



Application Note

AN2342

PSoC® Calculator

Author: Andrew Smetana

Associated Project: Yes

Associated Part Family: CY8C21x23, CY8C21x34, CY8C27x43, CY8C29x66

Software Version: PSoC Designer™ 4.2 SP3

Associated Application Notes: AN2343

Abstract

This Application Note describes, in detail, the design stages of a popular arithmetic device; the calculator. It covers the calculator's algorithm, binary coded decimal (BCD) library, and keyboard scanner.

Introduction

The development of calculation methods springs from the dawn of human appearance as homo sapiens. The first calculating machine used by the ancestors of modern people was the human hand. The oldest calculator, the abacus, was created in ancient China more than 4 millennia ago. Initially, it was a board that could be marked on. Later, it transformed into a counting frame. With time, the abacus was replaced with paper and pen. At the beginning of the 17th century the slide rule was created. The slide rule allowed people to quickly approximate the answers to multiplication and division problems. One of the first mechanical calculators was invented by Blaise Pascal of France in 1642. It was built on the basis of cogwheels and could add and subtract decimal numbers. All four basic arithmetic operations were performed by a device designed by German mathematician Gottfried Wilhelm von Leibniz in 1673. It became a prototype for mechanical calculating machines that were used from 1820 until the middle of the 20th century.

The first step on the way to the modern computer was invented by Englishman Charles Babbage in 1822. It was a software-programmable analytical calculating machine that had the equivalent of memory, an arithmetic unit, and input and output devices. His project was ahead of the technical abilities of that time and therefore not implemented.

Electromechanical devices were created toward the end of 19th century. One of them was a Hollerith's Tabulating Machine, which processed the information saved on punched cards. Herman Hollerith established a company that later became IBM.

The first electronic calculating machine, ENIAC (Electronic Numerical Integrator and Computer), was developed in 1946 in the USA. This marked the beginning of the epoch of electronic computers.

The automation of arithmetic operations resulted in the development of the calculator. The invention of the electronic calculator resulted in modern computers, because the first electronic computers, such as Mark-1 and ENIAC in the USA, and SECM (Small Electronic Computing Machine) in the USSR, were used mainly as quick calculating devices. The first computer developers were interested in the creation of powerful, fast, and precise electronic calculators. No one at that time guessed about the technical revolution and information boom.

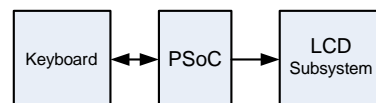
Calculation devices have a very long history. They have powerful modern presentations as PCs and supercomputers. What they will be in the future we can only suppose. No technical device in history has progressed as quickly as the digital computer.

The calculator described in this Application Note would probably be impractical to implement as a standalone calculator, but its components and algorithms are useful in many devices as subsidiary functions.

General Description

The calculator structure, as implemented in this project, is illustrated in Figure 1.

Figure 1. Calculator Block Diagram



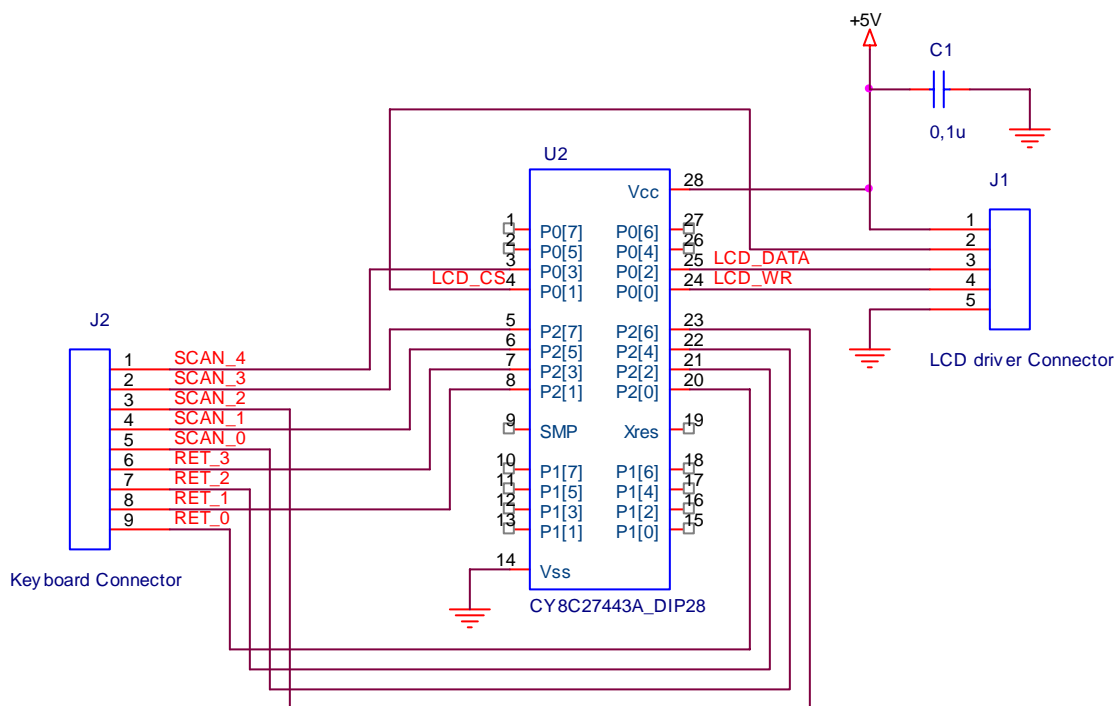
The PSoC device is at the heart of the system. It scans the keyboard, displays the results, executes the calculator algorithm, and processes the mathematical expressions.

The keyboard is a 5x4 button matrix. The LCD subsystem is a glass LCD (Wintek) and a memory-mapping LCD controller (HT1621). The LCD controller is described in [AN2343](#), "LCD Driver Based on the HT1621 Controller."

The calculator schematic is shown in Figure 2. It shows only the PSoC interface lines with the LCD and keyboard modules. A detailed description of each of these units follows. A PSoC device in the CY8C27x43 family was chosen to simplify debugging of the calculator using the in-circuit emulator, ICE-4000. However, given the resources used by the calculator, (mainly SRAM) the project can easily use the CY8C21xxx device family.

Note No PSoC block resources are used. The only device core resource used is the SleepTimer, which is required by the keyboard scanner.

Figure 2. Calculator Schematic



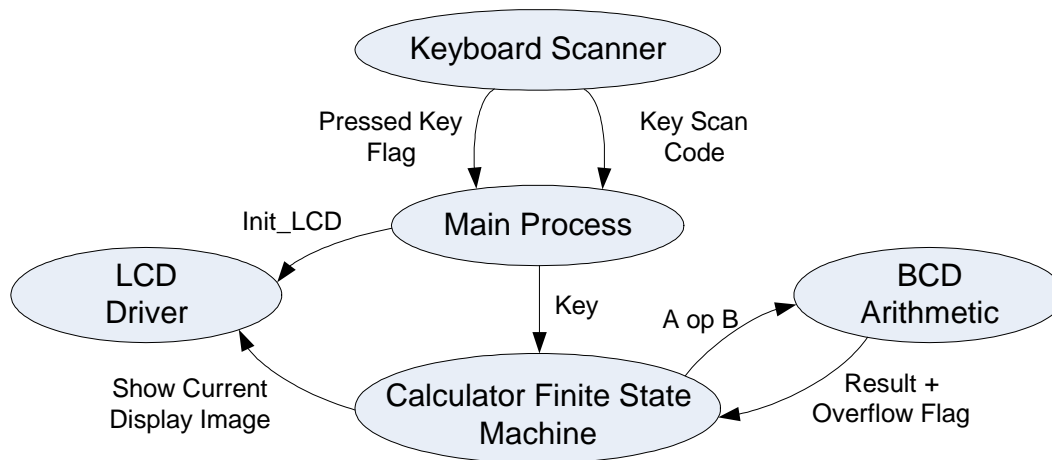
The calculator firmware consists of the following modules:

- Calculator finite state machine (CFSM) and other auxiliary routines
- Keyboard scanner
- Main process
- BCD library
- LCD driver

Figure 3 shows the interactions between the firmware modules.

The main process is a dispatch module. It initiates the calculator's hardware and then goes into a loop of keyboard polling. If a key is pressed, the key pressed flag is set and the key scan code is read. The main process passes this key scan code to the calculator finite state machine (CFSM) and clears the key pressed flag. The CFSM processes each character it receives according to standard calculator functions. The CFSM selects from input data stream operands, arithmetic operation characters, and control keys, and groups them into expressions. It then calls the BCD arithmetic library routines to calculate the result. The result is then shown on the display. Each operand input is also shown on the display.

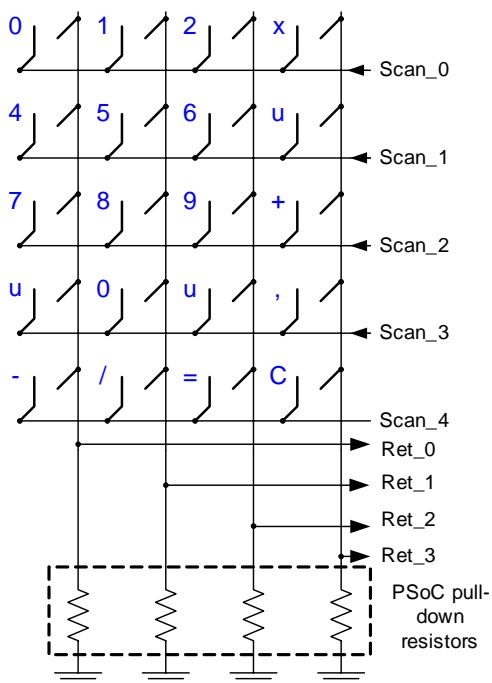
Figure 3. Firmware Module Interaction



Keyboard Scanner

The keyboard of most computing devices is a rectangle matrix with button contacts placed at the row and column intersections. The scanning device must determine the coordinates of the key press and output the corresponding binary code (scan code). In this case, the keyboard is a 5x4 matrix of buttons, as shown in Figure 4.

Figure 4. Keyboard Matrix



Keyboard symbols are defined as follows:

Table 1. Symbols Used in the Keyboard Matrix

Symbol	Meaning
u	Unused button
.x	Multiplication
+	Addition
-	Subtraction
/	Division
.	Decimal point
=	Calculate command
C	Reset button

The scanner works in one of two modes:

1. Waiting mode
 - o All rows Scan_0..Scan_4 are set to 1.
 - o All columns Ret_0..Ret_3 have a low level state.
 - o The state of return lines is polled at a given frequency. If one of the lines changes state from 0 to 1, it signals that a key was pressed and the scanner goes to the Scanning mode.
2. Scanning mode
 - o Scan_0 row is set to 1. All other rows are set to 0.
 - o Return lines are checked.
 - o If one of the return lines (for example, Ret_1) is changed from 0 to 1, it signals a shorting contact on the intersection of Scan_0 and Ret_1. Thus, the location of the pressed key on the key matrix is found.
 - o The scanner passes this information to the coder to form the scan code of the pressed key and then returns to the Waiting mode.
 - o If the level of all columns is low then scanner goes to the next row.

Table 2 shows an example of the scanner operation when the [9] key is pressed.

Table 2. Example of Scanner Operation

No.	Actions	Results	Notes
1	Scan="11111"	Ret="0000"	Scanner is in Waiting mode
2	Scan="11111"	Ret="0010"	Signal is sent that a key is pressed. Scanner goes to Scanning mode
3	Scan="00001"	Ret="0000"	Pressed button is not in the 0th row
4	Scan="00010"	Ret="0000"	Pressed button is not in the 1st row
5	Scan="00100"	Ret="0100"	Key is placed at intersection of 2nd row and 2nd column
6	KeyCode=(Scan<<4)+Ret	KeyCode="001000100"	KeyCode is translated into a scan code and the key pressed flag is set

The keyboard scanner is implemented in firmware. The keyboard line state is polled 64 times per second in the SleepTimer interrupt service routine (ISR). It executes the sequence of steps shown in Table 2. After the button press coordinates are determined (Scan and Return), it translates them into a key scan code using `scan_code_table[5][4]` and the key pressed flag is set. The main process polls the key pressed flag. When it is set, the scan code of the key is passed to the calculator and the flag is cleared. The ISR is implemented in `main.c`. The scanner definitions are in `keyboard.h`.

Calculator Implementation

The calculator implementation is presented in two parts:

- Calculator arithmetic operations of the given precision (10 decimal digits before point and after the decimal point)
- Calculator finite state machine, which processes input flow of scan codes and forms the current display image

Calculator Arithmetic

Two possible solutions to this problem were considered:

- Carry out arithmetic operations with binary arithmetic using fixed- or floating-point data types.
- Use BCD (binary coded decimal) arithmetic.

The first variant is more natural, but given the required precision (20 decimal digits), we need 9 bytes of storage per operand ($\log_2(10^{20}) = 67 \text{ bits} = 9 \text{ bytes}$).

We cannot use the 'C' float type because it uses 4 bytes for a floating-point variable. There are not enough digits in the mantissa to provide the required precision.

As an alternative, the double type could be used (8 bytes long), but it is not implemented in current version of the C compiler and developing it manually is an ineffective use of design time and MCU resources. If we use fixed-point integers, arithmetic operations with 9 byte operands are difficult. The 8-bit PSoC CPU core and its instruction set works with same operand lengths as the floating-point types. The most convincing argument against using these data presentations in our calculator is that it is difficult to translate binary operands into a decimal presentation on the LCD. The results of many of the calculator actions are reflected on the display, so using binary operands would require frequent binary-to-decimal and decimal-to-binary conversions.

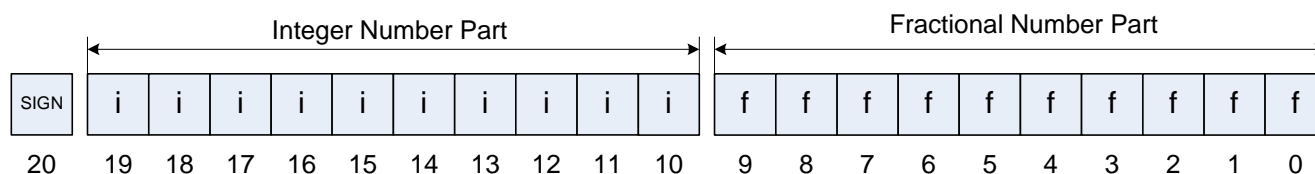
The second solution is to store operands as binary coded decimals (BCD) and do all arithmetic operations with BCD. Neither the PSoC device instructions nor the C compiler support BCD data. Choosing BCD requires that the developer create a BCD function library. The main drawback of BCD numbers is that they require more memory than binary for operands. The main advantage is that the data can be translated to a display view without resorting to difficult mathematical and algorithmic procedures.

Given the specific task of developing a calculator with the developer tools and memory resources of the device, BCD arithmetic is preferable.

This project's BCD library implementation uses 1 byte per decimal digit and 1 byte to represent sign. As a result, a 20-digit BCD number requires 21 bytes of RAM.

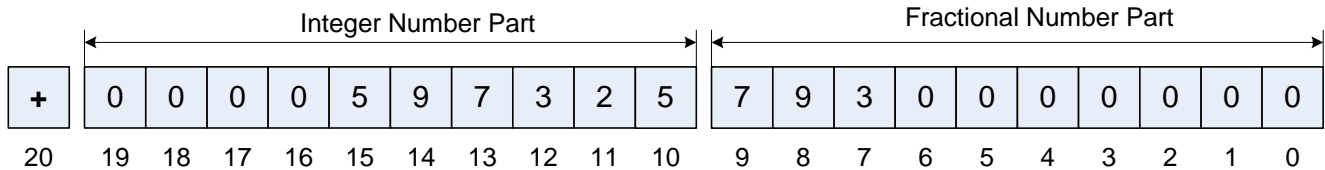
Figure 5 explains the storage of a BCD number in memory:

Figure 5. BCD Number Structure



The number's decimal point is fixed between the 9th and 10th digit positions. For example, the decimal number +597325.793 is stored as shown in Figure 6:

Figure 6. BCD Number Example



All arithmetic operations with BCD operands are implemented according to the decimal rules of long addition, subtraction, multiplication, and division. The BCD subroutines are located in *bcd.c*.

Calculator Finite State Machine

Because the calculator subroutines in the current call are dependent on the previous state, the calculator must be implemented in the form of a finite state machine.

The functional abilities of the calculator are the same as a usual, non-scientific calculator. The calculator flowchart is shown in Figure 7.

The program implementation of the calculator's APIs assumes all variables are located in static RAM. Among the variables are:

`char op1[BCD_OPERAND_LEN+1]` – the first calculator operand.

`char op2[BCD_OPERAND_LEN+1]` – the second calculator operand.

`CALCULATOR calc_display` – the current calculator display image.

```
typedef struct
{
    BYTE BCD[10];
    BYTE Dp;
    BYTE Sign;
} CALCULATOR;
```

- The BCD field contains the current display digits. The leading insignificant zeroes are represented as code 0x0A, and the other decimal digits (0..9) are represented as codes from 0x00 to 0x09.
- The `Dp` field points to the decimal point position in the BCD field. If `Dp==1`, the decimal point is not shown.
- The `Sign` field determines the number's sign; '+' for a positive number, '-' for a negative, and 'E' for an error indication.

`char calc_dot` – 1 if the decimal point has been entered for the current operand, 0 otherwise.

`char calc_op_len` – the length in display positions of the current operand. Used to prevent entering more digits than can be displayed.

`char operator_sign` – the last operator key that was pressed (+, -, *, /).

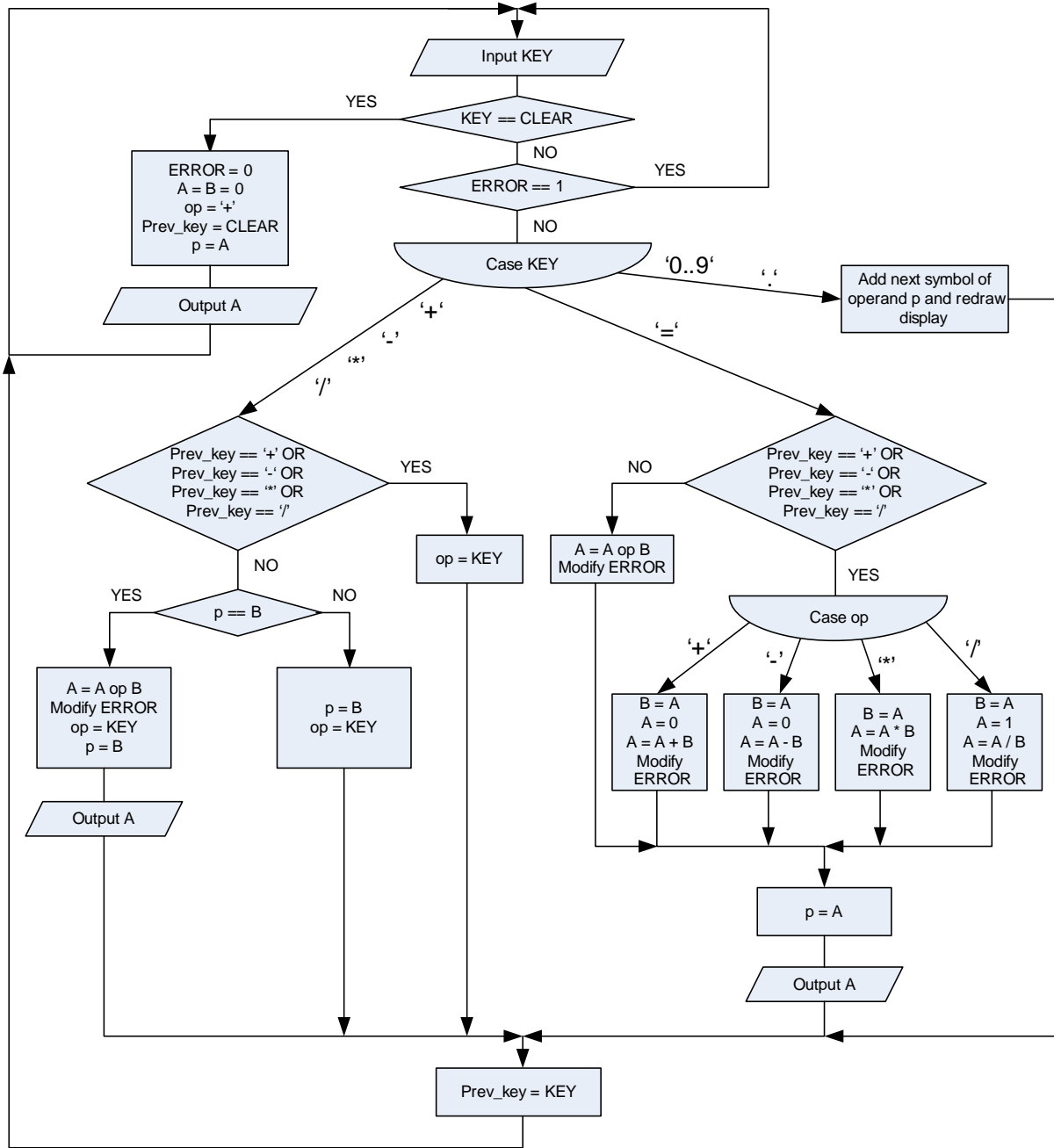
`char prev_key` – when a key press is detected, the key is tested by program logic to determine if it is allowed or not. The `prev_key` variable holds the last allowed key.

`char calc_error` – 1 if an error is present, 0 otherwise.

`char *calc_current_operand` – pointer to the operand currently being input, either `op1` or `op2`.

The calculator related functions (without LCD and keyboard) occupy about 5K of Flash in SMM (small memory model) devices and 6.5K in LMM (large memory model) devices. The RAM use is static memory – 66 bytes and the stack – for a total of about 100 bytes. This summary is useful if you have a project that requires a calculator as an auxiliary system function.

Figure 7. Calculator Flowchart



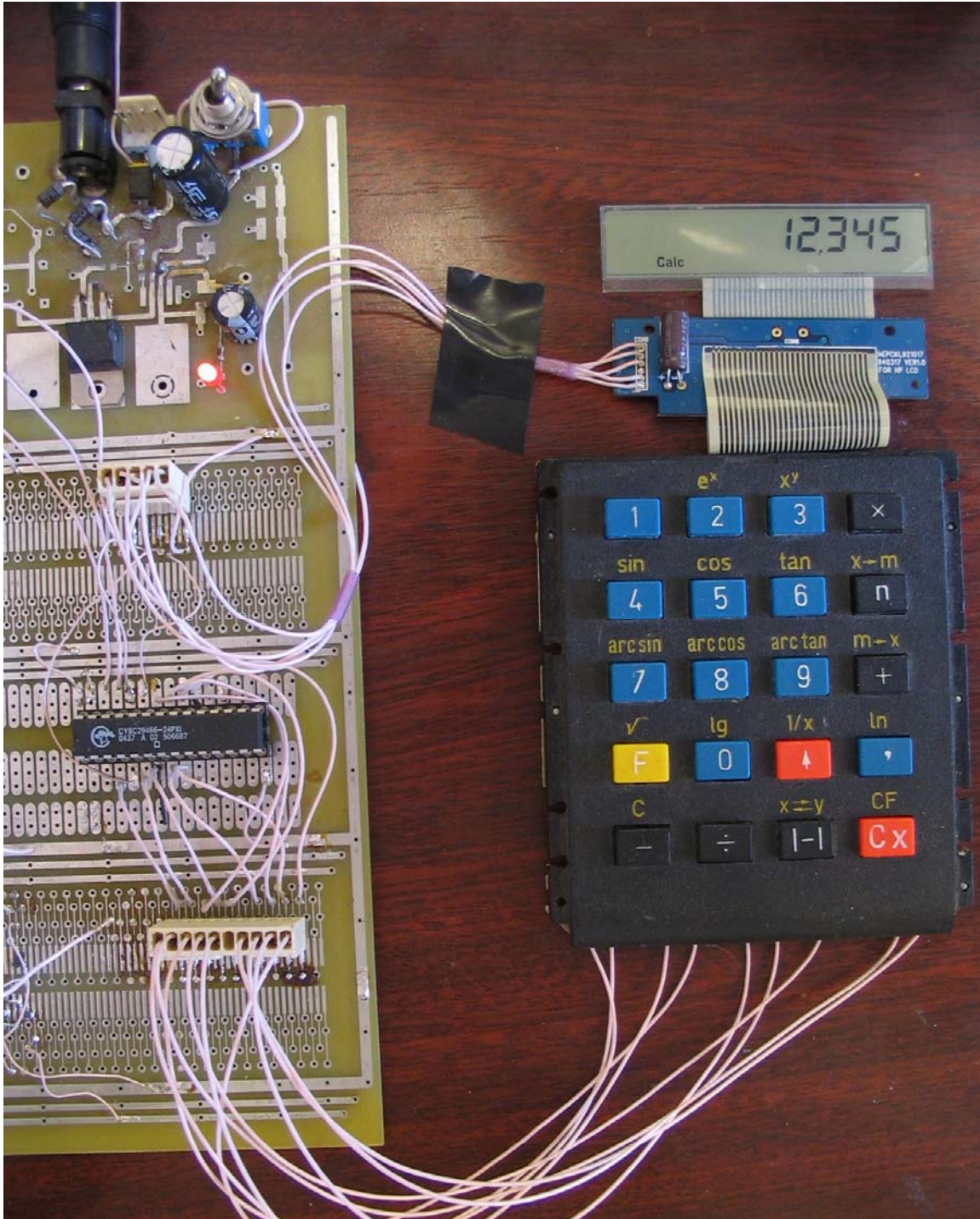
Conclusion

This Application Note contains all the details needed to create a 10-digit calculator that can add, subtract, multiply and divide using binary coded decimals. The associated project contains a complete calculator algorithm and a library of BCD math functions.

The calculator is probably not viable as a standalone product. It can be used, however, as a service function in systems that have keyboards and displays, such as cash machines, phones, industry controllers, and so on.

Appendix

Figure 8. Calculator Prototype Photograph



About the Author

Name: Andrew Smetana

Title: Electronic Engineer

Background: Andrew earned his Master of Science diploma in 2004 from National University "Lviv Polytechnic" (Ukraine). His interests include various aspects of embedded systems development.

Contact: smetana@ukrwest.net

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone: 408-943-2600
Fax: 408-943-4730
<http://www.cypress.com>

© Cypress Semiconductor Corporation, 2006. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.