7
9 Uploading new software. 7
10 Board Erase. 7
10.1 Erase 7
10.2 Debug and program 8
10.3 PCB information 8
10.4 Serial port handler 9
10.4.1 uart.h: 9
10.4.2 uart.c: 10

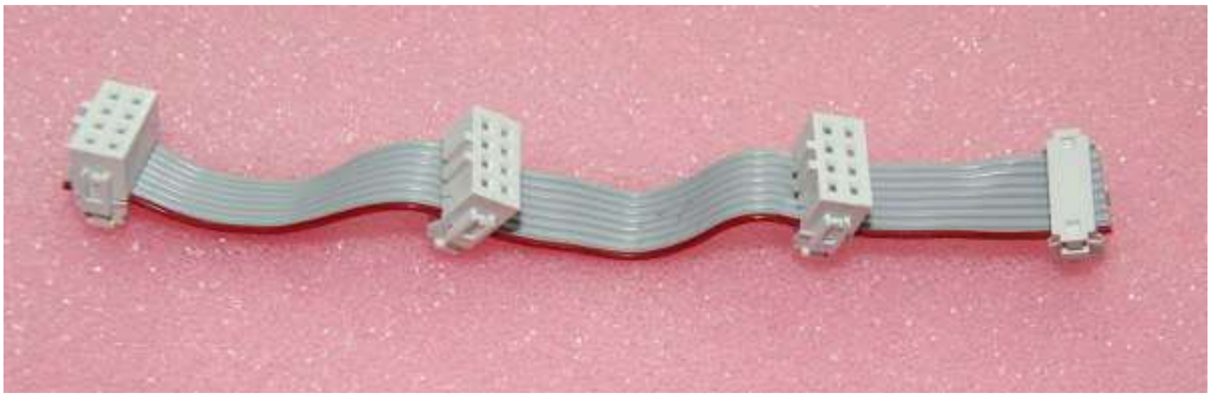
1 Introduction

This document describes the more complex and advanced options of the Gertbot.

For volume orders Fen Logic Ltd. can deliver bespoke solutions of the board for both hardware and software.
Contact Fen Logic Ltd for more information.

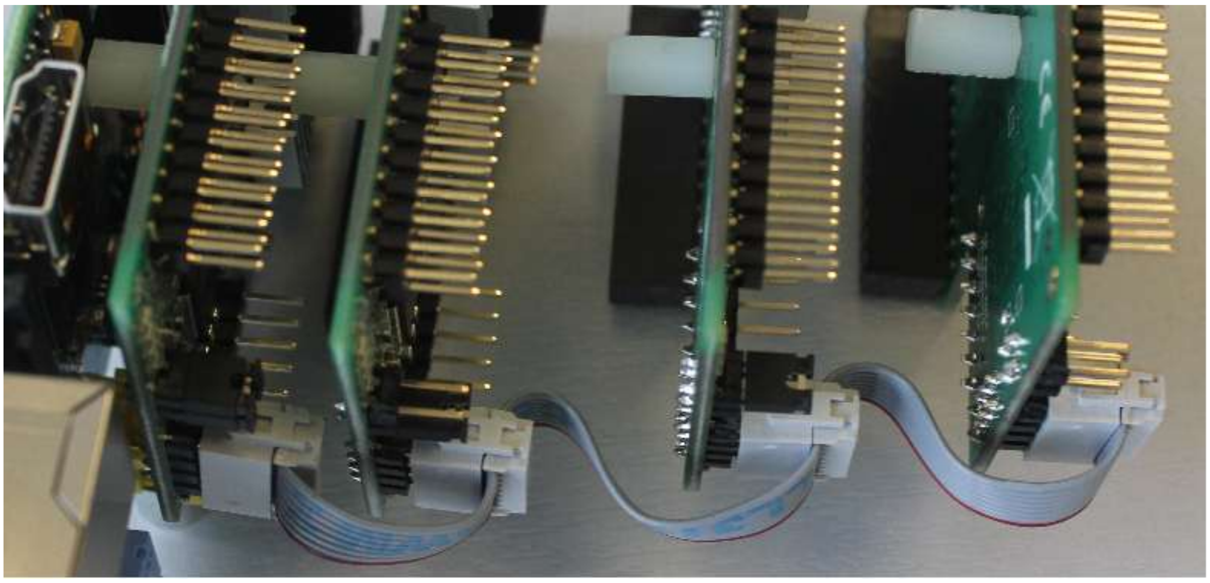
2 Cascade.

To use two or more boards you need to make a cascade cable. That is a flat cable with up to four 8-pin female headers. This is such a cable:



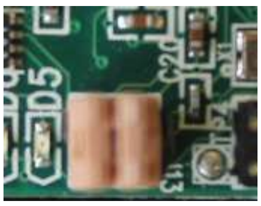
The length of the cable depends on how far apart you want to mount the boards. If you stack them using 15mm stand-offs you need 120 mm cable. You also need an 8-pin IDC press connector per board. I put a video on YouTube how to make a press connector a while ago: http://www.youtube.com/watch?v=sMiRoXY_oZg.

The following picture shows an example how the cable can be connected to four boards. Only one of these board (the most left one) is plugged into the Raspberry-Pi.

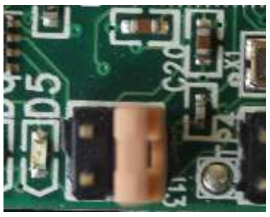


Note that the ID jumpers on the picture above are all different as you must give each board a unique ID.

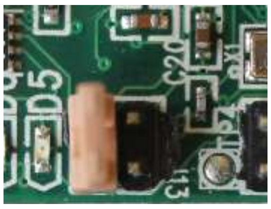
The following diagram shows the ID with the jumpers:



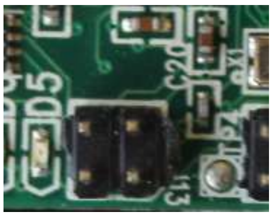
ID=0



ID=1



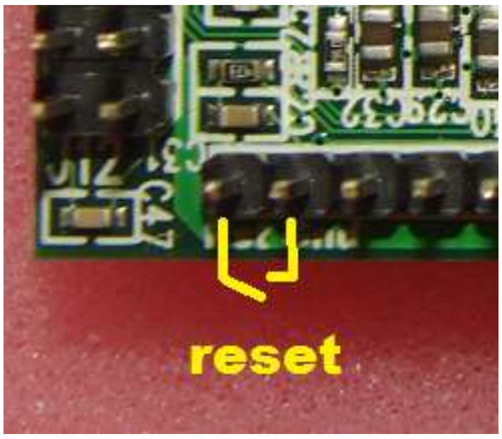
ID=2



ID=3

3 Reset.

In order to reset the Gertbot board you can shorten pins 6 and 7 of J10:



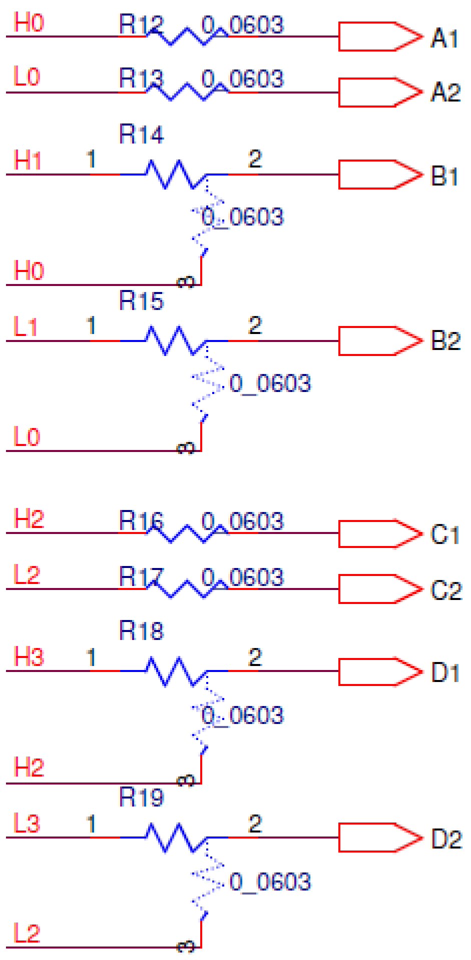
The reset is part of the cascaded connections. Thus if you cascaded the boards: if you reset one board you reset all of them.

4 Use 3V3 power

The on-board 3.3Volt regulator is specified for 300mA. The board itself uses about 65-70mA which leaves ~230 mA for the user. The drop-out voltage goes from ~100mV to ~200mV depending on the load, thus you need at least a 3.5V supply. (Yes, you can run the board from three 1.5V cells!)

5 More current for DCC and DC motors.

One H-bridge can pass 2.5A. As we have four of them it is possible to combine bridges to allow more then 2.5A. Here is the part of the ***schematics*** which deals with the signals we are going to mess about with:



Beware the A1, A2, .. D2 here are NOT the outputs of the bridge on the terminals. These are internal signals on the board. The picture above is taken from the Gertbot schematic

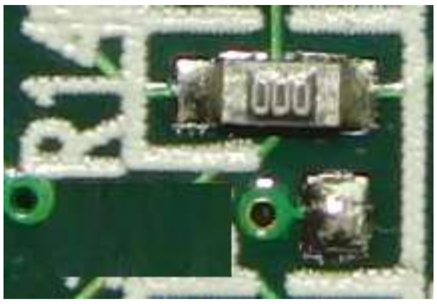
H0, L0, H1, L1, H2, L2, H3, L3 are signals from the controller. They are going to the H-bridges to control the outputs. Coming from the factory the following connections are present:

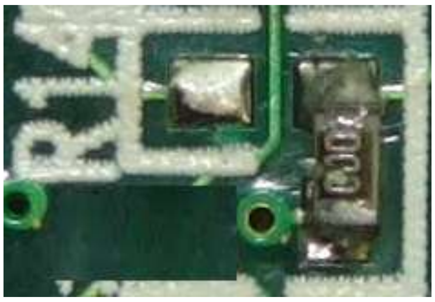
H0→A1	L0→A2	H1→B1	L1→B2
H2→C1	L2→C2	H3→D1	L3→D2

To make two bridges work in parallel for a ***DC motor*** we have to give them the same control signals. Thus to connect A and B in parallel we need the following connections:

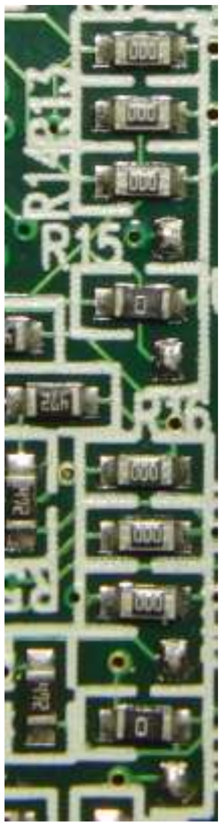
H0→A1	L0→A2
H0→B1	L0→B2

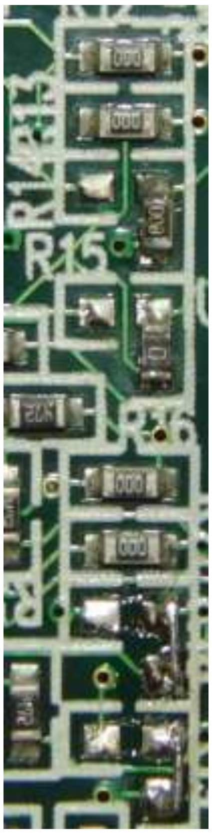
The board has a special layout to make this easy. In this case the board layout looks just like the schematic: Un-solder R14, rotate it 90 degrees and solder it down again:

Before: 

After 

The same is valid for R15, R18 and R19. You don’t have to re-use the “resistor”. You can use a piece of wire as well. The two images below show how I made all four connections. In the top two I re-used the resistors, in the bottom two I used wires.

Before: 

After 

This works for DC motors and only if you also parallel up the two outputs. Thus connect the A1 and B1 on the terminals together, you also have to connect A2 and B2 on the terminals together. (The same for C and D).

As H1 and L1 are not connected you can no longer use the motor 1 controls.
Again the same for H3 and L3 so the motor 3 can no longer be used.

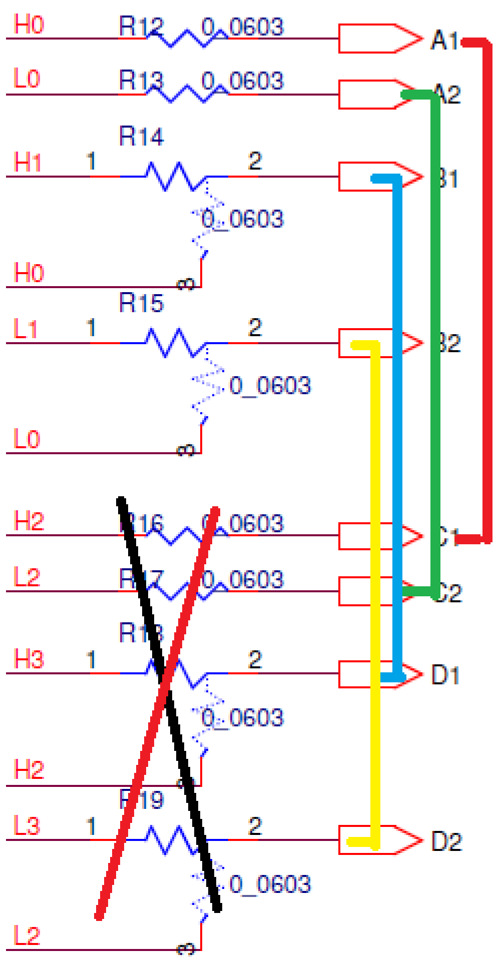
The current-limiting is per H-bridge so you do not need to make any changes to the current limiting logic. The current will be distributed about even between the two bridges so the current limiter will still kick in if one bridge starts to draw more than 2.5A. As that bridge will stop conducting the full current will try to go through the other bridge and its current detect will kick in as well.

For DCC you can connect the output to independent track segments. This has the advantage that if one track segment is shorted, the other(s) will keep generating a correct DCC sequence.

6 More current for Stepper motors.

I have NOT tried this so this chapter is based on theory. However the theory is simple and it should work.

A stepper motors is controlled with a set of signals: H0, L0, H1, L1. To double the current of a stepper motor you have to parallel up bridge A with C and bridge B with D. Remove R16, R17, R18 and R19. Then make connections like below.



You should set all error channels to 'stop board if high current detected'. Also you must parallel up the output on the terminals in the identical way: A1 with C1, A2 with C2 etc.

7 Going to DC 10 Amp.

Nope I have NOT tried this either.

Remove all of R14, R15, R16, R17, R18 and R19. Then Connect H0 (or A1, they are the same) to B1, C1 and D1. Connect L0 to B2, C2 and D2.

Again also connect the outputs in parallel in the right way: A1, B1, C1 and D1 together and A2, C2, B2 and D2 together.

I recommend you set all error channels to 'stop board if high current detected'.

8 Going to Stepper 10 Amp. (Not)

I would not recommend this as you may not have a good ground between the boards.

Therefore I will not describe how to do this. You have to be technical savvy to get this right. That again means you would not need a chapter like this to tell you how to do this.

If you want to try this don't forget to cross connect the high-current detection circuit too!!! (The "stop all boards if high current detected" may be a bit slow for this).

9 Uploading new software.

There is a GUI which can be used to upload new software: gb_upload. You cannot upload your own software as the code to be uploaded is encrypted and need to pass strict CRC checks before the board will accept it. To upload new code you must have only a single board connected. The procedure is straight forward: Start the gb_upload program, press the 'connect' button. The program will check if there is a single board connected. If so you can browse to a new image and press the 'upload' button.

I have not yet seen this go wrong yet, but beware that there is no back-up image on the board. If this goes wrong your board will be dead. There is a re-flash service but you will have to pay for that.

You also need to remove power and apply it again if you abort the upload procedure.

10 Run you own software: Board Erase.

The controller on the board has the protection bits set. This means you can NOT connect a debugger and see what is running. Neither can you upload your own code. If you want to program the Gertbot with your own code you are welcome to do so but you must first perform a 'Board Erase'.

A Board Erase will wipe all the code from the board.

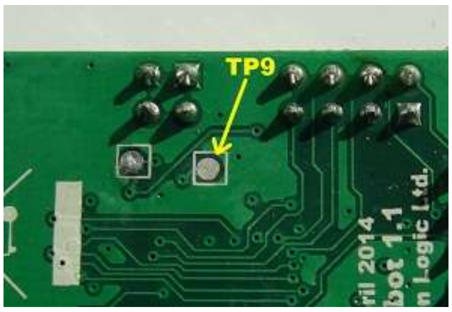
Afterwards you will need an Atmel JTAG debugger to re-program the device.

10.1 Erase

This is how you can erase the board:

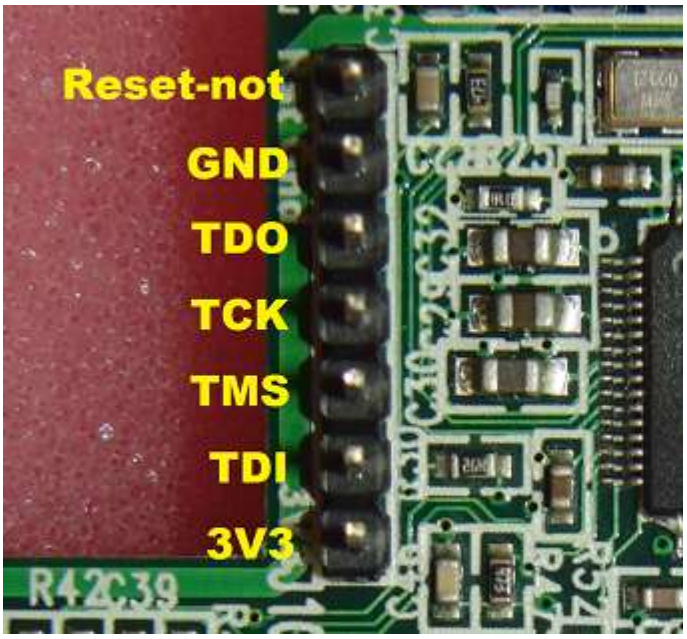
1. Make sure the board has no power.
2. Connect TP9 to a 3V3 pin. For example pin 1 of J10 or pin 17 of J3.
3. Apply 5V to the power supply.
4. Remove the power supply.
5. Remove the TP9 connection.
6. Apply 5V to the power supply.

The board is now erased. You can see this from the fact that the four LED's are slightly glowing, but none of them is blinking anymore.



10.2 Debug and program

J10 is the JTAG debug connector with the following pins:



There are several Atmel JTAG interface boxes out there. I have used the board with an Atmel SAM-ICE (Big Blue box) and an Atmel-ICE (Small White box). Beware that all those have different connectors so be careful when connecting up. If you have an Atmel-ICE (Small White box).the connections are as follows:

J10	ATM SAM ice cable
1	1 (red wire)
2	8
3	2
4	4
5	6
6	3,5,9
7	NC

10.3 PCB information

In order to use the board you have to know where the processor signals go to. I can't reveal the full board schematic as there are too many unscrupulous people who would copy the design. At the same time I realise that the board is of little use if the connection are unknown. The table below is a compromise between these two situations.

Below is a table which shows where the SAM signals are connected to:

Pin	Type	Drives	Goes to
9	PA17	LED0	
10	PA18	LED1	
11	PA21	LED2	
48	PA0	LED3	
3	PB0	ADC0	J3.14
4	PB1	ADC1	J3.13

Pin	Type	Drives	Goes to
5	PB2	ADC2	J3.16
6	PB3	ADC3	J3.115
23	PA24	Extra0	J3.2
25	PA25	Extra1	J3.1
26	PA26	Extra2	J3.4
37	PA27	Extra3	J3.3
38	PA28	Extra4	J3.6
41	PA29	Extra5	J3.5
42	PA30	Extra6	J3.8
52	PA31	Extra7	J3.7
35	PA5	UART Tx	
34	PA6	UART Rx	

10.4 Serial port handler

Getting the serial port to work is rather difficult. Not only do you have to cope with a new processor but in the documentation of the SAM devices it is difficult to find out which UART belongs to which set of I/O pins. It took me a long time. To save you that: the code below is an example UART driver which will work on the Gertbot, using the UART pins as described in the manuals.

10.4.1 uart.h:

```
//
//
// Gertbot Serial port handler
//
// G.J. van Loo
// Standalone version, 5 Dec. 2014
//
//
// Serial port works at 57600
// 1 start, 1 stop, 8 data, no parity = 10 bits/byte
// 57Kbits/sec

//
// Setup uart 0 57k6 n,8,1
//
void uart0_setup();

//
// Check uart0 receive queue has data
//
int uart0_available();

//
// Read byte from uart0 receive queue
// Returns byte or -1 if nothing available
//
int uart0_getch();

//
// Check a byte can be send
//
int uart0_space();

//
// Send byte to uart-0 transmit queue
// stall if full
int uart0_putch(uint8_t value);
```

10.4.2 uart.c:

```
//
// Gertbot Serial port handler
// Standalone version
//
// G.J. van Loo
// Initial version, 21 December 2013
// Copyright (c) 2015 Fen Logic Ltd.
// Usage is free for owners of a gert-bot.
//
// Serial port works at 57600
// 1 start, 1 stop, 8 data, no parity = 10 bits/byte
// 57Kbits/sec

#include <stdint.h>
#include "uart0.h"
#include "sam.h"

// input & output queue size
// Must be power of two!!
#define SER_INQ_LEN  128
#define SER_OUTQ_LEN 64

static uint8_t  ser_inq[SER_INQ_LEN];
static uint8_t  serin_getpnt;
static volatile uint8_t serin_putpnt;
uint32_t serin0_overflow; // Not static!

static uint8_t ser_outq[SER_OUTQ_LEN];
static uint8_t serout_putpnt;
static volatile uint8_t serout_getpnt;

//
// Setup UART0
//
void uart0_setup()
{
  //
  // Setup the hardware
  //

       // Enable clock to UART
  PMC->PMC_PCER0 = 1 << ID_USART0;

  // MAP UART0 on PIO pins
  // This requires function A on pins PA9 and PA10
  // Function select bits have same position in two different registers
  // {PIO_ABCDSR2,PIO_ABCDSR1} 00=A, 01=B, 10=C, 11=D
  PIOA->PIO_ABCDSR[0] &= ~(PIO_PA5A_RXD0|PIO_PA6A_TXD0);
  PIOA->PIO_ABCDSR[1] &= ~(PIO_PA5A_RXD0|PIO_PA6A_TXD0);
  // Disable the PIO from controlling these pins
  PIOA->PIO_PDR = (PIO_PA5A_RXD0|PIO_PA6A_TXD0);

  // Baudrate
  // Over = 1: 64M/(8*CD)
  // Over = 0: 64M/(16*CD)
  // 64MHz
  // Over   115K2   57K6
  //   0    34.72   69.44
  //   1    69.44  138.88
  USART0->US_BRGR = 70;
  // Control register : enable TX & RX
  USART0->US_CR = (US_CR_RXEN | US_CR_TXEN);
  /* Disable PDC channel whatever that may be */
  USART0->US_PTCR = UART_PTCR_RXTDIS | UART_PTCR_TXTDIS;
  // Mode register
  USART0->US_MR = US_MR_USART_MODE_NORMAL // normal mode
                | US_MR_USCLKS_MCK        // master clock
                | US_MR_CHRL_8_BIT        // 8 bits
                | US_MR_PAR_NO            // No parity
                | US_MR_NBSTOP_1_BIT      // 1 stop bit
//                | US_MR_OVER              // Undersampling bit
                ;
  // Setup interrupt handler
  NVIC_EnableIRQ(USART0_IRQn);
  // Receive interrupts enable
  USART0->US_IER = US_IER_RXRDY;

  //
  // Setup the software part of the UART
  //
  serin_putpnt  = serin_getpnt = 0; //  Set up the circular receive buffer
  serin0_overflow = 0;
  serout_putpnt = serout_getpnt = 0; //  Set up the circular transmit buffer

} // uart0_setup

//
// Put byte in serial receive queue
// Drop on input overflow and increment overflow counter
// Used only internally
//
void ser0_put(uint8_t value)
{ uint8_t next_put;
  next_put = (serin_putpnt + 1)& (SER_INQ_LEN-1);
  if (next_put==serin_getpnt)
  { serin0_overflow++;
    return;
  }
  ser_inq[serin_putpnt] = value;
  serin_putpnt = next_put;
} // ser0_put

//
// Check a byte is available
//
int uart0_available()
{
  return (serin_getpnt != serin_putpnt);
}

//
// Get byte from serial receive queue
// If empty return -1
//
int uart0_getch()
{ uint8_t value;
  if (serin_getpnt==serin_putpnt)
    return -1;
  value = ser_inq[serin_getpnt];
  serin_getpnt = (serin_getpnt + 1) & (SER_INQ_LEN-1);
  return value;
} // uart0_getch

//
// Check a byte can be send
//
```

```
int uart0_space()
{
  return ( ((serout_putpnt + 1 ) & (SER_OUTQ_LEN-1)) != serin_getpnt);
}

//
// Send byte to usart-0 transmit queue
// block if full
//
int uart0_putch(uint8_t value)
{ uint8_t next_put;
  next_put = (serout_putpnt + 1 ) & (SER_OUTQ_LEN-1);
  while (next_put==serout_getpnt)
    /* wait */ ;
  ser_outq[serout_putpnt] = value;
  serout_putpnt = next_put;
  // Enable TX interrupts
  // if they are already enabled that does not care but if not
  // the interrupt routine will pick up the data
  // we just put in the buffer**
  USART0->US_IER = US_IER_TXRDY;
  return 1;
} // uart0_putch

//
// Send null terminated string
//
void uart0_puts(char *s)
{ while (*s)
    uart0_putch(*s++);
} // usart0_puts

//
// USART interrupt routine
//
void USART0_Handler(void)
{ uint32_t status;
  status = USART0->US_CSR;
  if (status & US_CSR_RXRDY)
    // receive interrupt
    ser0_put(USART0->US_RHR);

  // Unfortunately for all the register in the SAMS3 they do NOT have
  // an uart-interrupt-cause register. This means I have to process the UART
  // transmit status each time over and over
  if (status & US_CSR_TXRDY)
  { // Can transmit character
    if (serout_getpnt == serout_putpnt)
    { // nothing in queue
      // disable further UART TX interrupts
      USART0->US_IDR = US_IDR_TXRDY;
    }
    else
    { // send character out
      USART0->US_THR = ser_outq[serout_getpnt];
      serout_getpnt = (serout_getpnt + 1) & (SER_OUTQ_LEN -1);
    }
  }
} // USART0_Handler

//** That is the same method I used 35 years ago with the Motorola 6850.
// I just LOVE devices which give a continuous interrupts whilst there is
// space in their output buffer. It makes for such elegant transmit interrupt
// routine code.
```