## Common Terminal Interface

In many embedded controller applications, there is a need for a simple, light-weight terminal interface. With very limited flash and SRAM, it is important that this interface use as few memory resources as possible.

In this interface, it is suggested that no C libraries be used to code the solution, allowing the entire interface to be coded in less than 1KiB of flash while using less than 128B of stack-based SRAM and no heap allocation. Not using the C library decreases the size of memory and also helps to mitigate multi-threaded C library reentrancy issues in RTOS solutions.

The proposed solution is broken into several parts:

- A user data structure defined as:

```
#define MAX_CHARS 80
#define MAX_FIELDS 5
typedef struct _USER_DATA
{
    char buffer[MAX_CHARS+1];
    uint8_t fieldCount;
    uint8_t fieldPosition[MAX_FIELDS];
    char fieldType[MAX_FIELDS];
} USER_DATA;
```

- getsUart0(USER_DATA* data)
  This is a function to receive characters from the user interface, processing special characters such as backspace and writing the resultant string into the buffer.
- parseFields(USER_DATA* data)
  This is a function that takes the buffer string from the getsUart0() function and processes the string in-place and returns information about the parsed fields in fieldCount, fieldPosition, and fieldType.
- char* getFieldString(USER_DATA* data, uint8_t fieldNumber)
  Returns the value of a field requested if the field number is in range or NULL otherwise.
- int32_t getFieldInteger(USER_DATA* data, uint8_t fieldNumber)
  Returns a pointer to the field requested if the field number is in range and the field type is numeric or 0 otherwise.
- bool isCommand(USER_DATA* data, const char strCommand[], uint8_t minArguments)
  This function returns true if the command matches the first field and the number of arguments (excluding the command field) is greater than or equal to the requested number of minimum arguments.

It is suggested that getsUart0() implement the following logic:

- If the count of characters in the buffer is > 0, process backspace characters (ASCII code 8 or 127) by decrementing the count of received characters, effectively erasing the character from the buffer.
- If the character received is a line feed (ASCII code 10) or carriage return (ASCII code 13), add a null terminator to the end of the buffer and return.
- Ignore any other characters that are unprintable (ASCII code < 32 “space”)

- For the printable characters, add each character received to the buffer, increment the character count, and return from the function if the count of characters in the buffer is equal to MAX_CHARS. You may want to make the interface case insensitive. If this behavior is desired, convert upper-case to lower-case or vice-versa to make string comparisons easier.

It is suggested that parseFields() implement the following logic:

- Decide on 3 sets of characters – alpha, numeric, and delimiter. Alpha is a-z and A-Z, numeric is 0-9 and optionally hyphen and period (or comma in some localizations), and everything else is a delimiter.
- Assume that the previous character type is a delimiter when starting to search the buffer.
- Go through the buffer from left to right, looking for the start of a field (a transition from a delimiter to a alpha or numeric character). For each field (at the transition), record the type of field (alpha or numeric – you can use ‘a’ or ‘n’ if you wish) in the type array, and the offset of the field within the buffer of the field in the position array, and increment the field count. Make the previous character stored equal to the new character and keep moving through the buffer string until the end is found. If the field count equals MAX_FIELDS, return from the function.
- Before returning, convert all delimiters in the string to NULL characters to aid in getter functions to follow.

This is the intended application pseudo-code:

```
USER_INFO info;

// Get the string from the user
getsUart0(&info);
// Echo back to the user of the TTY interface for testing
#ifdef DEBUG
putsUart0(info.buffer);
putcUart0(“\n”)
#endif

// Parse fields
parseFields(&info);
// Echo back the parsed field information (type and fields)
#ifdef DEBUG
uint8_t i;
for (i = 0; i < info.fieldCount; i++)
{
        putcUart0(info.fieldType[i]);
        putcUart0(“\t”);
        putsUart0(&info.buffer[fieldPosition[i]]);
        putcUart0(“\n”);
}
#endif

// Command evaluation
```

```
bool valid = false;
// set add, data → add and data are integers
if (isCommand(&info, “set”, 2))
{
        int32_t add = getFieldInteger(&info, 1);
        int32_t data = getFieldInteger(&info, 2);
        valid = true;
        // do something with this information
}
// alert ON|OFF → alert ON or alert OFF are the expected commands
if (isCommand(&info, “alert”, 1))
{
        char* str = getFieldString(&info, 1);
        valid = true;
        // process the string with your custom strcmp instruction, then do something
}

// Process other commands here

// Look for error
if (!valid)
        putsUart0(“Invalid command\n”);
```

Sample outputs:

If the user types “set 1, 2” followed by a carriage return, then the code should process the string as a set command with add=1 and data=2.

If the code above was run with DEBUG defined and the user inputs the string “sen<back_space>t 1, 2”, the output would be:

```
set 1, 2
a       set
n       1
n       2
```