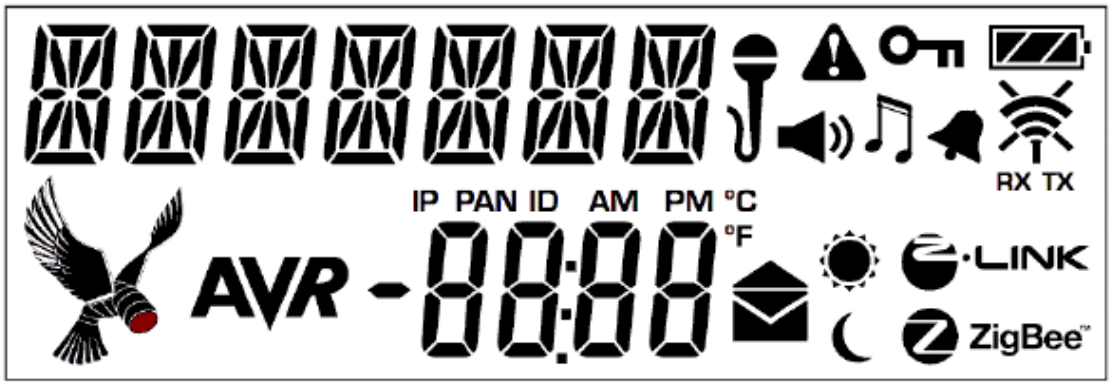
RZRAVEN Development Kit

The development kit contains 1X USB Zigbee radio controller & 2 X AVR Raven boards. The AVR Raven boards have plentiful I/O pins lead out to conectors, and an LCD with 7 alphanumeric digits, 4 digits, and various other symbols – see Figure 3-2 below.

Figure 3-2 AVRRAVEN - LCD Segments



RZRAVEN comes with a suite of Atmel software which allows text messages to be sent or received from RVRAVEN boards. This is well documented via Atmel. There is also available some ArchRock sample code, see below.

Guide to ArchRock code

Use given firmware on USBAVRSTICK – this implements an IP connection over Zigbee and never needs to be changed. Archrock provide a windows service which wraps this connection so that the USBAVRSTICK appears as a network interface and standard TCP/IP tools (telnet application, sockets inside programs, etc, can communicate with the remote board). There is also a wireless management web page which can be used to change settings and display connected devices etc – see archrock **windows service setup guide** documentation.

Use archrock "shell" application on RVRAVEN processors. This can be demonstrated using given firmware on AtMega1280 & AtMega3290P on AVR raven board.

To adapt this application for more general use:

See the provided **rzraven_atmel3290p_proj** files for source code on 3290P which accepts commands from the 1280 and interprets them to write to LCD etc. Most of this code is the LCD driver (and some other hardware interfaces). The key command handler which is given the next command from the UART and interprets it is in **app.c**:

```
/*  Process incoming uart message.
*/
void signal_uart_msg(cmd_msg_t *p_msg, uint16_t payloadlen)
{
      int num, tempval;
```

```
	switch(p_msg->header.type)
	{
		case LED:
			if (p_msg->header.cmd == ON)
				led_on();
			else if (p_msg->header.cmd == OFF)
				led_off();
			break;
		case LCD:
			if (p_msg->header.cmd == DISPLAY_MSG)
			{
				lcd_textbuf_clr();
				lcd_puta(payloadlen, p_msg->payload);
			}else if (p_msg->header.cmd == DISPLAY_NUM){
				num = *((int16_t *)p_msg->payload);
				lcd_num_clr();
				lcd_num_putdec(num, LCD_NUM_PADDING_SPACE);
			}else if (p_msg->header.cmd == DISPLAY_TEMP){
				tempval = get_temp();
				lcd_symbol_set(LCD_SYMBOL_F);
				lcd_num_clr();
				lcd_num_putdec(tempval, LCD_NUM_PADDING_SPACE);
			}
			break;
		case SPEAKER:
			break;

	}
}
```

NB – signal_uart_msg is called from the UART receive Interrupt routine and executed as interrupt code. In fact the whole of this application executes from UART receive interrupts – the main() function sets up the UART & LCD and returns.

Writing more complex code

For a user application it may be necessary to send data back to the 1280 from the 3290P. This means configuring and using the transmit channel of the UART, and the code to do this is not available in the archrock project. Required extra code to implement this is:

- UART transmit function in 3290p code – for example add timer interrupt driven code which transmits characters and possibly also implements other tasks. This preserves the callback event-driven structure of the code. You have complete control over the 3290p – there is no underlying kernel – so you can do what you like.

- UART receive function in 1280 code. One way to implement the necessary receive functionality would be to use the kernel **setitimer** callback (see **ping** application for example and include file **timer.h** for interface) to regularly poll the UART receiver collecting sent data and sending it back to the PC over the IP link.

Understanding the ArchRock 1280 kernel code

A good introduction can be found in the **mainpage.h** kernel include file. The code is callback-based all functions must execute quickly and if necessary code requests a callback to continue execution at a later time, instead of waiting.

The corresponding code on the AtMega1280 which accepts text commands from a PC over a Zigbee IP link and sends them to the 3290P is the Archrock **shell** project (see **archrock_atmelraven_asd_eval.zip**) in file **cmd.c**.

Commands are defined in a table:

```
//! data base of supported shell commands
cmd_desc_t cmd_table[] = {
  {help_fun,   "?",     "print cmds"},
  {ledOn_fun,  "lon",   "LED on"},
  {ledOff_fun, "loff",  "LED off"},
  {blink_fun,  "blink", "Blink N"},
  {lcd_fun,    "lcd",   "display msg"},
  {temp_fun,   "temp",  "display temp"},
  {time_fun,   "time",  "time of day"},
  {rte_fun,    "route", "info"},
  {if_fun,     "ifconfig", "IP config"},
  {exit_fun,   "exit",  "exit shell"}
};
```

For example LCD shell commands “lcd” are sent to the 3290P over a UART by **lcd_fun()**:

```
int lcd_fun(char *msg, char *buf, int curindex, int len, int16_t
sockfd) {
  print_msg(&buf[curindex], len-curindex);
  sprintf(msg, "\r\n");
  return 0;
}
```

See **cmd.c** for other functions. To add new functions, add to this table, add a new helper function, add to the 3290P switch which receives and acts on commands.

Note that the given 3290P source code is not completely implemented – some functions are missing.

The function which processes commands using this table (which you should not need to change) is:

```
//! processCmd

//!
//! @brief process shell command
//!
//! Process command in command buffer cmd[]
//! starting at cmd[curindex] and ending at cmd[len]
//! Dispatches to individual command handlers
//!
//! @param cmd is the pointer to the string of shell command
//! @param curindex is always zero
//! @param len is the number of characters in the shell command
//! @param sockfd is the TCP socket descriptor
//!
//! @return 0 or positive integers : Normal
//! @return negative integers      : Close
//!
int processCmd(char *cmd, int curindex, int len, int16_t sockfd)
{
  int c, mc, mx, nindex;
  char msg[SERIALMSGSIZE];
  int status = 0;

  memset(msg,0,SERIALMSGSIZE);
  nindex = skip(cmd, curindex, len, " \r\n\t");
  if ((nindex >= 0) && (nindex < len-1)) {
    for (c = 0; c < sizeof(cmd_table)/sizeof(cmd_desc_t); c++) {
      mc = strlen(cmd_table[c].key);
      mx = len - nindex;
      if ((mc <= mx) && (strncmp(&cmd[nindex],cmd_table[c].key,mc)
== 0)) {
      status = cmd_table[c].fun(msg, cmd, nindex+mc+1,len, sockfd);
      send(sockfd, msg, strlen(msg), 0);
      if (status == 0) prompt(sockfd);
      return status;
       }
    }
    sprintf(msg, "bad cmd: %s",&cmd[nindex]);
    send(sockfd, msg, strlen(msg), 0);
  }
  prompt(sockfd);
  return 0;
}
```