Chapter 4 UART & Joystick

4.1 UART: RS232 demo code

There are numerous examples on this topic. A whole chapter on RS232 serial communication is found inside "Microcontroller Projects in C for the 8051, Dogan Ibrahim". Serial communication is also called UART on data sheet of most microcontrollers, including AT89S52. There is a complete application note on C51 UART Program Examples on Atmel web page (www.atmel.com) as well.

On AT89S52-gLCD-STK1, U2 is the transceiver to convert TTL levels (0V to +5V) of AT89S52 to RS232 levels (-12V to +12V) of a PC. It is safe to connect the PC's COM port (9-pin male) directly with the female port of our development board by a straight 9-pin cable.

The source code of this section is found under cd:\src\chp4\src4_1\. Project name is UART1.uv2.

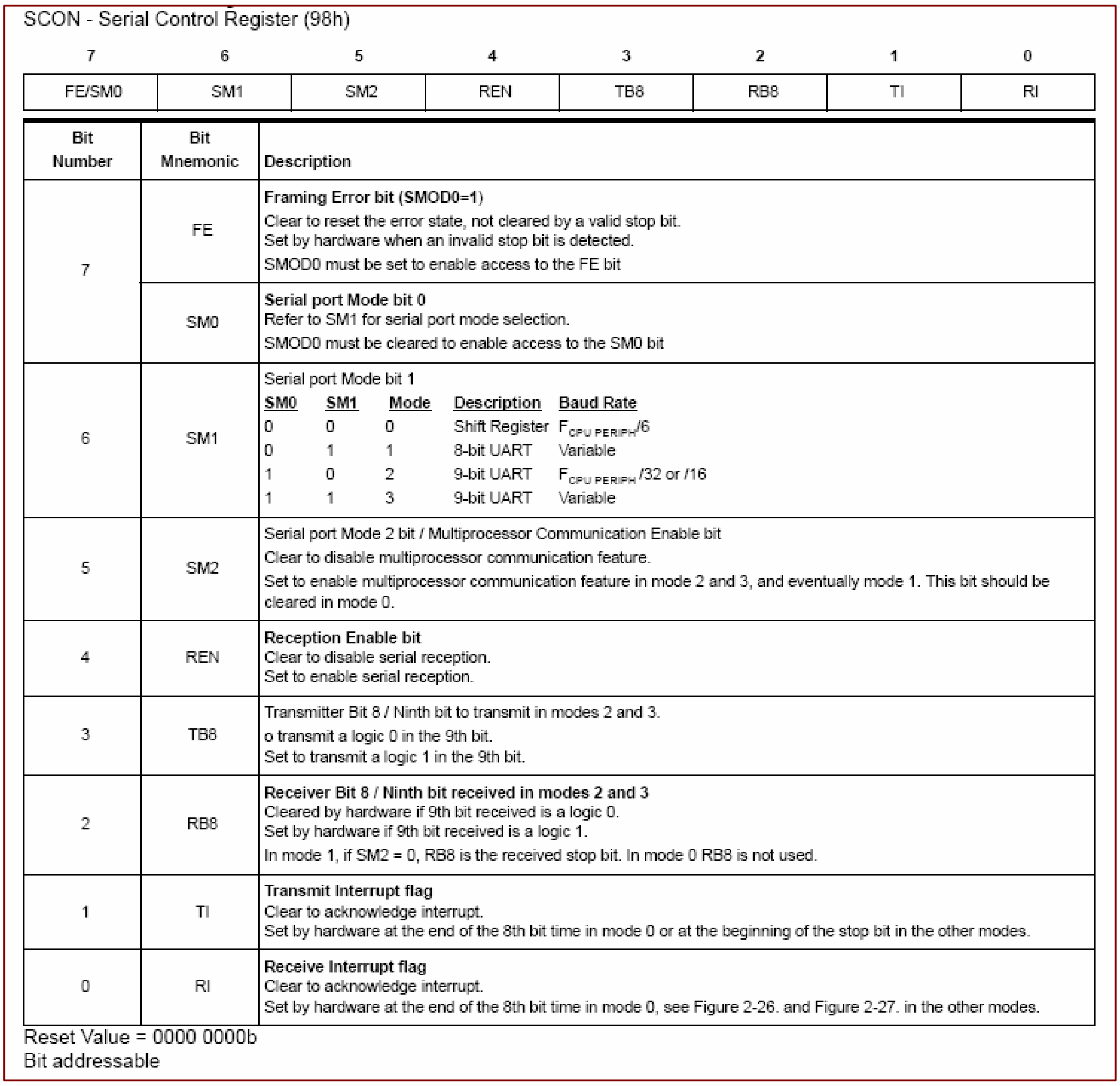
```
#include <REGX52.h>
#include "delay.h"
#include <stdio.h>                                                          (1)

unsigned char temperature;
unsigned char humidity;

void uartInit(void)
{
        SCON  = 0x52;                                                       (2)
        TMOD  = TMOD|0x20;                                                  (3)
        TH1   = 0xfd;                                                       (4)
        TR1   = 1;                                                          (5)
}

void main(void)
 {
  uartInit();
  for(;;)
  {
   printf("Temperature : %bu  Humidity : %bu \n", temperature++, humidity++);   (6)
   DelayMs(1000);
  }
}
```

Listing 4.1.1

Line (1) includes the standard library stdio.h header file which is necessary for printf function in the main loop.

Line (2) configures the serial port control register SCON. Structure of SCON is found in the 8051 hardware manual published by Atmel. Extract of the table is shown in Figure 4.1.1.

SCON = 0x52 => SM1 = 1, REN = 1, and TI = 1. The rest bits all 0.



SCON - Serial Control Register (98h)

7	6	5	4	3	2	1	0
FE/SM0	SM1	SM2	REN	TB8	RB8	TI	RI

Bit Number	Bit Mnemonic	Description
7	FE	**Framing Error bit (SMOD0=1)** Clear to reset the error state, not cleared by a valid stop bit. Set by hardware when an invalid stop bit is detected. SMOD0 must be set to enable access to the FE bit
	SM0	**Serial port Mode bit 0** Refer to SM1 for serial port mode selection. SMOD0 must be cleared to enable access to the SM0 bit
6	SM1	Serial port Mode bit 1 — SM0 / SM1 / Mode / Description / Baud Rate: 0 / 0 / 0 / Shift Register / $F_{CPU\ PERIPH}/6$; 0 / 1 / 1 / 8-bit UART / Variable; 1 / 0 / 2 / 9-bit UART / $F_{CPU\ PERIPH}$ /32 or /16; 1 / 1 / 3 / 9-bit UART / Variable
5	SM2	Serial port Mode 2 bit / Multiprocessor Communication Enable bit Clear to disable multiprocessor communication feature. Set to enable multiprocessor communication feature in mode 2 and 3, and eventually mode 1. This bit should be cleared in mode 0.
4	REN	**Reception Enable bit** Clear to disable serial reception. Set to enable serial reception.
3	TB8	Transmitter Bit 8 / Ninth bit to transmit in modes 2 and 3. o transmit a logic 0 in the 9th bit. Set to transmit a logic 1 in the 9th bit.
2	RB8	**Receiver Bit 8 / Ninth bit received in modes 2 and 3** Cleared by hardware if 9th bit received is a logic 0. Set by hardware if 9th bit received is a logic 1. In mode 1, if SM2 = 0, RB8 is the received stop bit. In mode 0 RB8 is not used.
1	TI	**Transmit Interrupt flag** Clear to acknowledge interrupt. Set by hardware at the end of the 8th bit time in mode 0 or at the beginning of the stop bit in the other modes.
0	RI	**Receive Interrupt flag** Clear to acknowledge interrupt. Set by hardware at the end of the 8th bit time in mode 0, see Figure 2-26. and Figure 2-27. in the other modes.

Reset Value = 0000 0000b
Bit addressable

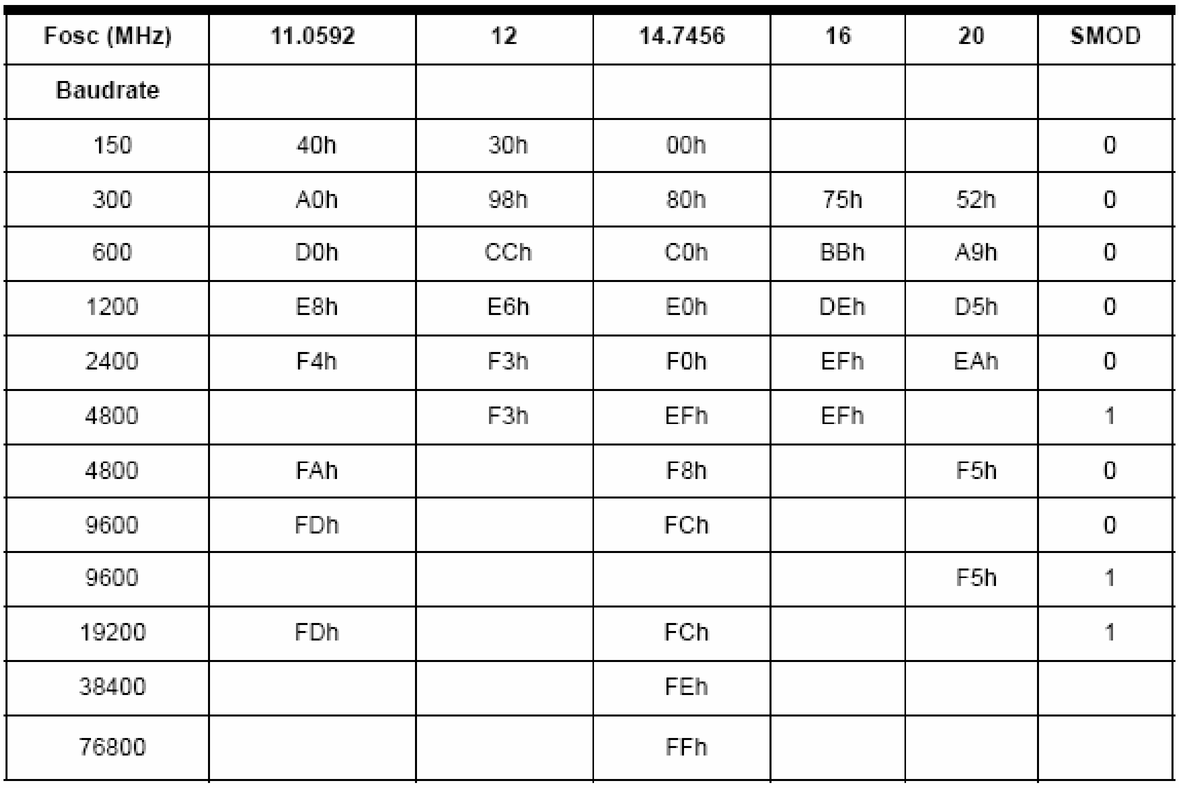
Figure 4.1.1

Line (3) configures Timer 1 for 8-bit auto-reload timer mode while leaving Timer 0 unchanged

Line (4) loads TH1 register for a baud rate of 9,600 bps. We don't need to calculate TH1 value every time we need to change the baud rate. There is a table in the 8051 hardware manual. Figure 4.1.2 shows an extract of that table. We are using a crystal frequency of 11.0592MHz. When SMOD = 0 (default reset value), we need to load TH1 a value of FDh for 9,600bps.

If a faster baud rate is required, we may set SMOD to 1 by keeping the same value of TH1 at FDh. The baud rate would be 19,200bps.

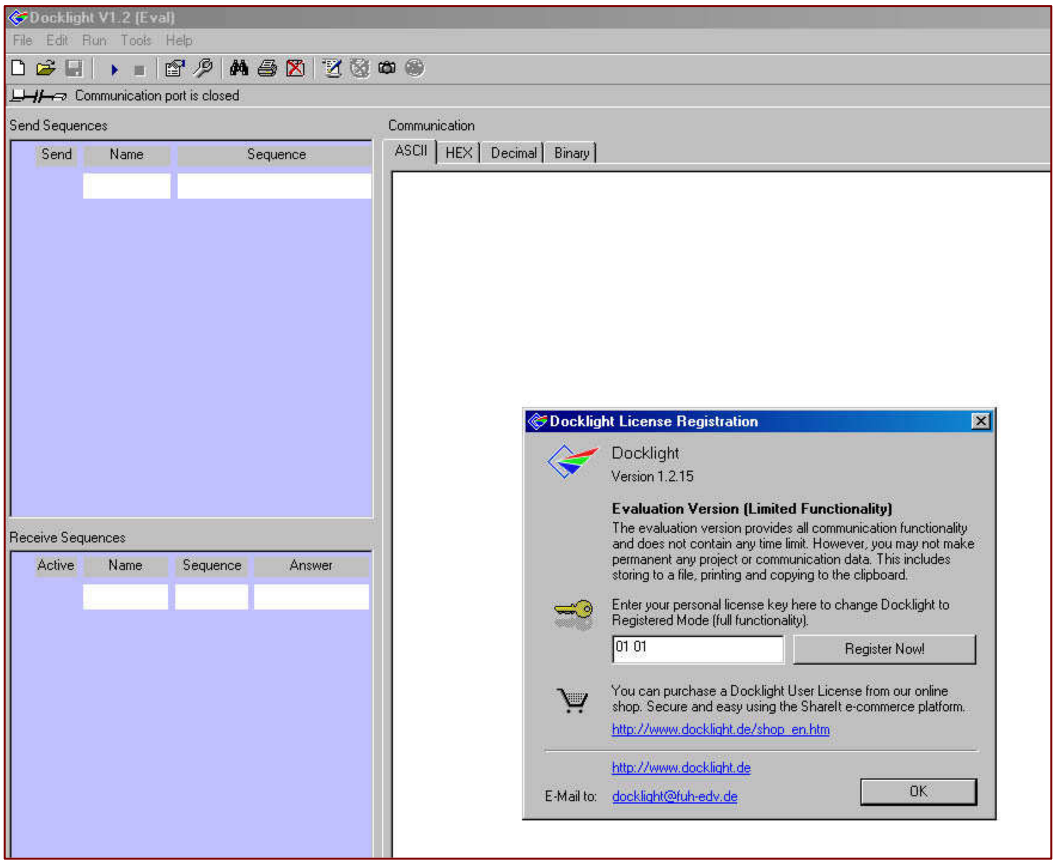
SMOD1 is bit 7 of PCON. Therefore we may add the line PCON|=0x80 between line 2 and line 3 to change the baud rate to 19,200bps.



Fosc (MHz)	11.0592	12	14.7456	16	20	SMOD
Baudrate						
150	40h	30h	00h			0
300	A0h	98h	80h	75h	52h	0
600	D0h	CCh	C0h	BBh	A9h	0
1200	E8h	E6h	E0h	DEh	D5h	0
2400	F4h	F3h	F0h	EFh	EAh	0
4800		F3h	EFh	EFh		1
4800	FAh		F8h		F5h	0
9600	FDh		FCh			0
9600					F5h	1
19200	FDh		FCh			1
38400			FEh			
76800			FFh			

Figure 4.1.2 Extract from page 99 of Atmel 8051 Microcontrollers Hardware Manual

Docklight is a test, analysis and simulation tool for serial communication protocols (RS232, RS485/422 and others). Its evaluation version can be downloaded from www.docklight.de. Download a copy from the web site and get it installed. Launching Docklight you will see the following screen (Figure 4.1.3). Click **OK** to skip the startup screen.



Docklight
Version 1.2.15

Evaluation Version (Limited Functionality)
The evaluation version provides all communication functionality and does not contain any time limit. However, you may not make permanent any project or communication data. This includes storing to a file, printing and copying to the clipboard.

Enter your personal license key here to change Docklight to Registered Mode (full functionality).

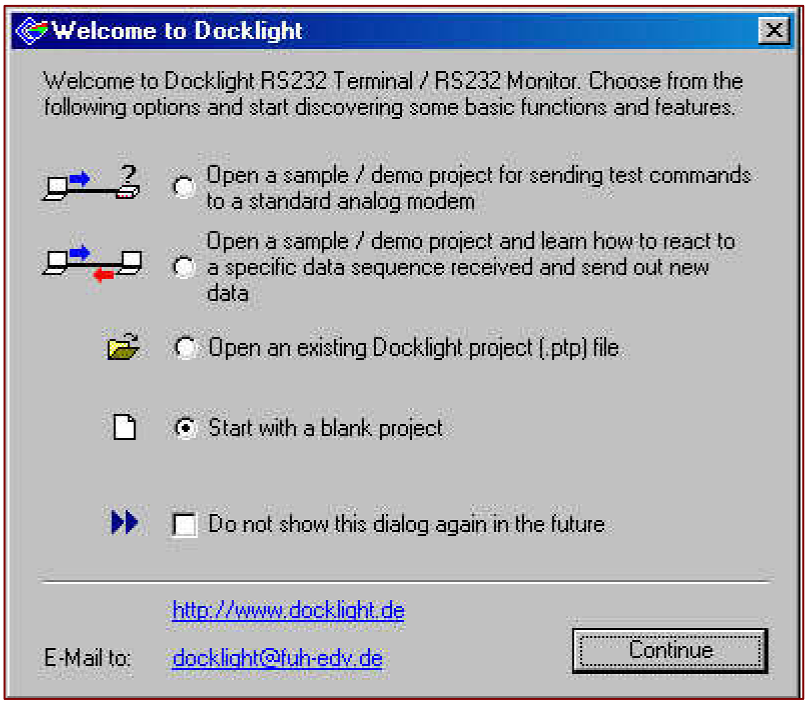
01 01

You can purchase a Docklight User License from our online shop. Secure and easy using the ShareIt e-commerce platform.
http://www.docklight.de/shop_en.htm

http://www.docklight.de
E-Mail to: docklight@fuh-edv.de

Figure 4.1.3 Startup screen

Start with a blank project, click **Continue**.



Welcome to Docklight

Welcome to Docklight RS232 Terminal / RS232 Monitor. Choose from the following options and start discovering some basic functions and features.

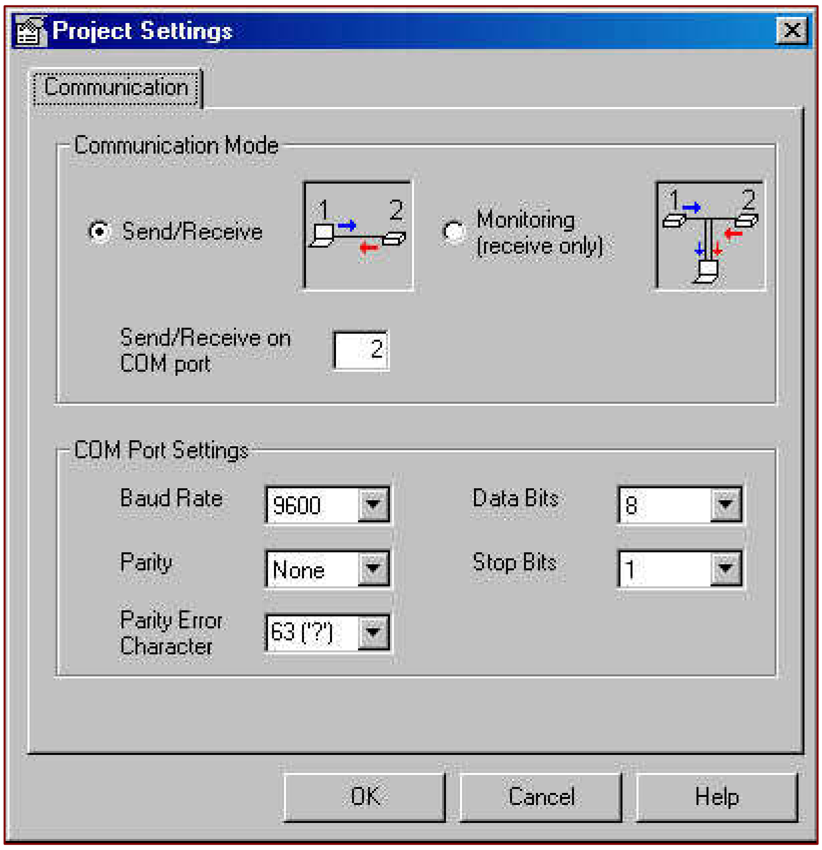
- ○ Open a sample / demo project for sending test commands to a standard analog modem
- ○ Open a sample / demo project and learn how to react to a specific data sequence received and send out new data
- ○ Open an existing Docklight project (.ptp) file
- [✓ Start with a blank project]
- ☐ Do not show this dialog again in the future

http://www.docklight.de

E-Mail to: docklight@fuh-edv.de

Figure 4.1.4

This is an important step. Under **Tools->Project Settings** to bring up the Project Settings dialog box. Select the COM port you use on the PC, and a Baud Rate of 9600 under the COM Port Settings.



Project Settings

Communication

Communication Mode

[✓ Send/Receive] ○ Monitoring (receive only)

Send/Receive on COM port: 2

COM Port Settings

Baud Rate	9600	Data Bits	8
Parity	None	Stop Bits	1
Parity Error Character	63 ('?')		

Figure 4.1.5

Finally, click on the Play icon to start communication.

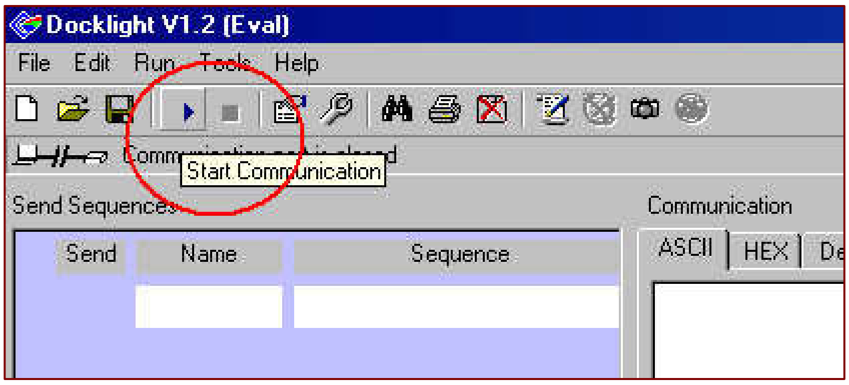
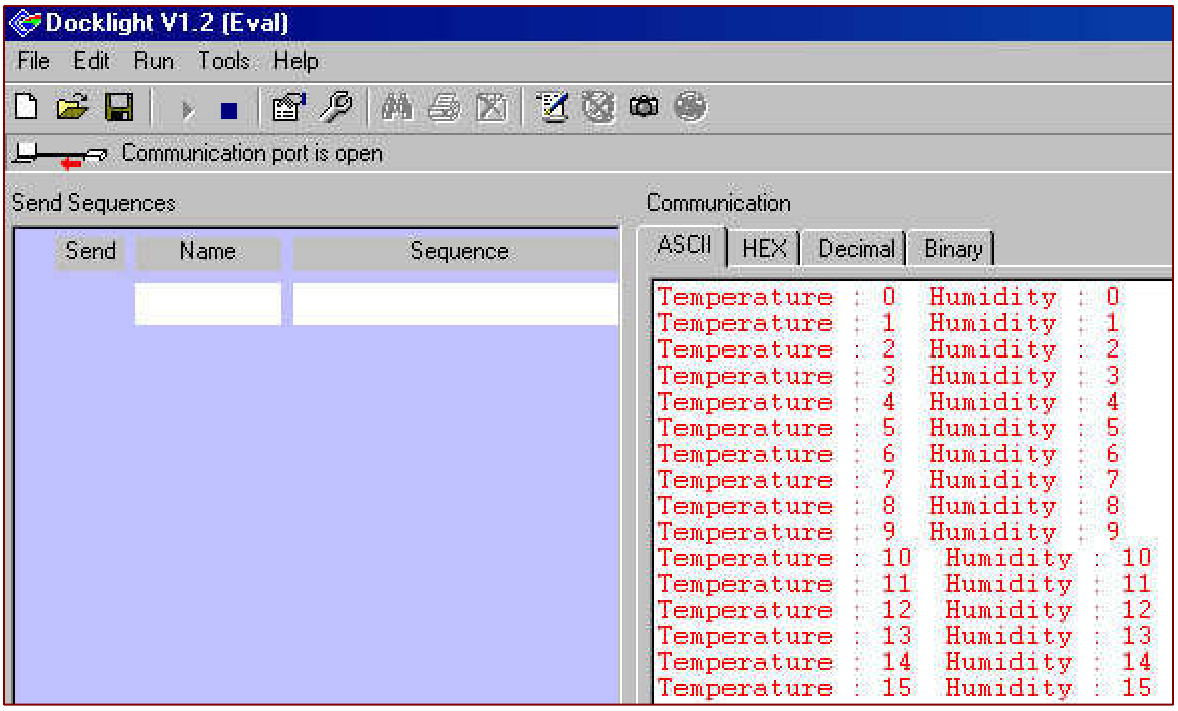


Figure 4.1.6

After having downloaded the hex code UART1.hex under cd:\src\chp4\src4_1\ to the mcu, you will see dummy data as shown below. We have a handy tool to debug a program now.



```
Temperature : 0  Humidity : 0
Temperature : 1  Humidity : 1
Temperature : 2  Humidity : 2
Temperature : 3  Humidity : 3
Temperature : 4  Humidity : 4
Temperature : 5  Humidity : 5
Temperature : 6  Humidity : 6
Temperature : 7  Humidity : 7
Temperature : 8  Humidity : 8
Temperature : 9  Humidity : 9
Temperature : 10  Humidity : 10
Temperature : 11  Humidity : 11
Temperature : 12  Humidity : 12
Temperature : 13  Humidity : 13
Temperature : 14  Humidity : 14
Temperature : 15  Humidity : 15
```

Figure 4.1.7

4.2 Joystick: Let's debug the driver of 5-way navigator joystick

It is very important to be able to debug a program during software development. Unlike the PC counterpart, there is no monitor for small microcontrollers. Serial data analysis software like Docklight allows us to keep track of our program flow by using technique similar to what we have done in section 4.1. There is a better alternative of course. Emulator is a common technique but it requires extra hardware. So, we would concentrate on using Docklight for debug at the moment.

Program in this section monitors joystick S1 for key press. It is a 5-way navigator joystick in the sense that, it is possible to click it up, down, right, left, and center. Its position is highlighted in Figure 4.2.1. Its simplified wiring diagram is shown in Figure 4.2.2. The actual schematic is more complicated though, but the idea behind is the same. Please refer to the full schematic for details.

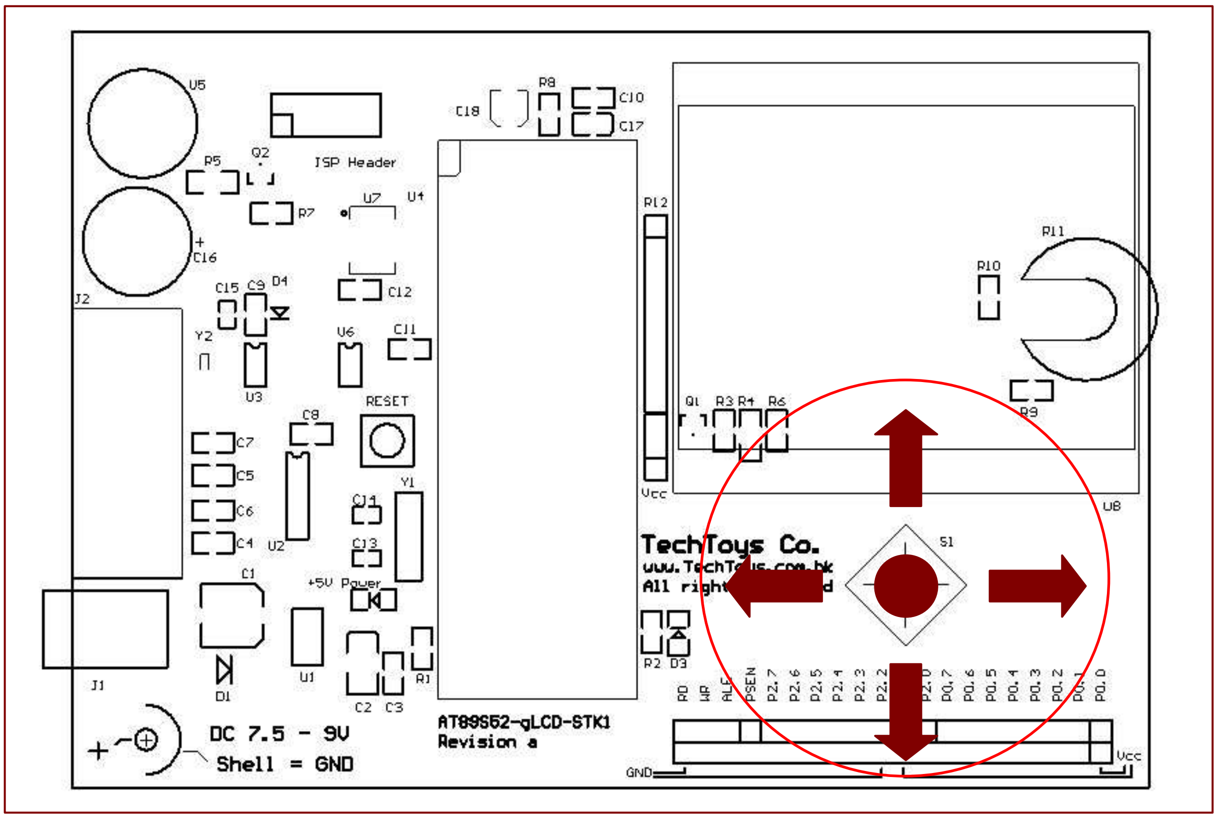


Figure 4.2.1

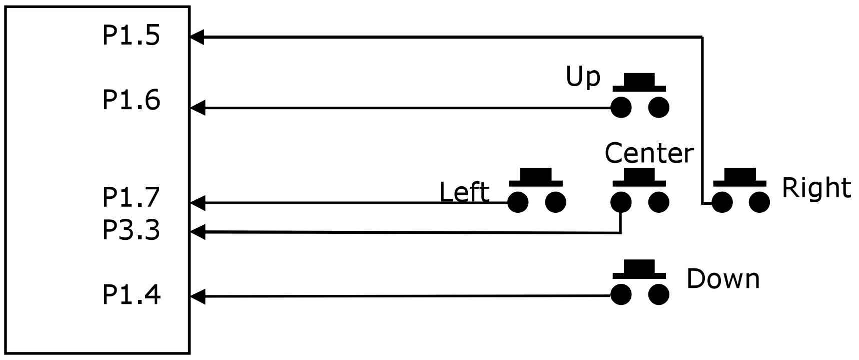


Figure 4.2.2 Remarks: keys' pins short to ground to read 0 when pressed

Again, there are many references from the internet on this topic. However, the best keyboard/keypad driver I've seen is from Jean J. Labrosse's book, Embedded System Building Blocks, 2nd edition.

The keypad module presented on Jean's book concentrates on matrix keyboard. The following features are available[1]:

- Scan any keyboard arrangement from 3x3 to an 8x8 key matrix
- Provides buffering with user configurable buffer size
- Supports auto-repeat
- Keeps track of how long a key has been pressed
- Allows up to three Shift keys

Because our navigator joystick has been wired in direct I/O configuration, a simplified version would be presented here with the following features for the sack of a smaller module to fit in 2K limit of eval Keil C:

- Scan individual switch
- Provides buffering with user configuration buffer size
- Supports auto-repeat
- Keeps track of how long a key has been pressed

The source code of this program is found under cd:\src\chp4\src4_2\. The project name is keyMON.uv2.

Joystick functions have been included in form of a software module. This is similar to software module provided by delay.c, however, the module xKey.c is much more complicated.

Listing 4.2.1 shows the xKey.h file. It would be enough for us to make use of all functions provided by the xKey.c module by just learning the constants and function prototypes (Application Interface) defined under xKey.h instead of going through the details of xKey.c.

Module flow diagram is shown in Figure 4.2.3.

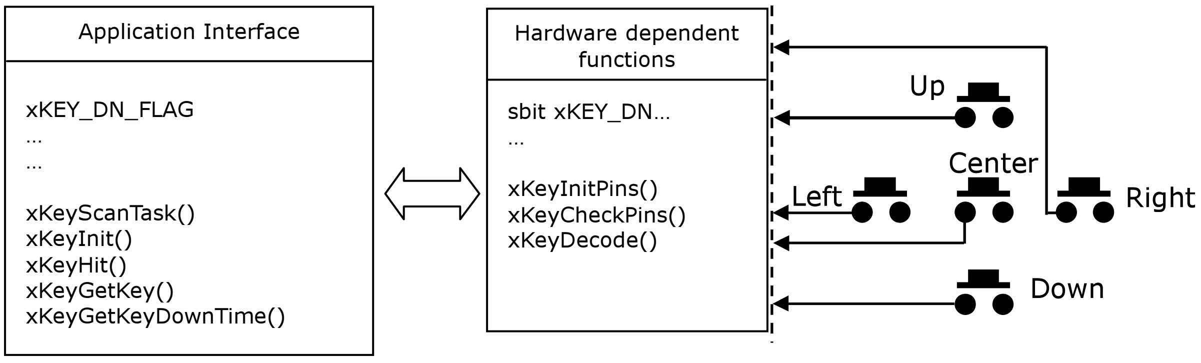


Figure 4.2.3

[1] Chapter 3, Keyboards, Embedded System Building Blocks, 2nd edition, by Jean J. Labrosse

```
/* Hardware pins suit AT89S52-gLCD-STK1 board */
sbit xKEY_DN  =      P1^4;                                  (1)
sbit xKEY_RT  =      P1^5;                                  (2)
sbit xKEY_LT  =      P1^7;                                  (3)
sbit xKEY_UP  =      P1^6;                                  (4)
sbit xKEY_CTR =      P3^3;                                  (5)

/* General definition */
#define  FALSE                      0                       (6)
#define  TRUE                       1                       (7)

/* Key definitions */
#define xKEY_DN_FLAG                1                       (8)
#define xKEY_RT_FLAG                2                       (9)
#define xKEY_LT_FLAG                3                       (10)
#define xKEY_UP_FLAG                4                       (11)
#define xKEY_CTR_FLAG               5                       (12)

/* Buffer size, delay constants etc. (see text) */
#define  xKEY_BUF_SIZE              3                       (13)
#define  xKEY_RPT_DLY               5                       (14)
#define  xKEY_RPT_START_DLY         30                      (15)
#define  xKEY_SCAN_TASK_DLY         50                      (16)

/* extern definition */
#ifdef  xKEY_GLOBALS                                        (17)
#define xKEY_EXT                                            (18)
#else
#define xKEY_EXT extern                                     (19)
#endif

/* Auto-repeat enable bit */
xKEY_EXT      bit      xKeyRptEn;                           (20)

/* API functions */
void                 xKeyScanTask(void);                    (21)
void                 xKeyInit(void);                        (22)
bit                  xKeyHit(void);                         (23)
unsigned char        xKeyGetKey (void);                     (24)
unsigned int         xKeyGetKeyDownTime(void);              (25)

/* Hardware dependent functions */
void                 xKeyInitPins(void);                    (26)
bit                  xKeyCheckPins(void);                   (27)
unsigned char        xKeyDecode(void);                      (28)
```

Listing 4.2.1 **xKey.h file**

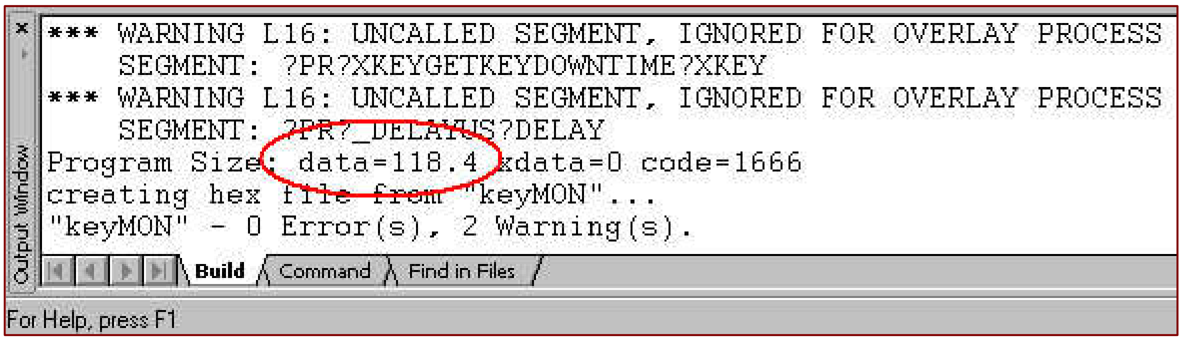
Line (1) – (5) provides hardware definition matching the development board. Caution: the type 'sbit' is specific to Keil C only. Other compilers may not use the same definition.

Line (6) & (7) defines the keyword FALSE and TRUE for portability.

Line (8) – (12) defines arbitrary constants for individual key. Constant xKEY_DN_FLAG (say), would be used in the main() program for detection of the DOWN key press.

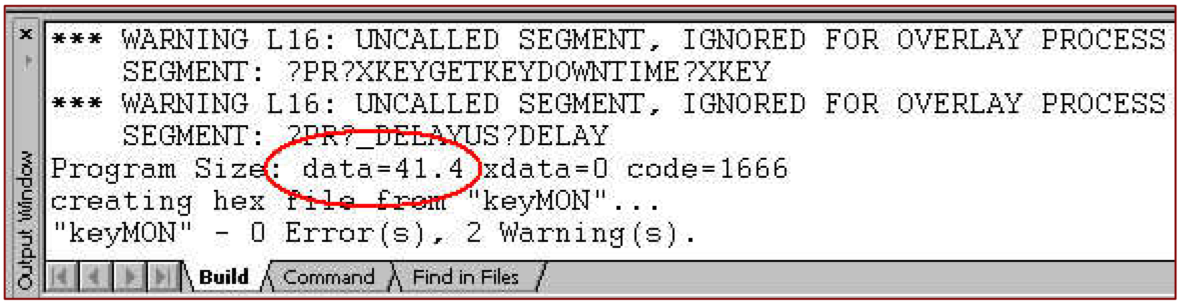
Line (13) defines the buffer size of the Joystick buffer. A cyclic buffer is implemented inside xKey.c. The direct consequence of using a buffer is that, it is possible to postpone reading the Joystick without losing keystrokes if there is/are more important tasks to be handled by the microcontroller. The size of the buffer depends on the application requirement. For our simple demonstration, a buffer size of 3 is good enough. It is possible to assign a large buffer of 50; unfortunately, key buffer is occupying RAM space so it fights for RAM resources with other tasks as well.

Figure 4.2.4a shows the data RAM used after setting xKEY_BUF_SIZE 80.
Figure 4.2.4b shows the RAM used with xKEY_BUF_SIZE 3.



```
*** WARNING L16: UNCALLED SEGMENT, IGNORED FOR OVERLAY PROCESS
    SEGMENT: ?PR?XKEYGETKEYDOWNTIME?XKEY
*** WARNING L16: UNCALLED SEGMENT, IGNORED FOR OVERLAY PROCESS
    SEGMENT: ?PR?_DELAYUS?DELAY
Program Size: data=118.4 xdata=0 code=1666
creating hex file from "keyMON"...
"keyMON" - 0 Error(s), 2 Warning(s).
```

Figure 4.2.4a



```
*** WARNING L16: UNCALLED SEGMENT, IGNORED FOR OVERLAY PROCESS
    SEGMENT: ?PR?XKEYGETKEYDOWNTIME?XKEY
*** WARNING L16: UNCALLED SEGMENT, IGNORED FOR OVERLAY PROCESS
    SEGMENT: ?PR?_DELAYUS?DELAY
Program Size: data=41.4 xdata=0 code=1666
creating hex file from "keyMON"...
"keyMON" - 0 Error(s), 2 Warning(s).
```

Figure 4.2.4b

Line (14) defines the number of scan times before auto-repeat executes again, i.e. it is the rate of auto-repeat. Scan time measured in units of xKEY_SCAN_TASK_DLY defined in Line (16).

Therefore, the actual time for auto-repeat in our case is 250 ms.

xKEY_RPT_DLY * xKEY_SCAN_TASK_DLY ms, which is 5*50ms

Line (15) defines the number of scan times before auto-repeat function started. Again, its unit is measured in xKEY_SCAN_TASK_DLY.

Line (16) is the number of milliseconds between keyboard scans. It is an important parameter for key debounce. We need debounce because switches are not perfect. They do not generate a clear-cut 1 or 0 when they are pressed or released. A normal person presses a key longer than 20ms, so a key debounce delay of 50ms is usually good enough.

Line (17) - (19) takes care of extern keyword for xKey.c and the application program. If we need a global variable that would be accessed by the application program, we need to prefix the variable with xKEY_EXT, like the case in xKeyRptEn.

Line (20) defines a global variable for enable/disable the auto-repeat feature. This is a Keil C specific variable as it is using the type 'bit'. xKeyRptEn defaults to FALSE on xKeyInit(), module initialization.

<u>Here come the API functions</u>:

Line (21)

xKeyScanTask()
void xKeyScanTask(void);

This is a direct modification from the original KeyScanTask(void *data) created by Jean J. Labrosse in his book Embedded System Building Blocks, 2nd edition. Because we do not have RTOS yet, this function has been made public, and the function OSTimeDlyHMSM() in the original version has also been removed. This function is the heart of the keyboard module. It should be called in a periodic manner with the rate defined by xKEY_SCAN_TASK_DLY. In this case, key debounce is taken care of during the xKEY_SCAN_TASK_DLY period. The principle is illustrated in Figure 4.2.5. Suppose at any time t_o , the key UP is clicked, after some time (max 50ms defined by xKEY_SCAN_TASK_DLY), xKeyScanTask() would be called for the first time. xKeyScanTask() would record the state of the key and just exit. A normal key press would last longer than 50 ms. When xKeyScanTask() is called the second time, algorithm inside xKeyScanTask() will decide if that key is still pressed. If "yes", the key code would be decoded and placed inside the Joystick buffer for future read; otherwise, no key code would be inserted as it must have been noise or not a real key stroke. It has been assumed that no key bounce will last longer than 50ms! If you've got a key like that, just through it away.
Example : Please refer to listing 4.2.2

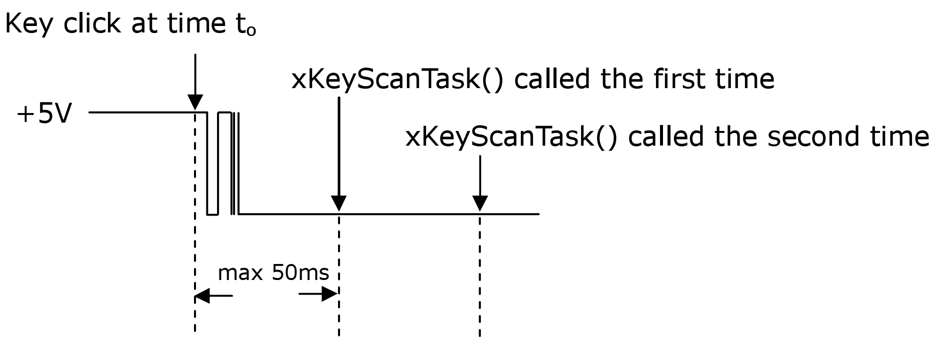

Key click at time t_o
xKeyScanTask() called the first time
+5V
xKeyScanTask() called the second time
max 50ms

Figure 4.2.5 **Key Debounce** (drawing not to scale, key bounce should be short)

Line (22)

xKeyInit()
void xKeyInit(void);

xKeyInit() is the initialization code for the module. It must be called before using any of the other functions. xKeyInit() is responsible for initializing internal variables used by the module and hardware ports as well.
Example : Please refer to listing 4.2.2

Line (23)

xKeyHit()
bit xKeyHit(void);

xKeyHit() allows your application to determine if a key has been pressed. It return TRUE if a key was pressed, and FALSE otherwise. **Arguments:** none **Return Value:** TRUE or FALSE **Warnings:** The return value type 'bit' is Keil C specific
Example : Please refer to listing 4.2.2

Line (24)

xKeyGetKey()
unsigned char xKeyGetKey(void);

xKeyGetKey() is called by the application to obtain a scan code from the Joystick buffer, in our case, scan code has been defined by:
 	
`/* Key definitions */`
`#define xKEY_DN_FLAG        1`
`#define xKEY_RT_FLAG        2`
`#define xKEY_LT_FLAG        3`
`#define xKEY_UP_FLAG        4`
`#define xKEY_CTR_FLAG       5`
 	
Arguments: none
Return Value: xKEY_XX_FLAG, or 0xFF if there is an error
 	
Warnings: This function should be called frequent enough before key buffer overflow.

Example : Please refer to listing 4.2.2

Line (25)

xKeyGetDownTime()
unsigned int xKeyGetDownTime(void);

xKeyGetKeyDownTime() returns the amount of time (in msec) that a key has been pressed. **Arguments:** none **Return Value:** The amount of time that the current key is being pressed. **Warnings:** The key down time is not cleared when the pressed key is released
Example: please request if an example is required.

Line (26) – (28) are hardware dependent functions. They are not explicitly called in user application. Please refer to the source code for details.

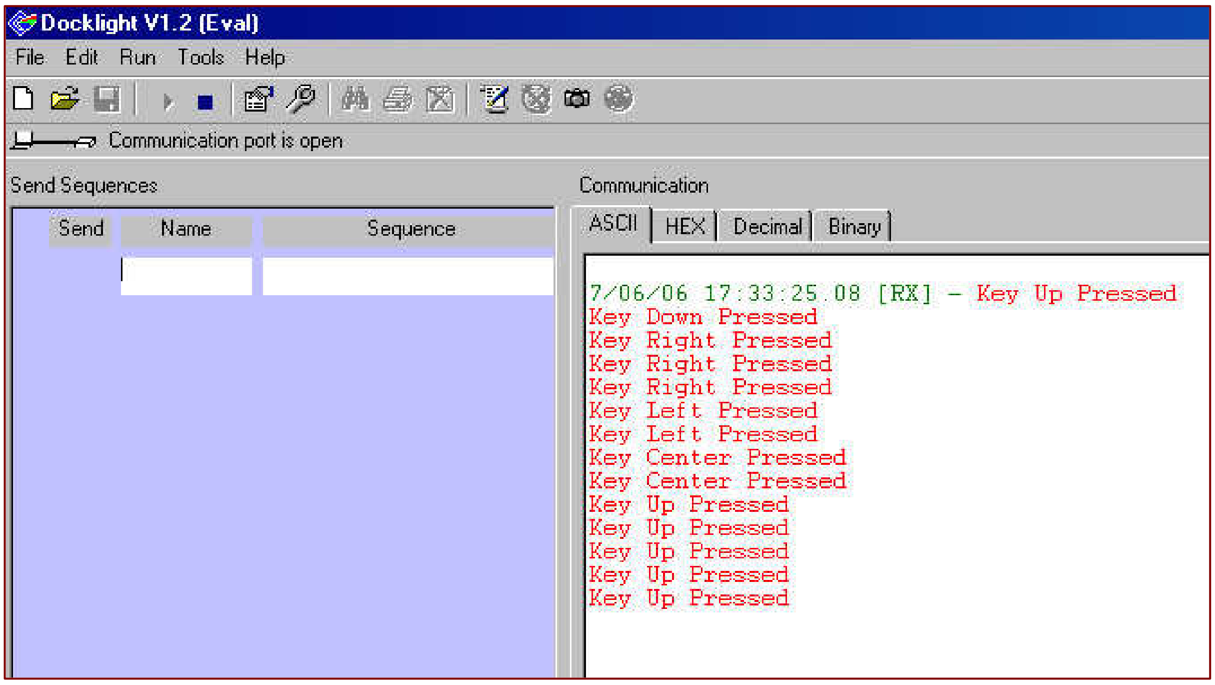
Listing 4.2.2 shows the demonstration application.

```
#include <REGX52.h>
#include "xKey.h"
#include "delay.h"
#include <stdio.h>

sbit LED = P2^0;

/* UART Initialization */
void uartInit(void)
{
        SCON  = 0x52;                   //UART in 8 bit mode
        TMOD  = TMOD|0x20;              //Timer 1 configured for 8-bit auto-reload timer mode,
                                        //Timer 0 unchanged
        TH1   = 0xfd;                   //TH1 value for a baud rate 9600bps
        TR1   = 1;                      //Timer 1 is turned ON
}

/* Main */
void main (void){

        unsigned char keyCode;

        xKeyInit();
        uartInit();

        for(;;){
                DelayMs(xKEY_SCAN_TASK_DLY);            //software delay for key debounce
                xKeyScanTask();                         //Key scan
                if(xKeyHit())
                {
                        keyCode = xKeyGetKey();
                        switch (keyCode)
                        {
                                case xKEY_UP_FLAG:
                                        xKeyRptEn=TRUE;
                                        printf("Key Up Pressed \n");
                                        break;
                                case xKEY_DN_FLAG:
                                        xKeyRptEn=TRUE;
                                        printf("Key Down Pressed \n");
                                        break;
                                case xKEY_RT_FLAG:
                                        xKeyRptEn=FALSE;
                                        printf("Key Right Pressed \n");
                                        break;
                                case xKEY_LT_FLAG:
                                        xKeyRptEn=FALSE;
                                        printf("Key Left Pressed \n");
                                        break;
                                case xKEY_CTR_FLAG:
                                        xKeyRptEn=FALSE;
                                        printf("Key Center Pressed \n");
                                        break;
                                default:  ;//inform user of an error, optional
                        }
                }
                LED = ~LED;
                }
}
```

Listing 4.2.2 5-way navigator joystick demonstration program

Program on listing 4.2.2 checks for any key press on S1 (5-way navigator joystick). Key debounce achieved by simple software delay function DelayMs(xKEY_SCAN_TASK_DLY)[2]. If key UP or DOWN pressed, the corresponding text message is transmitted via UART for debug purpose. Auto-repeat feature is enabled with UP/DOWN, so holding these keys would repeat sending text messages. Similarly, text messages debug feature included for keys RIGHT, LEFT, and CENTRE, with auto-repeat disabled. At the same time, an LED is blinked at a rate of 1/50ms (20Hz) to show the software loop is running under the extreme conditions of a key hold. Baud rate is 9600bps, Timer 1 is used as the baud rate generator. Standard library stdio.h is also included for printf function. Code size is 1664 bytes if printf is used; else, the code is around 494 bytes.

Launch Docklight to see the result as below (Figure 4.2.6). Try clicking S1 up, down, right, etc. Hold keys UP/DOWN and get a feeling of the auto-repeat feature. Now we know our xKey.c driver is at least working.



```
7/06/06 17:33:25.08 [RX] - Key Up Pressed
Key Down Pressed
Key Right Pressed
Key Right Pressed
Key Right Pressed
Key Left Pressed
Key Left Pressed
Key Center Pressed
Key Center Pressed
Key Up Pressed
Key Up Pressed
Key Up Pressed
Key Up Pressed
Key Up Pressed
```

Figure 4.2.6

⇒ Chapter 5

[2] Timer interrupt should be used for more serious applications