The *getline()* Function

WEEK
2

Up to this point you have learned about a multitude of functions, most of which have been relatively small in size. Today you'll concentrate on a single function to add to your library: the `getline()` function. In the process, you'll gain the use of a few more functions. Today you will:

- ☐ See what is needed to get formatted data from the display screen.
- ☐ Become familiar with the `getline()` function.
- ☐ Learn what the `getline()` function can do.
- ☐ Learn how to use the `getline()` function.

Why Is an Entire Day Dedicated to a Single Function?

You have seen a number of functions up to this point. Many of them expand functions that you had in your basic compiler. For example, `write_char()` expanded functions such as `putch()` by giving you the ability to write a character in color. In addition, you gained the ability to place the cursor anywhere on the screen. With the functions that you have learned up to this point, you can design a text graphics screen to look any way you want.

What hasn't been covered is retrieving data off the text graphic screen. There are functions such as `gets()` and `scanf()` that can get data; however, they aren't suited for data entry within text graphic applications. The `getline()` function is being presented as a replacement for these functions.

This still doesn't explain why an entire day is needed to cover `getline()`. As you will see, getting data from the screen can become complex depending on the amount of functionality you build into your applications. The `getline()` function has been developed with as much functionality as possible. With the `getline()` function you will be able to do the following:

- ☐ Get a string or a number. If a number is being entered, characters won't be accepted.
- ☐ Use the right and left arrow keys to move within the entered field.
- ☐ Use the backspace key to erase the preceding character.
- ☐ Set up keys to be used to exit the entry. You'll also be able to know which key was used to exit.

- ☐ Insert characters in the middle of the string.
- ☐ Set up colors for the information being entered.
- ☐ And more.

To accomplish all of the capabilities listed requires a powerful and large function. In fact, the amount of code in the `getline()` function may be more than in some of the programs you've written up to this point.

An Overview of the *getline()* Function

The `getline()` function is best understood in steps. Figure 10.1 presents a breakdown of the `getline()` function.

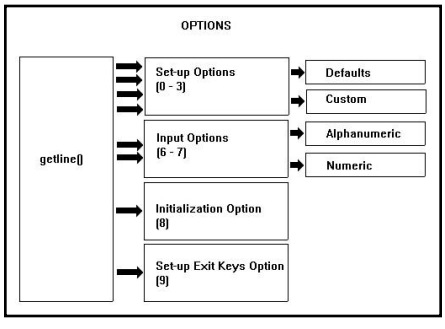

Figure 10.1. *The breakdown of the* `getline()` *function.*

As you can see, `getline()` breaks down into several groups of options. In using `getline()`, you'll need to use at least one of the set up options in order to set up the colors that will be used. In addition, you'll need to use the option to set up the exit keys. Once you have executed `getline()` with these options, you'll be ready to use the input options which allows either numerics or alphanumerics to be entered.

A New TYAC.H Header File

Before getting into the code and the individual parts of the `getline()` function, a new TYAC.H header file needs to be introduced. The `getline()` function needs several

new constants. It also needs a prototype for it and several of the sub-functions that it uses. Listing 10.1 presents a new TYAC.H header file which includes everything needed for `getline()`.

Listing 10.1. A new TYAC.H header file.

```
1: /* Program: TYAC.H
2:    *          (Teach Yourself Advanced C)
3:    * Authors: Bradle y L. Jones
4:    *          Gregory L. Guntle
5:    * Purpose: Header file for TYAC library functions
6:    *=========================================================*/
7:
8:   #ifndef _TYAC_H_
9:   #define _TYAC_H_
10:
11:  /* DOS and BIOS Interrupts */
12:  #define BIOS_VIDEO       0x10
13:  #define BIOS_KEY         0x16
14:  #define DOS_FUNCTION     0x21
15:
16:  /* BIOS function calls */
17:  #define SET_VIDEO        0x00
18:  #define SET_CURSOR_SIZE  0x01
19:  #define SET_CURSOR_POS   0x02
20:  #define GET_CURSOR_INFO  0x03
21:  #define WRITE_CHAR       0x09
22:  #define SET_COLOR        0x0B
23:  #define GET_VIDEO        0x0F
24:  #define WRITE_STRING     0x13
25:
26:  /* BIOS used to set scrolling direction */
27:  #define SCROLL_UP        0x07
28:  #define SCROLL_DOWN      0x06
29:
30:  /* DOS functions calls */
31:  #define GET_DATE         0x2A
32:
33:
34:  /*  Types of Boxes  */
35:  #define DOUBLE_BOX       1
36:  #define SINGLE_BOX       2
37:  #define BLANK_BOX        3
38:
39:  /*  Box fill flags  */
40:  #define BORDER_ONLY      0
41:  #define FILL_BOX         1
42:
43:  /* Colors */
44:  #define BLACK            0
```

```
45:  #define BLUE              1
46:  #define GREEN             2
47:  #define CYAN              3
48:  #define RED               4
49:  #define MAGENTA           5
50:  #define BROWN             6
51:  #define WHITE             7
52:  #define GRAY              8
53:  #define LIGHTBLUE         9
54:  #define LIGHTGREEN        10
55:  #define LIGHTCYAN         11
56:  #define LIGHTRED          12
57:  #define LIGHTMAGENTA      13
58:  #define YELLOW            14
59:  #define BRIGHTWHITE       15
60:
61:  #define BLANK             ' '
62:  #define SPACE             ' '
63:  #define NEWLINE           '\n'
64:  #define TAB               '\t'
65:  #ifndef NULL
66:     #define NULL           '\0'
67:  #endif
68:
69:  #define EOS               0
70:  #define YES               1
71:  #define NO                0
72:  #define TRUE              1
73:  #define FALSE             0
74:
75:  #define CR                13
76:  #define LF                10
77:  #define EOL               13
78:
79:  #define CLR_INS           1         /* Clear Inside Box Flag */
80:  #define NO_CLR            0         /* Don't Clear Inside the Box */
81:
82:  #define CTR_STR           1         /* Centering a string on the
                                            screen */
83:  #define NO_CTR_STR        0         /* Don't center the string */
84:
85:
86:  /* ------------------------------- *
87:   *       New Type Definitions      *
88:   * ------------------------------- */
89:
90:  typedef int BOOLEAN;
91:  typedef unsigned short PTR;         /*          0 -        65535 */
```

continues

Listing 10.1. continued

```
92:  typedef char CHAR;               /*          -128 -  127 */
93:  typedef unsigned char EXTCHAR;    /*             0 -  255 */
94:  typedef short NUM32K;             /*        -32768 - 32767 */
95:  typedef unsigned short NUM64K;    /*             0 - 65535 */
96:  typedef long NUM2GIGA;            /*-2,147,483,648 - 2,147,483,647 */
97:  typedef unsigned long NUM4GIGA;   /*             0 - 4,294,967,295 */
98:  typedef unsigned char ARRAY;       /* Used for creating an array */
99:
100:
101: /* ------------------------------ *
102:  *           KEYS                 *
103:  * ------------------------------ */
104:
105: /* Numeric keypad scan codes    */
106: #define  HOME       71        /*   home key    */
107: #define  UP_ARROW   72        /*   up arrow    */
108: #define  PAGE_UP    73        /*   page up     */
109: #define  LT_ARROW   75        /*   left arrow  */
110: #define  RT_ARROW   77        /*   right arrow */
111: #define  END        79        /*   end key     */
112: #define  DN_ARROW   80        /*   down arrow  */
113: #define  PAGE_DN    81        /*   page down   */
114: #define  INS        82        /*   insert      */
115: #define  DEL        83        /*   delete      */
116: #define  SHIFT_TAB  15        /*   shift tab   */
117: #define  ENTER_KEY  28
118:
119: /* Function key scan codes */
120: #define  F1         59        /*     F1 KEY    */
121: #define  F2         60        /*     F2 KEY    */
122: #define  F3         61        /*     F3 KEY    */
123: #define  F4         62        /*     F4 KEY    */
124: #define  F5         63        /*     F5 KEY    */
125: #define  F6         64        /*     F6 KEY    */
126: #define  F7         65        /*     F7 KEY    */
127: #define  F8         66        /*     F8 KEY    */
128: #define  F9         67        /*     F9 KEY    */
129: #define  F10        68        /*     F10 KEY   */
130:
131: /*  Other non scan keys as ASCII  */
132:
133: #define  BK_SP_KEY  8
134: #define  ESC_KEY    27
135: #define  CR_KEY     13
136: #define  TAB_KEY    9
137: #define  SPACE_BAR  32
138:
```

```
139: /* ---------------------------------- *
140:  *          Getline Options           *
141:  * ---------------------------------- */
142:  #define SET_DEFAULTS  0
143:  #define SET_NORMAL    1
144:  #define SET_UNDERLINE 2
145:  #define SET_INS       3
146:  #define GET_ALPHA     6
147:  #define GET_NUM       7
148:  #define CLEAR_FIELD   8
149:  #define SET_EXIT_KEYS 9
150:
151: /*------------------------*
152:     Function Prototypes
153:  *------------------------*/
154:      /* Gets the current date */
155: void current_date(int *, int *, int *);
156:
157:      /* Positions the cursor to row/col */
158: void cursor(int, int);
159:      /* Returns info about cursor */
160: void get_cursor(int *, int *, int *, int *, int *);
161:      /* Sets the size of the cursor */
162: void set_cursor_size(int, int);
163:
164:      /* clear the keyboard buffer */
165: void kbclear( void );
166:      /* determine keyboard hit */
167: int  kbhit( void );
168:
169:      /* scroll the screen */
170: void scroll( int row,   int col,
171:              int width, int hieght,
172:              int nbr,   int direction);
173:
174:      /* pause until ENTER pressed */
175: void pause(char *);
176:
177:      /* Video mode functions */
178: void set_video(int);
179: void get_video(int *, int *, int *);
180:
181:      /* Text Graphics functions */
182: void write_char(char, int, int);
183: void repeat_char(char, int, int, int);
184: void write_string(char *, int, int, int, int);
185: void box(int, int, int, int, int, int, int, int);
186: void set_border_color(int);
187: char getline(int, char, int, int, int, int, char * );
188:
```

continues

Listing 10.1. continued

```
189:      /* misc functions */
190: void boop( void );
191: void waitsec( double );
192: long get_timer_ticks( void );
193:
194: #endif
```

Analysis

You should review this header file to ensure that you are familiar with everything it contains. Lines 8, 9, and 194 used defined constants to prevent you from including the header file more than once. This is a common practice used by many programmers. If the constant `_TYAC_H_` isn't defined, then the header file hasn't been included before. You know this because line 9 defines this constant. If `_TYAC_H_` has been defined, the file skips to line 194.

Lines 11 through 83 define several different groups of constants. Lines 11 through 31 contain the defined constants for the BIOS functions that you worked with on previous days. Lines 27 and 28 contain the constants used with the scroll function from Day 8. Lines 35 to 41 contain constants used with the `box()` function that you created on Day 9. The colors are defined in lines 44 to 59. Various additional values are defined in lines 61 to 83. These are values that can be used by other functions that you create. This includes values for TRUE, FALSE, YES, NO, EOS (end of string), and more.

Lines 90 to 98 contain type definitions. Several new types have been declared that will be useful in your programs. Lines 106 to 137 declare type definitions for the keyboard keys. These values will be used when setting the exit keys for `getline()` which is covered later. Lines 142 to 149 contain the last of the defined constants. These constants can be used instead of numbers when calling `getline()`. The rest of the TYAC.H header file contains prototypes for the functions that should be in your TYAC.LIB library after you complete today's material.

Understanding the Use of *getline()*

Before presenting the code for the `getline()` function, the use of `getline()` needs to be explained. By covering the usage first, you'll find it much easier to understand the actual code.

The prototype for `getline()` is:

```
char getline( int option, char cParm1, int iParm1, int iParm2,
              int iParm3, int iParm4, char *char_ptr);
```

As you can see, the `getline()` function has several parameters. The most important parameter is the first, option. The *option* parameter determines exactly what `getline()` is going to do. `getline()` gains its functionality through the option parameter. Based on the option, the other parameters may or may not contain values. Because of the multiple uses of the rest of the parameters, they have been given generic names.

Note: When a `getline()` parameter isn't used, it's filled with 0.

The Option Parameter

There are eight different options that can be used when calling the `getline()` function. These options are based on a numeric value passed in option.

Options 0, 1, 2, and 3

If a value of 0, 1, 2, or 3 is passed to `getline()`, then different color values will be set. The colors that can be set in `getline()` are as follows:

`norm_color_fg`	The normal foreground color
`norm_color_bg`	The normal background color

The normal colors are used for the color of the text that is being entered. If you enter "BRADLEY", it will appear in the normal colors.

`high_color_fg`	The highlighted character's foreground color
`high_color_bg`	The highlighted character's background color

The highlighted character is the character that is currently being entered when the insert key is on. If the insert key is off, there won't be a highlighted character.

`undr_color_fg`	The underline character's foreground color
undr_color_bg	The underline character's background color

The `getline()` function has been written to display the extent of an enterable field. For example, if a string was to be entered that was 20 characters long, `getline()` would display 20 underline characters on the screen. The underline colors define the color of these underlines.

`ins_color_fg`	The "INS" message's foreground color
`ins_color_bg`	The "INS" message's background color

The `getline()` displays the characters "INS" in the lower-right corner of the screen. This insert message is toggled on and off with the insert key. The colors for this message are determined by the insert colors.

Each of the first four options set up different values. If you use option 1, you will set all of the colors to default values along with several other default values. Option 1 enables the normal and highlight colors to be set. Option 2 enables the underline colors to be set. Option 3 enables the insert message color, along with its position, to be set.

Options 4 and 5

The values 4 and 5 are not used as options. These two values are left open for future expansion of the `getline()` function.

Options 6 and 7

Options 6 and 7 are the options that allow `getline()` to get data. Option 6 accepts the input of alphanumeric values. Option 7 accepts the input of numeric values. Along with these options, you'll also need to specify screen coordinates for the input and the length of the value to be entered. These options will be detailed later.

Option 8

Using `getline()` with option 8 provides you with an alternative way to clear a string to nulls. By calling `getline()` with option 8 before calling it with option 6 or 7, you can ensure that no bad data is remaining in a variable that should be empty.

Option 9

This is the final option used with the `getline()` function. Option 9 enables you to set up the exit keys for `getline()`. An exit key is a key that enables you to leave the entry field. For example, the exit key for `gets()` is the enter key. The exit key returns control back to the program. Option 9 enables you to set up the keys that you want to exit the entry of a single field. Example of keys that are generally used for exiting are tab, enter, shift+tab, F1, and F3.

Other *getline()* Parameters

The values passed in the other parameters are dependent upon which option is called. The specific details of each parameter will be covered when `getline()` is analyzed in

the next section. Generally, the first parameter, `cParm1`, is rarely used. The next four parameters, `iParm1`, `iParm2`, `iParm3`, and `iParm4` are used to pass in numeric values. For the options that set colors, the `iParm#` parmeters contain color values. For options 6, 7, and 8, these variables contain the starting location and the length for the information to be entered. For option 9, the parameters contain the number of exit keys that are defined.

The last parameter in the prototype is `str_ptr`. This is used in options 6, 7, 8, and 9. In options 6 and 7, `str_ptr` contains the address of a character pointer. This address is where the information retrieved will be placed. For option 8, it should also be the location where data will be placed. Option 8 uses this pointer as the starting location to be cleared to nulls.

Option 9 uses the `str_ptr` differently. Option 9 will use the `str_ptr` as the pointer to the first `exit_key` in an array. These exit keys should be listed one after the other in a character array. The values used for the exit keys are those defined in the TYAC.H header file.

The Return Value of *getline()*

The return value of `getline()` is a character value. This value is the ASCII value of the last key pressed before exiting `getline()`. This value is one of the keys defined as a valid exit key with option 9.

The Code Behind *getline()*

Having an understanding of the options for `getline()` and the parameters prepares you for the function itself. Following is the complete `getline()` listing. It contains four different functions: `getline()`, `get_it()`, `check_list()`, and `setup_exit_keys()`. The `getline()` function uses `get_it()` with options 6 and 7 (getting data). The `setup_exit_keys()` function is used with option 9.

Note: The `getline()` function uses several of the functions presented earlier in this book. By keeping the `getline()` function in your TYAC.LIB library, you'll have access to these other functions. Additionally, you'll need a few more functions that you previously didn't have. The `getline()` function uses a function called `boop()` to make the computer beep. The `boop()` function in turn requires two other functions.

Type

Listing 10.2. GETLINE.C. The `getline()` function.

```
1:  /* Name   : getline.c
2:   * Authors: Bradley L. Jones
3:   *          Gregory L. Guntle
4:   *
5:   * Purpose: Receive string inputs from the user through
6:   *          the keyboard.  This routine replaces SCANF for
7:   *          reading keyboard responses from the user.
8:   *
9:   * Function  : getline(opt,ch1,int1,int2,int3,int4,str_ptr)
10:  *
11:  * Enter with: opt = One of the following:
12:  *
13:  *         0 - Set default parameters
14:  *             norm_color_fg = WHITE
15:  *             norm_color_bg = BLACK
16:  *             high_color_fg = BRIGHTWHITE
17:  *             high_color_bg = BLACK
18:  *             undr_color_fg = GREEN
19:  *             undr_color_bg = BLACK
20:  *             ins_color_fg  = YELLOW
21:  *             ins_color_bg  = BLACK
22:  *             ins_row       = 24
23:  *             ins_col       = 70
24:  *             stop_key      = CR_KEY
25:  *
26:  *         1 - Set highlight/normal colors for string
27:  *             int1 - foreground normal color
28:  *             int2 - background normal color
29:  *             int3 - foreground highlighted color
30:  *             int4 - background highlighted color
31:  *
32:  *         2 - Setup underline colors
33:  *             int1 - foreground color for underline character
34:  *             int2 - background color for underline character
35:  *             int3 - foreground color highlighting underline
36:  *             int4 - background color highlighting underline
37:  *
38:  *         3 - Setup INS message colors & row/col positioning
39:  *             int1 - foreground color for INS message
40:  *             int2 - background color for INS message
41:  *             int3 - Row where to display INS message
42:  *             int4 - Col where to display INS message
43:  *
44:  *         4 -
45:  *
46:  *         5 -
47:  *
48:  *         6 - Get alphanumeric input
```

```
49:   *              int1 - Upper left corner - row #
50:   *              int2 - Upper left corner - col #
51:   *              int4 - Max length of input
52:   *              str_ptr - Address for placing the chars
53:   *
54:   *          7 - Get numeric input
55:   *              int1 - Upper left corner - row #
56:   *              int2 - Upper left corner - col #
57:   *              int4 - Max length of input
58:   *              str_ptr - Address to store chars
59:   *
60:   *          8 - Clear char field value to NULLS
61:   *              int1 - Length of string for clearing
62:   *              str_ptr - Address of string to clear
63:   *
64:   *          9 - Clear and load valid exit keys
65:   *              The array is static and will remain
66:   *              until changed.
67:   *              There is no default value.
68:   *              int1 - # of keys in array
69:   *              str_ptr - Address of array (array name)
70:   *
71:   * Returns    : (applies to options 6,7 only )
72:   *              Char type of ascii value of the last
73:   *              key pressed that is within VALID_EXIT_KEYS.
74:   *
75:   * ------------------------------------------------------*/
76:
77:  #include <string.h>
78:  #include <conio.h>
79:  #include “tyac.h”
80:
81:  #define MAX_KEYS 17
82:
83:  /*--------------------------------*
84:   *   Global static variables      *
85:   *--------------------------------*/
86:
87:  static int  norm_color_fg;     /* foreground - normal color */
88:  static int  norm_color_bg;     /* background - normal color */
89:  static int  high_color_fg;     /* foreground - highlight */
90:  static int  high_color_bg;     /* background - highlight */
91:  static int  undr_color_fg;     /* foreground - underlines */
92:  static int  undr_color_bg;     /* background - underlines */
93:  static int  ins_color_fg;      /* foreground - INS message */
94:  static int  ins_color_bg;      /* background - INS message */
95:  static int  st_col, end_col;   /* constants     */
96:  static int  length;            /* string length */
97:  static int  row, col;
98:  static int  st_row;            /* constants     */
```

continues

Listing 10.2. continued

```
99:  static int  ins_row;          /* Row for INS message */
100: static int  ins_col;          /* Col for INS message */
101: static char stop_key;         /* key to stop accepting input */
102:
103:      /* valid exit keys loaded here */
104: static char VALID_EXIT_KEYS[MAX_KEYS];
105:
106: /*----------------------------------*
107:  *   Subroutines                    *
108:  *----------------------------------*/
109:
110:  char getline(int, char, int, int, int, int, char *);
111:  char get_it(int, char *);
112:  int  check_list( char );
113:  void setup_exit_keys( char *, int);
114:
115: /*-----------------------------------------*
116:  *           Start of Function             *
117:  *-----------------------------------------*/
118:
119: char getline(int option, char cParm1, int iParm1, int iParm2,
120:              int iParm3, int iParm4, char *char_ptr)
121: {
122:    int     ctr;       /* misc counter */
123:    char    last_key;  /* Holds last key pressed & returns it */
124:
125:    VALID_EXIT_KEYS[16] = '\0';    /* keep last key a null */
126:
127:    switch ( option )
128:    {
129:      case 0:  /* set default parameters */
130:               norm_color_fg=WHITE;
131:               norm_color_bg=BLACK;
132:               high_color_fg=BRIGHTWHITE;
133:               high_color_bg=BLACK;
134:               stop_key=CR_KEY;
135:               undr_color_fg=GREEN;
136:               undr_color_bg=BLACK;
137:               ins_color_fg= YELLOW;
138:               ins_color_bg= BLACK;
139:               ins_row = 24;
140:               ins_col = 70;
141:               break;
142:
143:      case 1:  /* set colors */
144:               norm_color_fg=iParm1;
145:               norm_color_bg=iParm2;
146:               high_color_fg=iParm3;
147:               high_color_bg=iParm4;
```

```
148:               break;
149:
150:       case 2:  /* insert normal and high colors */
151:               undr_color_fg=iParm1;
152:               undr_color_bg=iParm2;
153:               break;
154:
155:       case 3:  ins_color_fg = iParm1;
156:               ins_color_bg = iParm2;
157:               ins_row = iParm3;
158:               ins_col = iParm4;
159:               break;
160:
161:       case 4:  break;
162:
163:       case 5:  break;
164:
165:       case 6:  row = st_row = iParm1;
166:               col = st_col = iParm2;
167:               end_col = st_col + iParm4;
168:               length = iParm4;
169:               last_key = get_it(option,char_ptr);
170:               break;
171:
172:       case 7:  row = st_row = iParm1;
173:               col = st_col = iParm2;
174:               end_col=st_col + iParm4;
175:               length=iParm4;
176:               last_key = get_it(option,char_ptr);
177:               break;
178:
179:       case 8:  for (ctr=0; ctr < iParm1; ctr++)
180:                   char_ptr[ctr] = '\0';
181:               break;
182:
183:       case 9:  setup_exit_keys(char_ptr,iParm1);
184:               break;
185:
186:    }       /* end of switch */
187:
188:    return(last_key);
189:
190: }     /* end of subroutine */
191:
192:
193:
194: /*--------------------------------------*
195:  *  subroutine:   get_it()              *
196:  *                                      *
197:  *  this actually gets the data once    *
```

continues

Listing 10.2. continued

```
198:  *  everything has been setup           *
199:  *--------------------------------------*/
200:
201: char get_it(int option, char *str_ptr)
202: {
203:    int ins_pos;
204:    int ch;
205:    int str_ctr=0;       /* tracks current character position */
206:    int ins_on=FALSE;    /* tracks INS key being pressed */
207:    int prn_switch=FALSE;/* determines if char should be accepted */
208:    int loop_exit=FALSE;
209:    int test;
210:
211: /* ------------------------------------------------------ */
212:
213:    cursor(st_row,st_col);
214:    repeat_char('_', length, undr_color_fg, undr_color_bg);
215:    write_string(str_ptr, norm_color_fg, norm_color_bg,
216:                                         st_row, st_col);
217:
218:    while (loop_exit == FALSE)
219:    {
220:      if ( ( ch=getch() ) == 0 )   /* if scan code read next byte */
221:      {
222:        ch = getch();
223:        switch ( ch )
224:         {
225:            case HOME: /* goto to begining of string */
226:                       col = st_col;
227:                       cursor(row, col);
228:                       if (ins_on == TRUE)
229:                          write_string(str_ptr, norm_color_fg,
230:                                       norm_color_bg, st_row,
231:                                       st_col);
232:                       break;
233:
234:            case END:  /*  end key  - pos cursor at end */
235:                       col = strlen(str_ptr) + st_col;
236:                       cursor(row, col);
237:                       if (ins_on == TRUE)
238:                          write_string(str_ptr, norm_color_fg,
239:                                       norm_color_bg, st_row,
240:                                       st_col);
241:                       break;
242:
243:            case DEL:  /* 1 past end of string ? */
244:                       if ( col != strlen(str_ptr) + st_col)
245:                       {
246:                            /* save current position  */
```

```
247:                        str_ctr = col;
248:                           /* if nxt pos is != null move it */
249:                        while (str_ptr[col-st_col+1] != '\0')
250:                        {
251:                              /* the value is moved over */
252:                           str_ptr[col-st_col] = str_ptr
                                 [col-st_col+1];
253:                           col++;       /* next position  */
254:                        }
255:                           /* terminate end of string */
256:                        str_ptr[col-st_col] = '\0';
257:                           /* reprint string */
258:                        write_string(str_ptr, norm_color_fg,
259:                                     norm_color_bg, st_row,
260:                                     st_col);
261:                           /* Go to end of line  */
262:                        cursor(row, st_col+strlen(str_ptr));
263:                           /* Rewrite underline char */
264:                        write_char('_', undr_color_fg,
                             undr_color_bg);
265:                           /* restore cursor pos */
266:                        col = str_ctr;
267:                           /* Restore cursor position */
268:                        cursor(row,col);
269:                     }
270:                     if (ins_on == TRUE)
271:                         write_string(str_ptr,norm_color_fg,
272:                                      norm_color_bg, st_row,
273:                                      st_col);
274:                     break;
275:
276:            case INS:  if (ins_on == FALSE )
277:                     {
278:                        write_string("INS", ins_color_fg,
279:                                     ins_color_bg, ins_row,
280:                                     ins_col);
281:                        ins_on=TRUE;
282:                     }
283:                     else
284:                     {
285:                        write_string("   ", ins_color_fg,
286:                                     ins_color_bg, ins_row,
287:                                     ins_col);
288:                        write_string(str_ptr, norm_color_fg,
289:                                     norm_color_bg, st_row,
290:                                     st_col);
291:                        ins_on=FALSE;
292:                     }
293:                     break;
294:
```

continues

Listing 10.2. continued

```
295:            case LT_ARROW: if (col > st_col )
296:                           {
297:                             cursor(row, --col);
298:                             if (ins_on == TRUE)
299:                               write_string(str_ptr, norm_color_fg,
300:                                            norm_color_bg, st_row,
301:                                            st_col);
302:                           }
303:                           break;
304:
305:            case RT_ARROW: if ( col < end_col &&
306:                              ((col-st_col) < strlen(str_ptr)) )
307:                           {
308:                             cursor(row, ++col);
309:                             if (ins_on == TRUE)
310:                               write_string(str_ptr, norm_color_fg,
311:                                            norm_color_bg, st_row,
312:                                            st_col);
313:                           }
314:                           break;
315:
316:            default:   loop_exit = check_list(ch);
317:                       if (ins_on == TRUE)
318:                          write_string(str_ptr, norm_color_fg,
319:                                       norm_color_bg, st_row,
320:                                       st_col);
321:                          /* key a valid exit key ?*/
322:                       if ( loop_exit == FALSE )
323:                          boop();
324:                       break;
325:
326:          }         /* end of switch  */
327:      }       /* end of if  */
328:      else
329:      {
330:          switch ( ch )          /* test for other special keys  */
331:          {
332:             case BK_SP_KEY:
333:                       if (col > st_col )
334:                       {
335:                            /* move cursor left 1 */
336:                         cursor(row, --col);
337:                            /* save cur curs pos, len determ ltr */
338:                         str_ctr = col;
339:                           /* if next pos != null move it ovr */
340:                         while (str_ptr[col-st_col+1] != '\0')
341:                         {
342:                              /* move next char over  */
```

```
343:                         str_ptr[col-st_col] = str_ptr
                                [col-st_col+1];
344:                          /* next position  */
345:                       col++;
346:                     }
347:                        /* End string with a null */
348:                     str_ptr[col-st_col] = '\0';
349:                     write_string(str_ptr, norm_color_fg,
350:                                  norm_color_bg, st_row,
351:                                  st_col);
352:                        /* Move cursor to end of line */
353:                     cursor(row, st_col+strlen(str_ptr));
354:                        /* Rewrite underline char */
355:                     write_char('_', undr_color_fg,
                           undr_color_bg);
356:                        /* restore current cusor pos */
357:                     col = str_ctr;
358:                        /* Restore cursor position */
359:                     cursor(row,col);
360:                   }
361:                   break;
362:
363:           default:  if (col < end_col )
364:                   {
365:                        /* get numeric input */
366:                     if ( option == 7  &&  (ch >= 48 &&
                           ch <= 57))
367:                     {
368:                        prn_switch=TRUE;
369:                     }
370:                        /* greater than space */
371:                     if ( option == 6  && ch > 31 )
372:                     {
373:                        /* get alphanumeric input */
374:                        prn_switch=TRUE;
375:                     }
376:                     if ( prn_switch==TRUE )
377:                     {
378:                          /* field not full = shift */
379:                       if(ins_on==TRUE && strlen(str_ptr)
                             <length)
380:                       {
381:                           /* assign str_ctr to the cur
                                 location */
382:                         str_ctr=strlen(str_ptr);
383:                         while (str_ctr !=col-st_col)
384:                         {
385:                            str_ptr[str_ctr] = str_ptr
                                  [str_ctr-1];
```

continues

Listing 10.2. continued

```
386:                                    /* point to previous position  */
387:                                 str_ctr--;
388:                               }
389:                                  /* add character into string */
390:                               str_ptr[col-st_col]=ch;
391:                               write_string(str_ptr, norm_color_fg,
392:                                            norm_color_bg, st_row,
393:                                            st_col);
394:                               write_char((char)ch, high_color_fg,
395:                                                    high_color_bg);
396:                               cursor(row, ++col);
397:                               prn_switch=FALSE;
398:                             }      /* end of ins_on and strlen test */
399:                             else
400:                             {
401:                                  /* INS off put a char */
402:                               if (ins_on==FALSE )
403:                               {
404:                                  /* add character into string */
405:                                  str_ptr[col-st_col]=ch;
406:                                  write_string(str_ptr, norm_color_fg,
407:                                               norm_color_bg, st_row,
408:                                               st_col);
409:                                  cursor(row, ++col);
410:                                  prn_switch=FALSE;
411:                               }
412:                               else
413:                               {
414:                                  /* ins_on is TRUE and trying */
415:                                  /* to put a char past end    */
416:                                  boop();
417:                               }
418:                             }
419:                           }     /* end of prn == TRUE test */
420:                           else
421:                           {
422:                                /* exit key? */
423:                             if((loop_exit = check_list(ch)) == FALSE)
424:                             {
425:                                /* not a valid exit key */
426:                                boop();
427:                             }
428:                             else
429:                             {
430:                               write_string(str_ptr, norm_color_fg,
431:                                            norm_color_bg, st_row,
432:                                            st_col);
433:                             }
```

```
434:                         }
435:                       }      /* end of if from (col < end_col )  */
436:                       else     /* from ( col < end_col ) */
437:                       {
438:                            /* exit key? */
439:                         if ( (loop_exit = check_list(ch)) == FALSE )
440:                         {
441:                            /* not a valid exit key */
442:                            boop();
443:                         }
444:                         else
445:                         {
446:                            write_string(str_ptr, norm_color_fg,
447:                                         norm_color_bg, st_row,
448:                                         st_col);
449:                         }
450:                       }
451:                       break;
452:
453:          }   /* end of switch */
454:       }     /* end of else */
455:    }       /* end of while loop */
456:
457:    return(ch);
458:
459: }  /* end of subroutine getline  */
460:
461: /* ------------------------------------------  *
462:  * function: check_list()                      *
463:  *                                             *
464:  * This subroutine checks the key pressed against  *
465:  * a list of keys that can end the procedure.      *
466:  * It receives the key pressed and returns TRUE    *
467:  * if key is in the list, else FALSE if not in     *
468:  * list.                                       *
469:  * ------------------------------------------  */
470:
471: int check_list(char key_pressed)
472: {
473:       /* return a true or false to return_code */
474:   int return_code=FALSE;
475:   int loop_ctr = 0;
476:
477:   while ( loop_ctr <= MAX_KEYS && !return_code)
478:     if ( key_pressed == VALID_EXIT_KEYS[loop_ctr++])
479:        return_code=TRUE;
480:
481:   return(return_code);
```

continues

Listing 10.2. continued

```
482: }
483:
484: /* ------------------------------------------- *
485:  * function: setup_exit_keys(keys_array,num)    *
486:  *                                             *
487:  * Sets up valid exit keys in the VALID_EXIT_KEYS *
488:  * array.                                      *
489:  *                                             *
490:  * Enter with: - keys_array                    *
491:  *                char array of ASCII key values *
492:  *             - num                           *
493:  *                nbr of elements to processed *
494:  * Returns:     Nothing                        *
495:  * ------------------------------------------- */
496:
497: void setup_exit_keys(char *keys_array, int num)
498: {
499:   int ctr;                    /* misc counter */
500:
501:   for (ctr=0; ctr < num; ctr++)
502:   {
503:        /* load valid keys */
504:     VALID_EXIT_KEYS[ctr] = *(keys_array + ctr);
505:   }
506:
507:   while (ctr < MAX_KEYS)
508:   {
509:       /* clear unused portion */
510:     VALID_EXIT_KEYS[ctr++] = '\0';
511:   }
512: }
```

Analysis As you can see, this is an extremely long function. Having read the material presented earlier today, you should be able to follow some of this listing. To help in your understanding of `getline()`, the function contains many comments. In fact, the first 75 lines of the function are dedicated to a detailed description of the parameters.If you haven't already, then you should read these comments. They include a description of what each of the parameters that is passed to `getline()` should be.

Lines 87 to 104 contain variables that will be used by `getline()`. These variables are all defined as `static`. This is so their values will be retained for subsequent calls to `getline()`. The comments within the code state what each variable is used for.

Note: If you want the default values automatically set in `getline()`, you should assign the default values to the variables as they are declared. For example, line 87 would become:

```
static int norm_color_fg = WHITE;
```

Lines 110 to 113 are the last of the set-up before starting the `getline()` function. These lines declare the prototypes to the subroutines used by `getline()`. As you can see, the `getline()` function has three subroutines or functions that it uses.

Lines 119 to 190 contain `getline()`. This portion of the `getline()` process is straightforward. Line 125 ensures that the array that contains the exit keys (or will contain them if they aren't yet set up) ends with a null value. Line 127 then calls a switch statement. The program switches based on the option that was passed.

If the option was zero, the defaults will be set. The defaults include all of the colors, an exit key (line 134), and a position for the "INS" message (lines 139 and 140). The colors that have been set here—and that are stated in the comments in lines 14 to 24—are the defaults that I have choosen. The values you choose for your default values should be those that you will use most often. You can always change these values using `getline()`'s other options.

Options 1, 2, and 3 are set in lines 143 to 159. These options set different sets of the variables. By looking at each of these options, you'll see which parameters are translated to which variables.

Options 4 and 5 in lines 161 and 163 don't exist. These are left for future growth. If you later decide to expand on `getline()`, these two values are available for options.

Options 6 and 7 in lines 165 to 170 and 172 to 177 are identical. These functions each set a row and column value to the `iParm1` and `iParm2` parameters. Also set are `st_row` and `st_col` which are static constants used to retain the initial row and column positions. Lines 167 and 174 calcuate the ending column, `end_col`, of the information being input. Each of these two cases ends with a call to `get_it()` which does the work of retrieving the input information for `getline()`.

Option 8 is covered in lines 179 to 181. This case is easy to follow. A `for` loop is used to set each position of the passed string, `char_ptr`, to null values.

Option 9 in lines 183 and 184 is the final option. This option simply calls the `setup_exit_keys()` subroutine which is covered later today.

The *get_it()* Subroutine

The `get_it()` function is used to get both numeric and alphanumeric values. This function continues from line 201 through line 459. Although this function is very long, it's easy to follow because it's broken into segments by `case` statements.

Before starting into its main loop, the `get_it()` function sets up a few keys. In addition, line 213 sets the cursor to the starting position that was set in the `getline()` function. Line 214 then sets the underscore on the screen using the `repeat_char()` function. Line 215 writes the value in the string that may have been initially passed to `getline()` and forwarded to `get_it()`. Line 218 then begins a large `while` loop.

The `while` loop in line 218 begins the process of getting each character one at a time. Line 220 checks to see if the first character retrieved with `getch()` is a 0. If it is, the key entered is a scan code. A *scan code* is part of an extended key such as the home key, the end key, or the delete key. If a scan code is read, a second key is read to get the second half of the scan code. The second character contains a key value that is used in a `switch` statement in lines 225 to 326. The functions for each of the different keys is detailed later today.

If the initial character read in line 222 was not equal to 0, then the `else` statement in line 328 is executed. In this case, the character is a normal ASCII character. Included with the ASCII characters are characters such as the backspace key. Any ASCII characters that need special processing are checked first. For `getline()`, only the backspace character, `BK_SP_KEY`, needs to be handled specially. (All the other special exit keys are scan codes handled by the `if` in line 220.)

Line 363 is the default case for ASCII characters. It's here that `getline()` will determine whether the appropriate key has been entered.

Getting the Characters (Lines 363 to 451)

In lines 366 to 375, the character entered is compared with ASCII values to determine if it is valid based on the `getline()` option. For option 7, the character must be an ASCII value from 48 to 57 (line 366). For option 6, the character entered must be greater than 31. If the character read fits either of these options, then the `prn_switch` is set to `TRUE`.

If the character passed (the `prn_switch` was set to `TRUE`), then lines 364 to 435 are executed. Line 379 then checks to see if the insert key is on. If it is, and if the length of the string is less than the total length of the field being entered, then the character is added to the string. Because the character could be in the middle of the string, the

rest of the string is adjusted to the right (lines 383 to 388). Line 391 then redisplays the updated string to ensure that it's displayed on the screen properly. Because the insert key is on, the added character should be highlighted. This is done in line 394 before the cursor is adjusted. If the insert key was off, or if the character is being set in a field that is already full, then the `else` in lines 399 to 417 is executed. If the insert key is off, then the character is added to the string at the current position (line 405), the string is rewritten to ensure that it is displayed properly (line 406), and the cursor is repositioned (line 409). If the insert key is on and the string is full, then the computer beeps with the `boop()` function, which is covered later today.

If the character entered didn't meet the valid characters for options 6 or 7, then the `else` statement in line 420 is executed. In this `else`, the character entered is checked to see if it is actually an exit key. This is accomplished by using the `check_list()` function. The `check_list()` function in lines 461 to 482 simply loops through the exit key array to see if a match is found. If a match is found, a code of `TRUE` is returned in line 481. If the key isn't a valid exit key, the value of `FALSE` is returned. This value of FALSE causes the `get_it()` function in line 426 to execute `boop()`, which beeps the computer. This is done because the character entered wasn't valid for the option, nor was it an exit key. If the key was a valid exit key, then line 430 reprints the string to the screen, and the `loop_exit` causes the looping to end along with `getline()`.

The Backspace Character (Lines 332 to 361)

The backspace character is a special case that is handled in lines 332 to 361. If the current position isn't the first position, then the code for the backspace character is entered in lines 335 to 359. The code in these lines starts by moving the cursor to the left one column (line 336). It then shifts each character one space to the left to effectively delete the character that was backspaced over. Line 349 then rewrites the string to the screen to ensure that it is displayed correctly. Because this moving of the characters to the left will mess up the underlines that mark the end of the field, lines 353 to 356 redraw them.

The Delete Key (Lines 243 to 274)

The delete key (DEL) works nearly identically to the backspace character. The main difference is the cursor isn't moved to the left like the backspace. Instead, the function shifts the characters starting to the right of the cursor. Each is shifted one space to the left. The string is then rewritten, and the underscores for the field redrawn.

The Home Key (Lines 225 to 232)

The home key (HOME) is much easier to follow than the delete or backspace key. The home key simply adjusts the cursor position to the starting column. If the insert character is on, the string is redrawn to ensure it is displayed correctly.

The End Key (Lines 234 to 241)

The end key (END) works just like the home key. Instead of moving the cursor to the beginning of the string, it is moved to the end of the entered characters.

The Insert Key (Lines 276 to 293)

The insert key (INS) is different from the others. If the insert key wasn't already on, `ins_on` is equal to `FALSE`, then the "INS" string is written in the lower-right corner of the string and `ins_on` is set to `TRUE`. If the insert key was on, then a blank string is written over the "INS" that is in the lower-right corner and the flag is set to FALSE. This flag was used when entering a key earlier.

The Left Arrow (Lines 295 to 303)

If the cursor isn't already at the beginning of the field being entered, the left arrow key (LT_ARROW) executes lines 297 to 302. The left arrow key adjusts the cursor by subtracting one from the column, and then redisplaying it. This, in effect, moves the cursor one space to the left. If the insert key was on, then the string is redisplayed to ensure that it is correct on the screen.

The Right Arrow (Lines 305 to 314)

The right arrow (RT_ARROW) does just the opposite of the left arrow. In line 305, it first checks to see if the cursor is already at the end of the string. If it isn't, then one is added to the column, and the cursor is redisplayed. This has the effect of moving the cursor one space to the right.

The Default Scan Key

If the scan key entered wasn't one of the designated keys, then the default case is executed in lines 316 to 324. The default case checks to see if the entered key is an exit key (line 316). If it is, then the `loop_exit` flag is set. After this test, the string is redisplayed in line 318 to ensure that it is properly presented on the screen. If an exit key wasn't entered in this default case, then `boop()` beeps the computer in line 323 to signal that a bad key was entered.

The *setup_exit_keys()* Subroutine

The `setup_exit_keys()` function is all that is left to the `getline()` function. This function initializes the `VALID_EXIT_KEYS` array to the keys provided by the calling program. Each key in the array passed by the calling program is placed in the array. Any array positions that aren't used are then filled with null values (lines 507 to 511).

The *boop()* Function

The `boop()` function is used by `getline()` to cause the computer's speaker to beep. This function, which is presented in Listing 10.3, is also useful at other times, and therefore, makes a good addition to your TYAC.LIB library.

Type

Listing 10.3 BOOP.C. The `boop()` function.

```
1:    * Program: boop.c
2:    * Authors: Bradley L. Jones
3:    *          Gregory L. Guntle
4:    *
5:    * Purpose: Toggles the speaker to produce a sound.  This
6:    *          sound is used for notifying the user of an
7:    *          invalid key pressed.
8:    *
9:    * Note:    This is not an ANSI compatible function. When
10:   *          compiled, you may receive warnings. The value
11:   *          of result is not used; however, it is needed
12:   *          in order to compiler on some computers.
13:   *-----------------------------------------------------------*/
14:
15:  #include <conio.h>
16:  #include "tyac.h"
17:
18:  #define CLOCKFREQ    1193180L          /* Timer frequency */
19:  #define SPKRMODE     0xB6              /* Set timer for speaker */
20:  #define T_MODEPORT   0x43              /* Timer-mode port */
21:  #define FREQPORT     0x42              /* Frequency control port */
22:  #define FREQ0        0x12c             /* A frequency */
23:  #define DIV0         CLOCKFREQ / FREQ0 /* Set frequency to use */
24:  #define CLICK        .15               /* Tone duration */
25:
26:  #define SPKRPORT     0x61              /* Speaker port */
27:  #define SPKRON       0x03              /* On bits for speaker */
28:
29:
30:  void boop()
```

continues

Listing 10.3. continued

```
31:  {
32:     unsigned char port0;
33:     unsigned int div0  = DIV0*2;
34:     float delay = CLICK;
35:     int result;
36:
37:     result = outp(T_MODEPORT, SPKRMODE);  /* setup timer */
38:     port0 = inp(SPKRPORT);                /* get old port setting */
39:
40:     result = outp(FREQPORT, (div0 & 0xFF));    /* send low byte */
41:     result = outp(FREQPORT, (div0 >> 8));      /* send high byte */
42:     result = outp(SPKRPORT, (port0 | SPKRON)); /* turn on speaker */
43:
44:     waitsec(delay);                          /* wait */
45:
46:     result = outp(SPKRPORT, port0);   /* restore original setting */
47:  }
```

Analysis

This function uses the `outp()` and `inp()` functions to send information to the speaker port. These aren't ANSI-compatible functions. Because of this, the listing may not be compatible with all compilers.

Lines 18 to 27 define several constants that are then used in the actual code. The `boop()` function creates a beep by writing information directly to the speaker port. Before doing so, line 38 saves the original port setting. Line 46 then restores the settings.

Lines 41 and 42 send the values to the port. Line 42 then turns on the speaker. The speaker will then remain on until turned off. The speaker is turned off when the original setting is restored. To allow the beep to last long enough to be heard, the program is paused using the `waitsec()` function in line 44. The `waitsec()` function is a new function that is covered next.

The *waitsec()* Function

The `waitsec()` function causes the computer to pause for a specified period of time, which is defined in seconds. Most people choose to use a looping function to pause the computer. This may be coded as follows:

```
for( ctr = 0; ctr < 10000; ctr++ ) { /* pausing */ };
```

This will cause the computer to pause for different lengths of time depending on how fast the computer can process the `for` loop. This can cause a problem because you can

never be sure how long the loop will last. The `waitsec()` function gets around this problem. Listing 10.4 presents the `waitsec()` function.

Type

Listing 10.4. WAITSEC.C. The `waitsec()` function.

```
1:  /* Program:  waitsec.c
2:   * Authors:  Bradley L. Jones
3:   *           Gregory L. Guntle
4:   *
5:   * Purpose:  Causes the program to wait a number of seconds.
6:   *
7:   * Enter with: seconds - Number of seconds to pause program.
8:   *
9:   * Returns   : N/A
10:  * ------------------------------------------------------------ */
11:
12: void waitsec( double secs )
13: {
14:    unsigned long count0, count;
15:
16:    count0 = get_timer_ticks();
17:    count = count0 + secs * 18.2;
18:    while ( get_timer_ticks() < count );
19:
20: }
```

Analysis

The `waitsec()` function uses the computer's timer to determine exactly how much time has passed. The function starts by getting the number of timer ticks using the `get_timer_ticks()` function. Listing 10.5 will present the `get_timer_ticks()` function. Once the timer tick count is obtained, it is used as a base to determine at what number the time will be up. In a second, 18.2 ticks will occur. By taking 18.2 times the number of seconds requested to wait, you determine the total number of ticks that must occur. This calculated number is added to the original number that was received by the call to `get_timer_ticks()` (line 17). The program is then put into a `while` loop that continuously calls the `get_timer_ticks()` function until the returned value is a number greater than the number that was calculated. Once the appropriate number of ticks has passed, the function returns.

The *get_timer_ticks()* Function

The `get_timer_ticks()` function is a new BIOS function. As stated in the analysis of the `waitsec()` function, the `get_timer_ticks()` function simply returns the computer's current tick counter value. Listing 10.5 presents this function.

Type

Listing 10.5. GETTICKS.C. The `get_timer_ticks()` function.

```
1:  /* Program : getticks.c
2:   * Authors : Bradley L. Jones
3:   *           Gregory L. Guntle
4:   *
5:   * Purpose   : Returns the number of clock ticks.
6:   *
7:   * Function  : get_time_ticks()
8:   *
9:   * Enter with: N/A
10:  *
11:  * Returns   : Number of clock ticks that has elapsed.
12:  *             This is a long value.
13:  * ------------------------------------------------------- */
14:
15: #include <dos.h>
16:
17: #define INT_TIME  0x1A
18:
19: long get_timer_ticks()
20: {
21:    union REGS inregs;
22:    long tc;
23:
24:    inregs.h.ah = 0;
25:    int86(INT_TIME, &inregs, &inregs);
26:    tc = ((long) inregs.x.cx) << 16;  /* get high bytes */
27:    tc += inregs.x.dx;                /* add low bytes */
28:    return(tc);
29: }
```

Analysis

There isn't a lot to analyze about this listing. Interrupt 0x1Ah is used to get the current clock tick count. The values returned in the `x.cx` and `x.dx` registers are used to determine the exact number of ticks. Line 28 then returns this value.

Note: The new functions that you have created today should be added to your TYAC.LIB library. This library will be used in creating the programs throughout the rest of this book.

Using *getline()*

Now, you are ready to use `getline()` in a program. Listing 10.6 presents a very simple program that uses the `getline()` function. This program in Listing 10.6 will allow a string to be entered.

Listing 10.6. GL_TEST.C using the `getline()` function.

```
1:  /* Program:  gl_test.c
2:   * Author:   Bradley L. Jones
3:   *           Gregory L. Guntle
4:   * Purpose:  Demonstrate the getline function.
5:   *=========================================================*/
6:
7:  #include <stdio.h>
8:  #include "tyac.h"
9:
10: int main()
11: {
12:     char ch;
13:     char strng[40];
14:     int i;
15:     char exit_keys[] = {ESC_KEY, F1, F10, CR_KEY};
16:
17:          /* Initialize getline w/defaults */
18:      ch = getline(0,0,0,0,0,0,0);
19:          /* Clear the array to hold input */
20:      ch = getline(8,0,40,0,0,0,strng);
21:          /* Load valid exit keys */
22:      ch = getline(9,0,4,0,0,0,exit_keys);
23:
24:      write_string("Enter string:", LIGHTBLUE, BLACK, 10, 5 );
25:      ch = getline(6,0,10,20,0,20,strng);  /* Get line */
26:
27:      printf("\n\nThe string that was entered = %s\n",strng);
28:      printf("The key used to exit getline is: ");
29:      switch( ch )
30:      {
31:          case ESC_KEY:  printf("Esc key\n");
32:                         break;
33:          case F1:       printf("F1 key \n");
34:                         break;
35:          case F10:      printf("F10 key\n");
36:                         break;
37:          case CR_KEY:   printf("CR key\n");
38:                         break;
39:          default:       printf("Unknown\n");
40:                         break;
```

continues

Listing 10.6. continued

```
41:         }
42:
43:       return 0;
44:   }
```

Output

```
C:\>gl_test


    Enter string:  _____________________
```

Analysis

This is a short program that does a lot of work by using the `getline()` function. It's a good program for showing just how the `getline()` function and its options should be used. Lines 7 and 8 include the appropriate header files. The TYAC.H header file should be the same as the one presented in Listing 10.1.

The main part of the program starts in line 10 where several variables are declared. In line 15, the character array, `exit_keys` is declared and initialized to four keys. The constants defined in the TYAC.H header file are used as the values for the exit keys. In your programs, you should create a similar character array that contains the keys that will stop entry of information. From line 15, you can see that the escape key, the F1 function key, the F10 function key, and the carriage-return (or enter) key will all stop entry of information.

Line 18 presents the first call to `getline()`. The first parameter is the option parameter. In the case of line 18, option 0 is being called. Option 0 sets the default colors and values for `getline()`. Line 20 calls `getline()` a second time. In Line 20, option 8 is used. Option 8 clears the field passed in the last parameter to null values. In this case, the `strng` character array is being set to nulls to ensure that there is no bad data in it. It's a good practice to always initialize your data fields so that you are sure what is in them.

Line 22 calls the `getline()` function a third time. With this call, option 9 is used. This sets up the option keys that were defined in the last parameter, `exit_keys`. Once this

call is made, you have set up `getline()` for the rest of the program. In this case, that is only one more call to `getline()`; however, generally, you will be calling `getline()` with options 6 or 7 several times after these initial setup calls.

Line 24 prints a prompt on the screen so that the user will know what to enter. This prompt can be seen in the output. Line 25 then calls `getline()` with option 6. Option 6 enables the user to enter a string. In this case, the string will be displayed at row 10 and column 20. The string can be up to 20 characters long. It will be stored in `strng`. All this was stated in the call to `getline()` on line 25.

With the call in line 25, `getline()` does its job of enabling the user to input data. If the insert key is pressed, then the "INS" message will be toggled on and off. In addition, if insert is on, then the inserted character will be highlighted. Once the user presses one of the exit keys, `getline()` returns control to the key pressed. The rest of this program prints what was entered. In addition, lines 29 to 41 display which exit key was used to exit.

You should take time to play with this program and the other `getline()` options. Practice setting up different colors and different exit keys. The `getline()` function will be a critical function in creating the applications later in this book.

DO	DON'T
DO understand the `getline()` function. It will be used intensively on Day 13.	
DO initialize data fields if you are unsure what is in them. This way you can be certain.	
	DON'T forget to set up your exit keys when using the `getline()` function. You need to define what values can be used to exit the function.

Summary

Today's materials present a function that will replace `gets()` and `scanf()` in getting data from the screen. This function, called `getline()`, will provide you with much more functionality than the functions generally provided. The `getline()` function will allow text or numeric information to be entered. In addition it will allow for color and cursor placement. In addition to the `getline()` function, several other functions

are presented. These functions are `boop()`, `waitsec()`, and `get_timer_ticks()`. These functions are used by `getline()` and can also be used by your other functions.

Q&A

Q Why can't `gets()` be used instead of `getline()` for reading character strings?

A `gets()` does not enable you to have control over the color or position of the text being entered. You could use a function such as `cursor()` to place the prompt in the correct location before reading; however, you'll still have problems. The additional problem is that `gets()` won't limit the length of the string that you are reading. If you have a last name field that is only 15 characters, `getline()` will enable you to read only 15 characters. This is not true with `gets()`.

Q Why can't `scanf()` be used instead of `getline()` for reading character strings?

A `scanf()` is another good function to use; however, like `gets()`, it isn't as full-featured as `getline()`. (See the answer to the previous question.)

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. What is the purpose of `getline()`?
2. What does `boop()` do?
3. What is a reason for using `boop()`?
4. What is the advantage of `waitsec()` over the `pause()` function that you learned on Day 9?
5. How do you set the `getline()` input text color to yellow on red?

6. What are the default colors for `getline()`?
7. What is the difference between `getline()`'s option 6 and option 7?

Exercises

1. Add the functions that you created today to your TYAC.LIB if you have not done so already. The functions from today are:

```
getline()

boop()

waitsec()

get_timer_ticks()
```


Note: `get_it()` and `setup_exit_keys()` are a part of `getline()` so they don't need to be added on their own.

2. **BUG BUSTER:** What, if anything, is wrong with the following:

```
#include <stdio.h>
#include "tyac.h"

int main()
{
    char ch;
    char strng[40];

     ch = getline(0,0,0,0,0,0,0);
     ch = getline(8,0,40,0,0,0,strng);
     write_string("Last name:", LIGHTBLUE, BLACK, 10, 8 );
     ch = getline(6,0,10,20,0,20,strng);  /* Get line */

     return 0;
}
```

3. Use the functions that you have learned to create a box on the screen. In the box, display a message and ask the user to enter Y or N. Use the `getline()` function to get the Y or N.
4. Modify Exercise 3 to beep if a wrong character is entered.
5. **ON YOUR OWN:** Use the `getline()` function to create a data entry screen. Use the functions that you have used in the previous chapters also.

Note: Day 13 does just this! It uses `getline()` and most of the other functions presented so far to create an entry and edit screen.