Application Note

5x7 LED Matrix Display with Z8 Encore! XP®

AN014402–1207

Abstract

This application note explains the method to use Zilog's Z8 Encore! XP® microcontroller's General-Purpose Input/Output (GPIO) features to control the display of characters on a 5x7 LED matrix.

This document discusses the hardware interface between the 5x7 LED matrix and Z8 Encore! XP MCU and provides ready-made display patterns for all the printable ASCII characters, in the form of a two-dimensional array.

Note: *The source code associated with this application note (*`AN0144-SC01.zip`*) contains code for scrolling and/or blinking text on a multi-unit 5x7 LED panel display and is available for download at* www.zilog.com.

Family Overview

Z8 Encore! XP® Flash MCU

The Z8F640x/Z8F480x/Z8F320x/Z8F240x/Z8F160x Flash Microcontroller products, hereafter referred to collectively as Z8 Encore! XP, are based on the new 8-bit eZ8 CPU and introduce Flash memory to Zilog's extensive line of 8-bit microcontrollers. Flash memory in-circuit programming capability allows faster development time and program changes in the field. The high-performance register-to-register-based architecture of the eZ8 core maintains backward compatibility with Zilog's popular Z8® MCU.

The new Z8 Encore! XP microcontrollers combine the 20 MHz eZ8 CPU core with Flash memory, linear-register SRAM, and an extensive array of on-chip peripherals. These peripherals make the Z8 Encore! XP suitable for various applications including motor control, security systems, home appliances, personal electronic devices, and sensors.

Features

Key features of Z8 Encore! XP MCU include:

- New high-performance 20 MHz eZ8 CPU
- Up to 64 KB Flash memory with in-circuit programming capability
- Up to 4 KB register SRAM
- 12-channel, 10-bit Analog-to-Digital Converter (ADC)
- Two full-duplex UARTs
- Two Infrared Data Association (IrDA)-compliant encoders and decoders
- Serial Peripheral Interface (SPI) and I^2C ports
- Four 16-bit timers with capture, compare, and PWM capability
- Watchdog Timer (WDT) with internal RC oscillator
- Three-channel DMA
- Up to 60 I/O pins
- 24 interrupts with configurable priority
- On-chip debugger
- Voltage Brownout (VBO) protection
- Power-On Reset (POR)

The Z8 Encore! XP MCU is capable of up to 10 MIPS throughput at 20 MHz. The 4 KB SRAM extends Z8 Encore! XP to a wider range of applications, the 10-bit sigma/delta ADC provides high measurement resolution, and the SPI, UART, and I^2C interfaces can be used concurrently. The versatility of the DMA controllers allow many useful configurations to free the CPU from overhead tasks.

Discussion

Light-emitting diodes (LEDs) provide a cheap and convenient way to display information electronically. LEDs are tiny light sources that are illuminated solely by the movement of electrons in semiconducting materials. They emit light when forward-biased, fit easily into an electrical circuit, and are durable. LEDs are often arranged in patterns to display information. The seven-segment configuration of an LED arranged in the form of the digit 8 can be restrictive in that it does not adequately allow the display of some alphanumeric characters. By contrast, the versatility of a dot-matrix arrangement allows an LED unit to display complicated shapes.

The following sections discuss the construction of a dot-matrix LED arrangement and how it interfaces with a microcontroller.

LED Matrix Construction

Based on the electrode connections, two types of LED matrices are explained below:

An LED matrix with a common anode for LEDs in a row—When a row of LEDs feature a common anode, the LEDs in each column must include a common cathode.

An LED matrix with a common cathode for LEDs in a row—When a row of LEDs feature a common cathode, the LEDs in each column must include a common anode.

Figure 1 displays a matrix construction of the common anode type. A single matrix is formed by thirty-five LEDs arranged in five columns and seven rows (5x7). The anodes of the five LEDs forming one row are connected together. Similarly, the cathodes of the seven LEDs of a column are connected together. In this arrangement of LEDs, the cathodes are switched to turn the LEDs of a row ON or OFF.

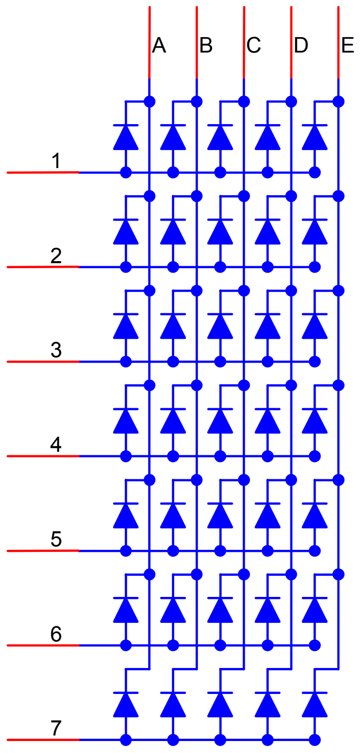

Figure 1. LED Matrix with Common Anode Arrangement

Figure 1 displays the matrix (unit) which can be used to display a single alphanumeric character. Several such units can be placed next to each other to form a larger panel to display a string of characters.

Interfacing the LED Matrix to MCU Hardware

For a 5x7 LED matrix, it is impractical to assign one GPIO pin per LED, as a four-unit, 5x7 matrix requires 35x4=140 GPIO pins. Such a display scheme can cause higher power dissipation and is not easily scalable. Multiplexing can be used to reduce the number of GPIO pins, with only a small increase in the amount of external hardware, such as latches.

Each row of the LED is driven for a brief period before switching to the next row. Because of a visual phenomenon termed *persistence of vision*[1], rapid switching between rows produces the illusion that all of the rows are ON at the same time. To function as intended, the two additional requirements must be met are:

1. The LEDs must be *overdriven* proportionately or they can appear dim. The dimness occurs because a row is ON for only a fraction of time.
2. The rows must be updated often enough (for example, each row is scanned about 30–40 times per second), to avoid display flicker.

For actual character display, it is necessary to map the shape of the character to the 5x7 LED matrix. Figure 2 displays the characters A and B. For any given character, a corresponding pattern of LED ON and LED OFF must be generated and stored in memory and must be used to display the character at run time. For example, the character A, as displayed in Figure 2, is formed with the pattern shown in Table 1.

Other characters/objects can be developed in a similar manner and stored in memory to be used while displaying. By frequently switching the rows or columns with the proper selection of LED ON/OFF patterns, the human eye perceives the display as continuous.

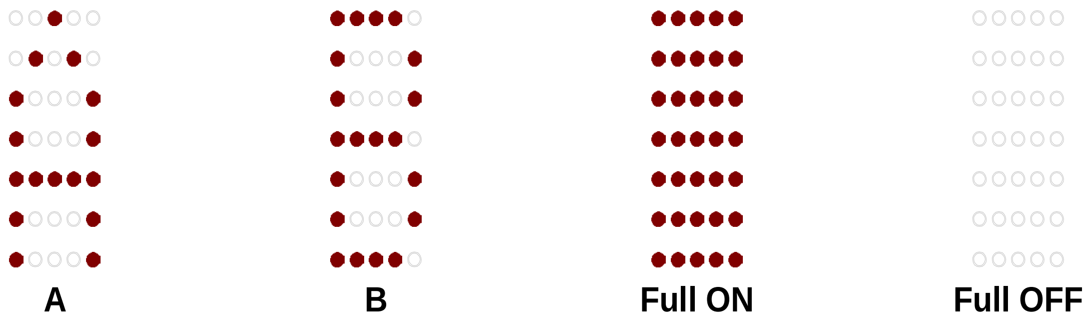

Figure 2. Illustration of the Characters A and B

1.The human eye retains a visual impression of an object for a short duration after the object is removed. Retention time depends on the brightness of the image. This capacity to retain an image is known as *persistence of vision*.

Table 1. Display Pattern for the Character A*

	C0	C1	C2	C3	C4	C0	C1	C2	C3	C4	Data
R0	OFF	OFF	ON	OFF	OFF	1	1	0	1	1	0x1B
R1	OFF	ON	OFF	ON	OFF	1	0	1	0	1	0x15
R2	ON	OFF	OFF	OFF	ON	0	1	1	1	0	0x0E
R3	ON	OFF	OFF	OFF	ON	0	1	1	1	0	0x0E
R4	ON	ON	ON	ON	ON	0	0	0	0	0	0x00
R5	ON	OFF	OFF	OFF	ON	0	1	1	1	0	0x0E
R6	ON	OFF	OFF	OFF	ON	0	1	1	1	0	0x0E

Note: *0 = LED ON; 1 = LED OFF.

Z8 Encore! XP® GPIO Control for 5x7 LED Matrix Display

Hardware Architecture

Four 5x7 LED units are placed adjacent to each other to display four characters at a time. This arrangement results in twenty columns (5x4) and twenty-eight rows (7x4) of LEDs to multiplex.

Figure 3 displays a block diagram showing four units of 5x7 LEDs. In Figure 3, seven rows of a unit lead to a row latch and five columns of the same unit lead to a column latch. There are four row latches (R0–R3) and four column latches (C0–C3). These latches are driven by GPIO pins. The clocks for the row latch and the column latch of a single unit are connected together and are driven by a single GPIO pin. For example, CLK0 drives R0 and C0 for unit 0.

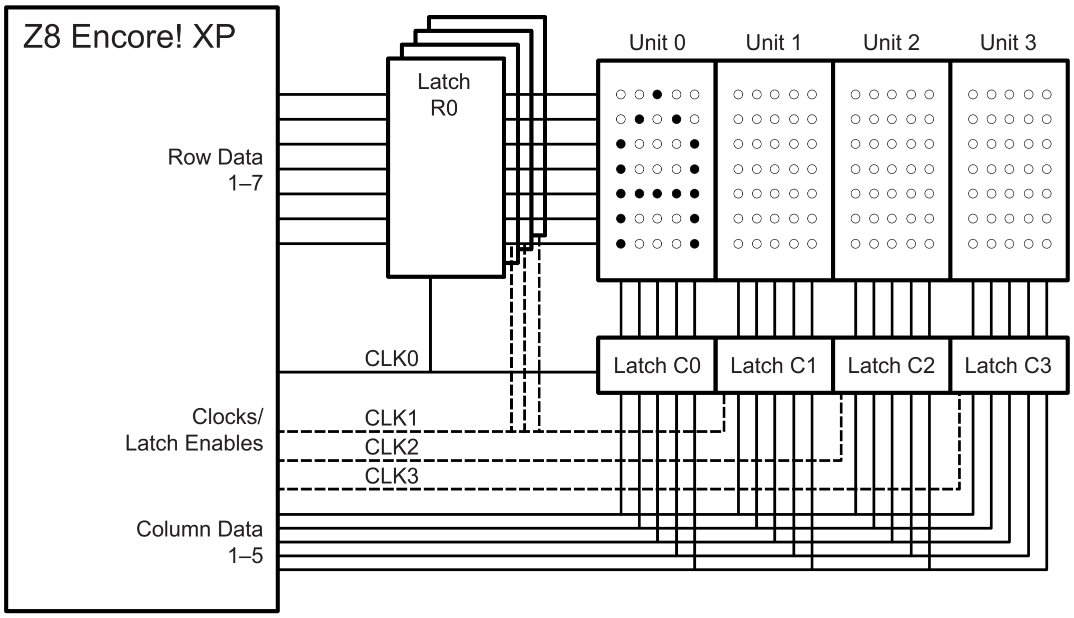

Figure 3. Hardware Block Diagram of Four units of 5x7 LEDs

The GPIO pins that drive the row data for all units are multiplexed, as are the GPIO pins that drive the column data for all units. The clocks, however are different for individual units. Switching data between these units is achieved by using clocks (CLK0 through CLK 3) that connect the latch of a row to the corresponding latch of a column. For example, CLK0 connects Latch R0 and Latch C0.

A latch holds latched data even in the absence of further input from the MCU, thereby serving a dual purpose explained below:

- It affords less frequent display refreshes as the data is held by the latches and continues to be displayed in between the refreshes.
- It allows higher current sinking and sourcing capabilities than when the microcontroller is used alone.

Software Implementation

This section describes how the arrangement outlined in the previous section is used to display strings.

Data Structure for Display Patterns

Each unit matrix that displays a single character is composed of seven rows and five columns. The data that displays a single row is held in one byte of data. Thus, each character is represented by seven bytes of data—one byte each to hold the data for a particular row.

The total number of characters supported is 95, ranging from ASCII code `0x20` to `0x7E`. The memory space required to store character data is, in effect, 95 characters x 7 bytes, or 665 bytes.

Information is arranged in a two-dimensional, 95x7 array that comprises the data structure for the display patterns. The three most significant bits (msb) in each of these bytes do not contribute to the display and are arbitrarily set to zero while forming the data byte.

As derived in Table 1, the data array for character A (ASCII value = `0x41`) is:

```
{ 0x1B, 0x15, 0x0E, 0x00, 0x0E,
0x0E, 0x0E }
```

The file `matrix.h` contains the two-dimensional array for all 95 printable ASCII characters, arranged from the *space* character (ASCII value = 0x20) to the *tilde* (~) character (ASCII value = 0x7E), in a sequential manner.

Displaying Characters

The software implementation for the 5x7 LED display is divided into foreground and background tasks. The `main` routine works as a foreground task and is responsible for the initialization of the GPIO ports. It copies the string to be displayed into a user-specified memory location, and appends the string with:

- Leading spaces
- Trailing space
- NULL character

These spaces visually mark the start and end of a string, which allows a string to be easily read by the human eye when text is scrolling. The `main` routine also tracks character display data and keeps that data ready for display at the latch inputs

The array data starts from ASCII character `0x20` (the *space* key), therefore 0x20 is subtracted from the character ASCII code. The resulting number provides an array index between 0–94 that points to the first row for the character.

According to its row position on the unit matrix, the corresponding row data for the character is fetched and output to the port. Although this data is available, it is not displayed immediately; rather, it waits until the data is latched by the background task.

When a string is longer than the number of LED units available—it is necessary to scroll the characters to display the entire string.

A two-step approach is necessary to scroll a string of characters, as the following sequence shows:

1. A section of the string containing the same number of characters as the number of units is *windowed*, or selected, for display.
2. This window is moved by one character to exclude the extreme left character and include the next character to be displayed at the extreme right position. As a result, the display scrolls from left to right, effectively creating a perception that the string is moving.

Blinking a message can be performed by alternately switching the display on and off at suitable intervals. The software implementation supports both the scrolling and blinking mechanisms, as well as combination of the two.

The `main` routine (with other user application code) is interrupted at 1 ms intervals by an interrupt service routine (ISR) that acts as a background task. It obtains the present unit and row position from the `main` routine. From the `update_flag` set by the `main` routine, the ISR also detects when data is ready. The ISR is responsible for latching the data kept ready by the `main` routine on appropriate latches. It also resets the `update_flag` to force the `main` routine to keep fresh data ready for latching.

The sequence of tasks to be performed by the `main` routine are provided below:

1. Initialize the GPIO and Timer0.
2. Place the string to be displayed into the `active_string` array.
3. Check the `update_flag` to decide if a data update is required. If not, return to step 9.
4. Check for the most recent unit and row position displayed.
5. If the displayed unit is the extreme right unit, point to the next row in the extreme left unit, or else point to the same row in the next unit.
6. Get the data byte for the next unit/row position.
7. Output data on Port E (column latch input).
8. Enable the selected row through Port G (row latch input).
9. Return to step 3.

The sequence of tasks for the ISR is as follows:

1. Generate a pulse for latching data on the selected unit.
2. Set the `update_flag` to indicate to the `main` routine that fresh data is required for the next unit/row position.

See Appendix A—Flowcharts on page 8 for the flowcharts which illustrate the above routines. The APIs are described in Appendix B—API Description on page 11.

Testing

Test Setup

The software for this application note was tested on the Z8 Encore! XP® Evaluation Board. For schematics and further details of the Z8 Encore! XP Development Kit, refer to *Z8 Encore!® Flash Microcontroller Development Kit User Manual (UM0146)*. The compiled code was downloaded to Flash memory using the on-board debug interface. For testing purposes, a 95-character string is used to display all of the characters supported.

Test Procedure

The procedure to test the LED matrix software:

1. On reset, the string is rolled on to the display matrices.
2. Different scrolling and blinking options are set and fresh code is downloaded to Flash memory

to check each possible mode of operation.

3. Scroll rate and blink rate were varied by adjusting the definitions of the SCROLL_DELAY, BLINK_PERIOD, and BLINK_ON_TIME variables. The effects are observed on the display.

Test Results

The display functioned as intended based on the settings downloaded. These settings can be changed through user software during run time. The settings as provided in the code produce a blink rate of approximately one blink per second and a scroll rate of approximately three characters per second.

Summary

This application note explains how to control the display of characters on a 5x7 LED matrix display using the GPIO ports of the Z8 Encore! XP® MCU. A four-unit 5x7 matrix display (140 LEDs) is controlled by two GPIO ports (16 GPIO pins). The APIs provided in this application note facilitate easy implementation for displaying characters on a 5x7 LED matrix.

The scroll and blink display modes are supported. The display scrolling speed and the blink rate can be easily changed. The software is written to decouple these functions, thus enabling combinations of these modes as per your requirements.

All of the printable ASCII characters (ASCII code 0x20 to 0x7E) are supported. The procedure outlined for displaying printable ASCII characters can also be used for displaying nonstandard characters and symbols.

References

The documents associated with Z8 Encore! XP® MCU available on www.zilog.com are provided below:

- Z8 Encore! XP® 64K Series Flash Microcontrollers Product Specification (PS0199)
- Z8 Encore!® Flash Microcontroller Development Kit User Manual (UM0146)
- ZDS II-IDE — Zilog Developer Studio II–Z8 Encore!® User Manual (UM0130)

Appendix A—Flowcharts

Figure 4 displays the flowchart of main routine.

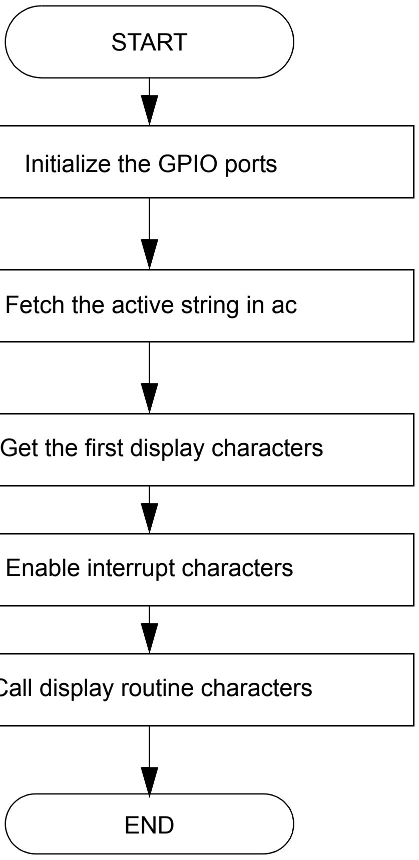
START
Initialize the GPIO ports
Fetch the active string in ac
Get the first display characters
Enable interrupt characters
Call display routine characters
END

Figure 4. Main Routine

Figure 5 displays the flow of the ISR routine.

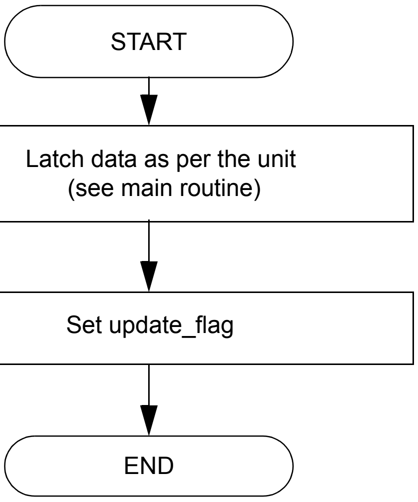

Figure 5. ISR Routine

Figure 6 displays the flow of the scrolling and blinking routine.

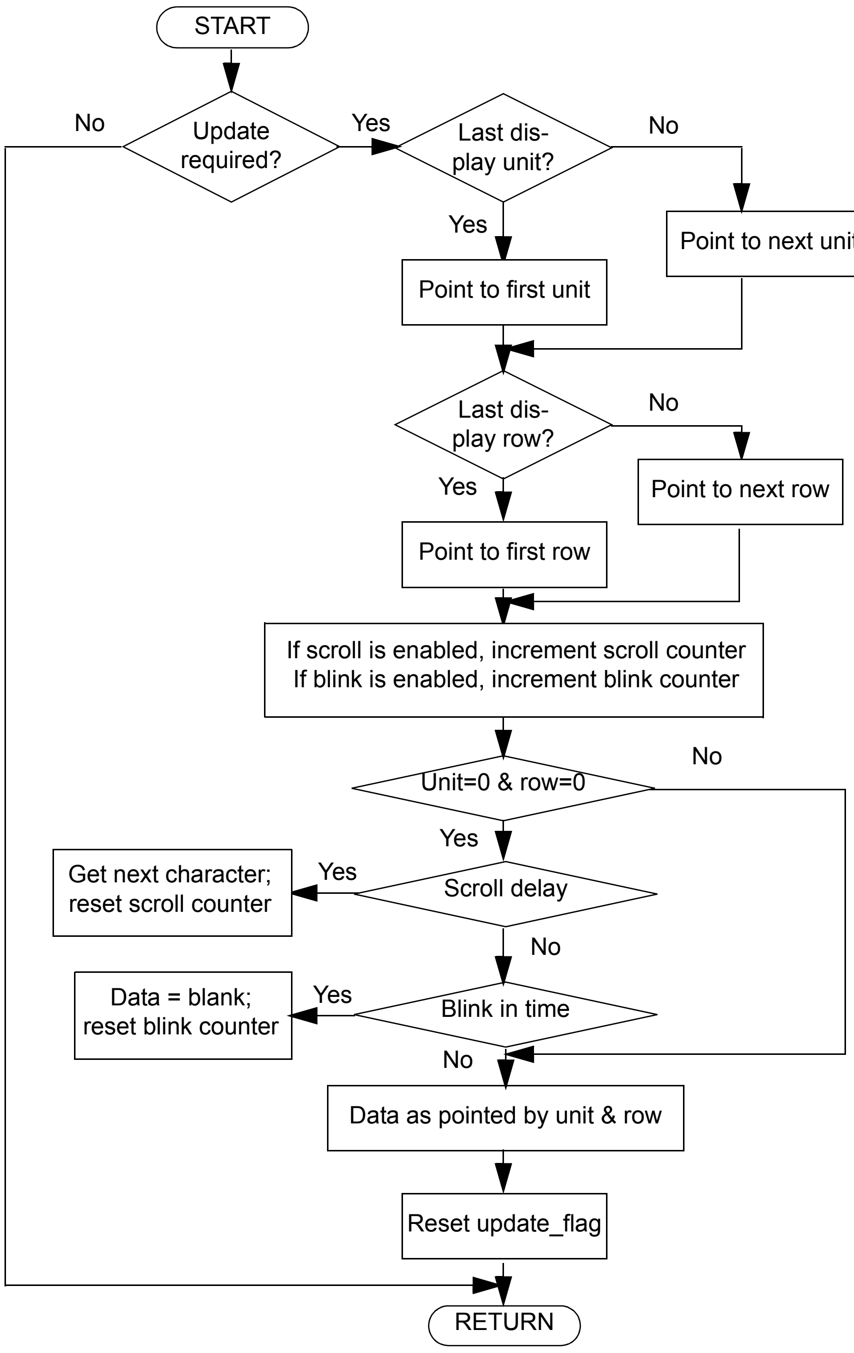

Figure 6. Scrolling and Blinking Routine

Appendix B—API Description

This appendix describes the APIs that perform LED functions.

LED_init

Prototype

```
void LED_init(void);
```

Description

This API initializes the following Z8 Encore! XP® functions:

- Ports E and G
- Timer0
- Row and digit positions

It also loads the default string into the active string location.

Argument(s)

None.

Return Value(s)

None.

Example

Ports E and G are utilized as GPIOs to drive the display through latches. Timer0 generates an interrupt every 1 ms to allow an update of each row every 28 ms in a four-unit, seven-row display configuration, thus achieving a rate of 35 refreshes per second.

LED_clear

Prototype

```
void LED_clear(void);
```

Description

The `LED_clear` API fills the active string locations with a space character (ASCII code `0x20`).

Argument(s)

None.

Return Value(s)

None.

Example

The `LED_clear()` API maintains the scrolling action of the MCU. It can be used when the display must be cleared and updated as soon as the next string is ready. Contrast the usage of this API with `LED_off`.

LED_update_string

Prototype

```
void LED_update_string(unsigned char *src, unsigned char
*dest)
```

Description

The `LED_update_string` API puts the string in the active string space.

Argument(s)

`src`	pointer to the source from where the string is picked up
`dest`	pointer to where the string is to be placed.

Return Value(s)

None.

Example

The `LED_update_string()` API is used to fill the active string locations with the appropriate string. Its basic design offers choices for the source and destination of the string. There can be many sources within one implementation; however, the destination is usually constant and is where other display APIs expect the string to be.

LED_scroll

Prototype

```
void LED_scroll(void)
```

Description

The `LED_scroll` API performs the following tasks:

- Checks for a data update flag
- Gets the next data byte to display
- Keeps track of row/column positions

Argument(s)

None.

Return Value(s)

None.

Example

Note: *Read the following paragraph in conjunction with the usage of the ISR.*

The `LED_scroll()` API assumes the string to be present in the active string location. It keeps the next data byte ready on the ports for latching. The data is fetched, based on the requirement, by testing a flag.

LED_blink_on

Prototype

```
void LED_blink_on(void)
```

Description

The `LED_blink_on` API checks for the blink flag

Argument(s)

None.

Return Value(s)

None.

Example

The `LED_blink_on()` API clears the display in alternate row-scrolling cycles, thus reducing the refresh rate. A reduction in the refresh rate causes the blinking effect, which is valid for the entire active string.

LED_blink_off

Prototype

```
void LED_blink_off(void)
```

Description

The `LED_blink_off` API stops blinking the message. It is the default state.

Argument(s)

None.

Return Value(s)

None.

Example

The `LED_blink_off()` API is used to turn off message blinking. The active string continues to roll without disruption.

LED_off

Prototype

`void LED_off(void)`

Description

The `LED_off` API performs the following tasks:

- Retains the active string in RAM
- Clears the display
- Disables interrupts

Argument(s)

None.

Return Value(s)

None.

Example

Note: *Read the following paragraph in conjunction with the usage of the* `LED_clear()` *API.*

The `LED_off()` API is used to turn off the display mechanism. The API retains the present active string in the RAM location. The string is not displayed in full or in part. It frees Ports E and G along with Timer0. To restart the display, you must regain the ports and timer. `LED_on()` API is provided to exit `LED_off()` mode.

LED_on

Prototype

```
void LED_on(void)
```

Description

The `LED_on` API initializes the following Z8 Encore! XP® functions:

- Ports E and G
- Timer0
- Row and unit matrix positions

Argument(s)

None.

Return Value(s)

None.

Example

The `LED_on()` API is used to exit `LED_off` mode. It is different from the `LED_init()` API in that it does not loads the default string to the active string locations. The display starts with the string that is active prior to the use of the `LED_off()` API.

Warning: DO NOT USE IN LIFE SUPPORT