Developer's Guide to comaidsystem.c, comaidsystem_functions.c, comaidsystem_functions.h, and keyboard.h

Brief System Description and Etymology:

The name comaidsystem is derived from Communication Aid System as the system is designed to assist on-road communication with deaf driver.
Logo:

Hardware specs: ATmega168-20PU microcontroller (AVR)
PS-2 Keyboard
New Haven LCD NHD-0116GZ-FSW-FBW

Rationale for Approach:

Our approach in designing *comaidsystem* was to consider it not as a ps-2 keyboard driver but rather as a dedicated system that fully integrate a ps2-keyboard driver in its structures. *comaidsystem is not a keyboard driver; it is a system featuring a fully integrated keyboard driver dedicated to assisting on-road communication with a deaf driver.* This philosophy explains why we did not model comaidsystem on the various free ps-2 keyboard drivers available mostly from online sources.

At the heart of *comaidsystem* is an array of scancode structs with 2 attributes, a char variable and a function pointer (Please, refer to description of keyboard.h for details). We opted for this model with upgradability and easy maintanance in mind. We beleive that this model optimizes upgradability becuase the function pointers for the various keys can easily be changed to point to new functions depending on what result we want a key press to produce. For instance, the system can easily be turned from a communication aid device to a game console or a motor control just by adjusting the function pointers and the hardware without

having to rewrite whole sections of core code. This array of scancode structs (scancodes[0xf1]) is directly indexable with scan-code values, however it has a gigantic memory footprint[only large for this environment!]. Thus our choice involved a trade-off between footprint and upgradability/maintenance/performance. We thought upgradability/ maintenance/performance was a better trade-off because without the scancode structures the code might have involved numerous *if* statements and *switch* blocks which are actually long assembly instructions resulting in relatively big footprint, not to mention code complexity and cumbersome upgradability. Also we thought future development for various dedicated real-time embedded systems might benefit from O(1) indexing, instead of an O(n) loop through a smaller array.

Many other approaches were possible, such as designing a tiny "OS" that would interact with a keyboard and an LCD driver. However, we belive our approach might be best for a dedicated embedded system with numerous possibilities for development of by-product dedicated systems. One of the easy improvements to the current software would be separate-chaining hashing with a hashcode function for hashing scancodes into a much smaller array.

The fact is, the code only uses a fraction of the available memory leaving room for more as shown in this AVR (ATmega168p) memory usage breakdown[Here “Program” memory is flash memory, thus read-only.]:

```
AVR Memory Usage
----------------
Device: atmega168p

Program:    3608 bytes (22.0% Full)
(.text + .data + .bootloader)

Data:        793 bytes (77.4% Full)
(.data + .bss + .noinit)
```

Top-level file:

comaidsytem.c:
Contains main(), calls to API (for initializations...), a "forever loop", and "pure-AVR" functions for enabling interrupts, setting up the keyboard, and handling interrupts. Its primary goal is to initialize the system in order to receive interrupts from the keyboard whenever a bit is sent, and then decode the received scancode (a series of bytes from the keyboard when a key is pressed, held, and/or released) direct indexing into the scancode struct array to atomically determine what opreation to execute, either display on LCD or specific commands (by calling the API's *process_scancode(unsigned char)* function). The content of the file is described below:

- **int main()**
  {
    bitcount = NUM_BITS; //initializes bitcount to 11 (1 start, 1 parity, 8 data, 1 stop

    init_sysvarStates(); //initializes system global variables

```
  enable_pcint(KB_PCINT);      //Enable pin change interrupts from the keyboard
clock

  initTables();                  //Initialize scancode tables

  keyboard_setup();              //Sets up the keyboard after BAT test

  sei();                         ///sets AVR global interrupt flag

  nlcd_init();
  nlcd_enable_scrolling();       // enable: nlcd_init does not by default

  deletefn(NO_USE_CHAR);     //clears the LCD screen

  while(1)                       //forever loop     . Everything will happen within the
interrupt handler whenever interrupts are set off by the keyboard
     ;

  return 0;
}
```

- **ISR(PCINT1_vect) :Interrupt Service Routine for pin change interrupts from PCINT8-14**

Receiving scancodes from the keyboard:
After the initialization that allows for the pin to be used as an interrupt when a rising edge is received, the scancode will set off the ISR for the PCINT that its signal is connected to.

1. Negative edges are what indicate that a bit has just been received. We are ignoring the start bit, the parity bit, and the stop bits.
We read whether or not the edge is negative or positive. The ISR is set off regardless of whether it is a rising or falling edge, so it was theoretically possible to simply have a variable alternate between two states to indicate whether we are on the rising or falling edge. When testing with this approach, however, we found that the variable switch was unreliable,while directly reading from the PCINT port [pin] was very reliable.
Using an *edge* switching variable would have been more appropriate if we had used AVR's ATmega168p designated external interrupt pins, INT0 and INT1. These pins would have been better suited for use with a switching *edge* variable as they can be set to trigger interrupts either at the rising, falling, high or low edges, However they are in use by the programmer and LCD in our configuration. Therefore we had to use the PCINT's (precisely PCINT11), Atmega168p pin change interrupts, which can be set as external interrupts by setting their data direction registers to input. However, the downside with them is that they toggle between high and low (whenever the keyboard sends signals) and set off interrupts whenever this happens.

The bitcount variable counts down from 11bits (start+stop+parity+8data)

2. The received data bits are buffered in a local char variable and once have received 11 bits altogether, we decode the isolated data bits by passing the local char variable holding them to the API's function process_scancode(char_data) as shown in the code below:

```
ISR(PCINT1_vect)
{
 static unsigned char char_data = 0;

 //If negative edge, ignore start, parity and stop bits and read bit
 if ( (PINC & (1 << PINC3)) == 0 ) {

   if (bitcount < NUM_BITS && bitcount > 2) {          //we only take the 8 data bits
     char_data = (char_data >> 1);
     if (PINC & 0x04)
       char_data = (char_data | 0x80);
   }

   //If we received a byte...
   if (--bitcount == 0) {
     bitcount = NUM_BITS;

     process_scancode(char_data);                      //Decode and process received
scancode
   }
 }
}
```

- **enable_pcint(int pcintnum) :Enables pin change interrupts**

AVR's ATmega168p provides INT0 and INT1 as main sources for external interrupts. These pins would have been better to use as they can be set to trigger interrupts either at the rising, falling, high or low edges, but they are used by the programmer and LCD in our configuration. Therefore we had to use the PCINT's (precisely PCINT11), Atmega168p pin change interrupts, which can be set as external interrupts by setting their data direction registers to input. However, the downside with them is that they interrupt on any toggle between high and low and set off interrupts whenever this happens.

Here is how we configured PCINT11 as an external interrupt source:

```
void enable_pcint(int pcintnum)
{
 if ((pcintnum >= 0) && (pcintnum < 8))                    //We use
PCINT11 but we could have used other PCINT's
```

```
    ;

  if ((pcintnum >= 8) && (pcintnum <= 14)) {                    //We use
PCINT11 but we could have used other PCINT's
    PCICR  |= 0x2;  //Enables pin change interrupts from PCINT8-14
    PCMSK1 |= 0x8;  //Unmasks pin change interrupt from PCINT11 only

    DDRC  &= ~(1 << DDC3);   //Sets PINC3 as input for data
    PORTC |= (1 << PORTC3); //Sets pull-up on PINC3
    MCUCR |= (1 << PUD);    //Completes Tri-state (Hi-Z) DDxn:0 PORTxn:1
PUD:1 (MCUCR)

    DDRC  &= ~(1 << DDC2);   //Sets PINC2 as input for data
    PORTC |= (1 << PORTC2); //Sets pull-up on PINC2
    MCUCR |= (1 << PUD);    //Completes Tri-state (Hi-Z) DDxn:0 PORTxn:1
PUD:1 (MCUCR)
  }

  if ((pcintnum >= 16) && (pcintnum <= 23))                     //We use
PCINT11 but we could have used other PCINT's
    ;
}
```

- **keyboard_setup(void) :Sets up the keyboard after BAT test, clears keyboard buffer and resets**

  We thought it would be better to actually reset the keyboard just after system start-up to make sure all devices are in sync.

```
void keyboard_setup(void)
{
  //Set up for output
  DDRC  |= (1 << DDC2);          //Sets PINC2 as output for data
  PORTC |= (1 << PORTC2);

  PINC = 0xFF;  //Tell keyboard to reset.

  delay_ms(100);

  //Set up for input again
  DDRC  &= ~(1 << DDC2);         //Sets PINC2 as input for data
  PORTC |= (1 << PORTC2);
}
```

Files Linked:

keyboard.h: contains constant declarations for keyboard scancodes and definition for the scancode struct (struct scancode_struct).
comaidsystem_functions.h: contains macros for heavily used chars and ints, and function definitions for comaidsystem_functions.c
comaidsystem_functions.c: contains the core API for comaidsystem.c, system global state variables declarations, constant array declarations, as well as EEPROM and Flash varibles and constants declarations.
nlcd.h: contains macros and function definitions for nlcd.c
nlcd.c: LCD driver (API for comaidsystem_functions.c to use) Please follow this link for lcd documentaion

- **keyboard.h:** contains constant declarations for keyboard scancodes and definition for the scancode struct (struct scancode_struct). defines the fundamental struct:

```
typedef struct scancode_struct{
  unsigned char ascii_char;
  void *(*scancode_function)(unsigned char);
} scode;
```

- **comaidsystem_functions.h:** contains macros for heavily used chars and ints, and prototypes for comaidsystem_functions.c as shown below:

```
 KB_PCINT 11       //Pin change interrupt 11
 SCODE_SIZE 0xF1   // Array size
 NUM_BITS 11       // number of bits to receive (1 start, 1 parity, 1 stop)
 BUFF_SIZE 17
 NO_USE_CHAR ' '   //passed to functions as non-used argument because scancode
structs' function pointers point to functions taking unsigned char arguments

void process_scancode(unsigned char scancode);
 //Function to process received scancode within PCINT ISR
void buffer_char(unsigned char);
 //function to buffer decoded chars, but actually just sends them to LCD
void enable_pcint(int pcintnum);                                    //enables
pinchange interrupts
void initTables(void);
//initializes the scancode struct array
void init_sysvarStates(void);                                       //initializes
states of system global variables
void keyboard_setup(void);                                          //sets up the
keyboard: clears buffer
void programfn(unsigned char);                                      //in
program mode, buffer and programs received chars
void program(unsigned char *, unsigned char*);                      //in
program mode, programs F-key display texts when called from FxFunction()
void read_eeprom_string(const uint8_t*);                            //read a
string from an eeprom  address

 //special key functions
```

```
void Fkeys_Function(unsigned char Fx_char);                    //Function
called by all Fx keys
void FxFunction(uint8_t* Fxtext, unsigned char* Default_Fxstring);     //called
within Fkeys_Function based on Fx_char
void end_codefn(unsigned char);                                //EN_CODE
function
void E0fn(unsigned char);                                      //E0 scancode
function
void E1fn(unsigned char);                                      //E1 scancode
function
void bkspfn(unsigned char);                                    //BKSP
scancode function
void deletefn(unsigned char);                                  //DELETE
scancode function
void homefn(unsigned char);                                    //HOME
scancode function
void enterfn(unsigned char);                                   //ENTER
scancode function
void escapefn(unsigned char);                                  //ESCAPE
scancode function
void l_ctrlfn(unsigned char);                                  //LCTRL and
R_CTRL scancode function
void caplockfn(unsigned char);                                 //CAPSLOCK
scancode function
void defaultfn(unsigned char);                                 //default
scancode function: for unimplemented keys
void shiftfn(unsigned char);                                   //SHIFT
scancode function
```

- **comaidsystem_functions.c:** contains the core API [implementation of key actions] for comaidsystem.c, system global state variables declarations, constant array declarations, as well as EEPROM and Flash varibles and constants declarations

-All uint8_t EEMEM F*x*text[18]; are char arrays stored in EEPROM for F-key text display programmability right from the keyboard

-unsigned char Fntext[18]; is a buffer for strings read from and written to EEPROM F*x*text[18]'s.

-scode scancodes[SCODE_SIZE]; is the array for all the scancode structures

-const unsigned char regular_keys[][2] PROGMEM is an array placed in flash memory to save RAM space, and that will serve to initialize the scancode struct array with direct scancode indexing into the appropriate scancode structure.

-const unsigned char shifted_keys[][2] PROGMEM is an array placed in flash memory for handling the shifted keys

- All the unsigned char F*x*default[] PROGMEM are constant strings holding the default text to display with the F*x*-keys. They are used during reverse programming to the default texts.

API Functions:

1) void initTables() : Function to intialize array of scancode structs with ascii chars and functions to execute

This function initializes scancodes[] array with the different scancode_structs

2)void init_sysvarStates(void): Function to initialize states of system global variables

This function initializes the various system global state variables

3)void process_scancode(unsigned char char_data) : Function to process received scancode within PCINT ISR

This function decodes the scancodes and execute a specific code by indexing into the scancodes[] array and calling the struct's function with its char attribute as an argument--which serves as an ID for the struct (i.e actual scancode's ascii character or a default character for unimplemented keys, or an ID for F-keys) .

Internal Functions:

4)void end_codefn(unsigned char empty) : END_CODE function, ignores next char

This sets *charsLeftToIgnore* so that the next received char can be ignored

5)void E0fn(unsigned char control): EXTENDED E0 function, enters control mode on 0xE0 and execute commands unless control repeat is set

This function sets *control_mode* to signal the PCINT11 ISR to skip all other processing and only process the next chars in control mode

6)void bkspfn(unsigned char empty)

This function calls an nlcd.c API to execute a backspace on the LCD

7)void deletefn(unsigned char empty): Delete function: resets LCD screen

This function calls an nlcd.c API to clear the LCD screen

8)void l_ctrlfn(unsigned char a_char): Left CONTROL function: enbales programming of Fx keys

This function sets *program_mode* to signal the PCINT11 ISR to also buffer the received chars for programming once a F-key is pressed

9) void homefn(unsigned char empty)

This function sets *program_default* to signal the *program (unsigned char *keytext, unsigned char* default_text)* to program the next pressed F-key with the default text to display

10)void enterfn(unsigned char empty): Enter function: not used at the moment
This function is a "nop" at this point

11)void escapefn(unsigned char empty): Escape function: Notifies need for emergency stop
This function calls an nlcd.c API to display "emergency stop"

12)void defaultfn(unsigned char key): Default function: does nothing
This function is a "nop" for all scancode_structs mapped to unimplemeted keys

13)void Fkeys_Function(unsigned char Fx_char): Function called by all Fx keys: its argument determines which arguments to pass to FxFunction
This function is mapped to all F-key structs and serves to identify which F-key command to execute through its argument, which is the char ID attribute of the indexed scancode_struct.

14)void FxFunction(uint8_t* Fxtext, unsigned char* Default_Fxstring): called within Fkeys_Function based on Fx_char
This function is called from *Fkeys_Function* with the proper strings for the specific F-key

15)void shiftfn(unsigned char char_received): Function to handle the shift key
This function sets *shift_pressed* to signal the PCINT11 ISR to only process the received scancode on the shifted_keys[] array.

16)void buffer_char(unsigned char thechar): buffer_char function: Buffers received and decoded characters. But actually just prints the char to LCD
This function is mapped to all the regular keys scancode_structs and sends the decoded scancodes to the LCD by calling an nlcd.c API, passing it its argument.

17)void programfn(unsigned char the_char): Function to buffer received chars for programming
This function , clled from the PCINT11 ISR when *program_mode* is set, serves to buffers the received chars for programming as soon as an F-key is pressed after CTRL and text typing.

18)void program (unsigned char *keytext, unsigned char* default_text): Function to program either received chars or default text into F-key eeprom addresses
This function performs the actual programming when an F-key is pressed after CTRL and text typing. It detrmines whether to program the buffered characters or rather the default string for the F-key that is pressed by checking whether *program_default* is set

19)void read_eeprom_string(const uint8_t * Ftext): Function to read contents from eeprom addresses into RAM buffer (Fntext)
This function reads a string from EEPROM (any EEMEM Fxtext[18]) and stores it in an array in RAM (Fntext[18]).

State variables:

IN REGULAR MODE:

control_mode

Activated: When an extended code scancode byte is sent, as when pressing the arrow keys, Page Up, Page Down, etc.

When active: The next received scancode will determine which key was pressed: what happens is up to the designer (we often implemented macros to these characters).

Deactivated: after programming when and F-key is pressed

charsLeftToIgnore

Activated: To value of one after a normal character's scancode has been received from the keyboard.

When active: The next full byte (scancode) received from the keyboard will be ignored

Deactivated: after a character has been ignored by the PCINT11 ISR

IN CONTROL MODE:

control_repeat

Activated: whenever a key whose scancode starts with E0 is pressed

When active: disables text displayed by an E0-key from being repeated when the key is held down (if it is one of the extended code scancodes noted in control_mode description.)

Deactivated: When an EXTENDED code key is released within control mode

IN PROGRAMMING MODE:

program_mode

Activated: When the Ctrl key is pressed.

When active: Places any received characters into a buffer for programming into a specified Function key (F1, F2, etc.).

Deactivated: after programming when and F-key is pressed

program_default

Activated: When Home is pressed immediately following the Ctrl key being pressed.

When active: The Function key next pressed will be restored to its factory-default macro.

Deactivated: after programming when and F-key is pressed within programming mode

IN SHIFT MODE:

shift_pressed

Activated: When the Shift key is pressed or held

When active: As expected on a keyboard, prints the shifted character (!, @, #, ?, etc.) for the key pressed. Note that this does not print capital letters: lower letters are not used in this code.

Deactivated: When the SHIFT key is released (END_CODE + SHIFT CODE)

shift_release

Activated: When a key is released within Shift mode

Deactivated: When the scancode following the END_CODE is not SHIFT