THIS SPEC IS OBSOLETE

Spec No: 002-04933

Spec Title: How to reduce the microcontroller ROM-size

Replaced by: NONE

The following document contains information on Cypress products. Although the document is marked with the name "Spansion" and "Fujitsu", the company that originally developed the specification, Cypress will continue to offer these products to new and existing customers.

Continuity of Specifications
There is no change to this document as a result of offering the device as a Cypress product. Any changes that have been made are the result of normal document improvements and are noted in the document history page, where supported. Future revisions will occur when appropriate, and changes will be noted in a document history page.

Continuity of Ordering Part Numbers
Cypress continues to support existing part numbers. To order these products, please use only the Ordering Part Numbers listed in this document.

For More Information
Please contact your local sales office for additional information about Cypress products and solutions.

About Cypress
Cypress (NASDAQ: CY) delivers high-performance, high-quality solutions at the heart of today's most advanced embedded systems, from automotive, industrial and networking platforms to highly interactive consumer and mobile devices. With a broad, differentiated product portfolio that includes NOR flash memories, F-RAM™ and SRAM, Traveo™ microcontrollers, the industry's only PSoC® programmable system-on-chip solutions, analog and PMIC Power Management ICs, CapSense® capacitive touch-sensing controllers, and Wireless BLE Bluetooth® Low-Energy and USB connectivity solutions, Cypress is committed to providing its customers worldwide with consistent innovation, best-in-class support and exceptional system value.

Colophon

The products described in this document are designed, developed and manufactured as contemplated for general use, including without limitation, ordinary industrial use, general office use, personal use, and household use, but are not designed, developed and manufactured as contemplated (1) for any use that includes fatal risks or dangers that, unless extremely high safety is secured, could have a serious effect to the public, and could lead directly to death, personal injury, severe physical damage or other loss (i.e., nuclear reaction control in nuclear facility, aircraft flight control, air traffic control, mass transport control, medical life support system, missile launch control in weapon system), or (2) for any use where chance of failure is intolerable (i.e., submersible repeater and artificial satellite). Please note that Spansion will not be liable to you and/or any third party for any claims or damages arising in connection with above-mentioned uses of the products. Any semiconductor devices have an inherent chance of failure. You must protect against injury, damage or loss from such failures by incorporating safety design measures into your facility and equipment such as redundancy, fire protection, and prevention of over-current levels and other abnormal operating conditions. If any products described in this document represent goods or technologies subject to certain restrictions on export under the Foreign Exchange and Foreign Trade Law of Japan, the US Export Administration Regulations or the applicable laws of any other country, the prior authorization by the respective government entity will be required for export of those products.

Trademarks and Notice

The contents of this document are subject to change without notice. This document may contain information on a Spansion product under development by Spansion. Spansion reserves the right to change or discontinue work on any product without notice. The information in this document is provided as is without warranty or guarantee of any kind as to its accuracy, completeness, operability, fitness for particular purpose, merchantability, non-infringement of third-party rights, or any other warranty, express, implied, or statutory. Spansion assumes no liability for any damages of any kind arising out of the use of the information in this document.

Compress

This months microcontroller article is based on an application note and discusses how to reduce the microcontroller ROM-size requirement by using a compression method for storing text-data, having in mind an information display application with on-chip LCD controller.

Background

Our microcontrollers are quite strong when it comes to micros with on-chip LCD controller, because we have quite a lot of series with on-chip LCD controllers, and some which can drive quite a large number of LCD segments.
For example the MB89820 series with 4 COM-lines x 20 SEG-lines could drive 200 individual LCD-segments.

With such a number of segments, its even possible to drive alpha numerical displays based on 14-segment characters. See figure 1.

Note that in the area of alpha numerical displays, you will also often see a requirement for Dot Matrix Display drivers, but these usually require a dedicated LCD controller because the number of segments to drive is to high.
For example compared to the 12 character display mentioned here, a similar dot matrix would require 12x 7x5 = 420 individual segments, which is to much for us.
On the other hand, if someone can live with 14-segment based characters, the system cost will be a lot lower.

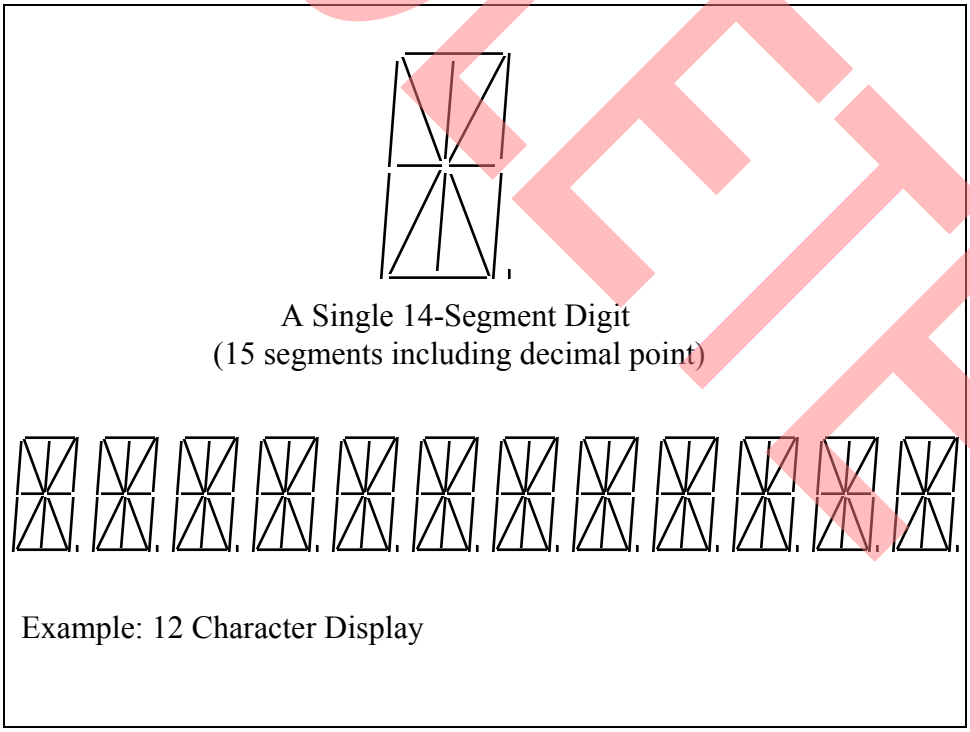

Fig. 1, Example for a 14-Segment Display

Now the general problem is, that small microcontroller systems which can display a set of pre-defined text messages never have enough ROM-space to store all the possible text messages or strings.
This is because the more text-based messages or instructions the system can generated, the more user-friendly it is, so software-developers always tend to run into the limit.

Before showing a solution to store even more strings in the available ROM-space, lets first take a look at the standard way to store strings:

The normal way to store text strings

The normal way to define a text string in the microcontroller ROM, is to store it as a chain of ASCII-characters, which means one byte per character, and eventually an additional special character which defines the end of the string.

In a C-Program this would look like:

```
const char ReadyTextMsg[] = "READY"
```

In the generated Assembly language source code this looks like:

```
_ReadyTextMsg     DB    "READY",H'00
```

with the additional 0-byte as the termination character.

Displaying this string on the LCD display is a rather complicated process which shall be explained only briefly, since its not in the focus on this article.

Basically a display sub-routine function must be coded to display a character on the LCD display.
This function will receive some parameters, like the ASCII code of a single character to be displayed, and the character position.
The display function will use the ASCII code value as an index into a bit-map table.
This will define which of the 14-segments have to be switched on or off to actually display a certain character. The bit-map information is then written into the so called VRAM of the LCD controller.
The write-address must be computed form the character position.
Note that the necessary bit-map structure and calculations depend very much on the physical structure of the LCD display and the correspondence between individual segments and individual bits in the VRAM, and thus the connection scheme of the LCD-display.

The big drawback in the normal way to store strings is the usage of the ASCII standard.
Using one byte or 8-bits per character is good if one really needs the complete character set of 256 different characters, like in the PC are.

In our case, where our display is only capable of displaying a reduced character set
(Capital Letters ´A´-´Y´, numbers ´0´-´9´ and some special, so about 40 different characters),
using the ASCII-code seems quite a waste.

Special way to store text strings

A solution to improve text-storage efficiency seems quite obvious:
Just store the text strings in a compressed manner, for example 4-bits per character.

This is easy to say, but difficult to put into practice, since the C compiler nor the Assembler know a different storage method other than storing text strings byte by byte using the ASCII code.

One could of course use a different utility program to compress text messages into a different binary format and enter the output on a hex-byte basis into the source-code again, but this would be a rather unpracticed solution.

Considering the data processing flow of our C-compiler, its even more straight forward to introduce such a compressing stage automatically.

The Intermediate-Text Compressor

Fig. 2 shows the process flow of a normal compile / assemble process and the modified one, which uses an intermediate processor or my so called compressor to convert ASCII strings in the C-Compiler output into a compressed format.
This intermediate processor is a simple self written utility program, which searches for string definition source lines and replaces them by a special compressed data.

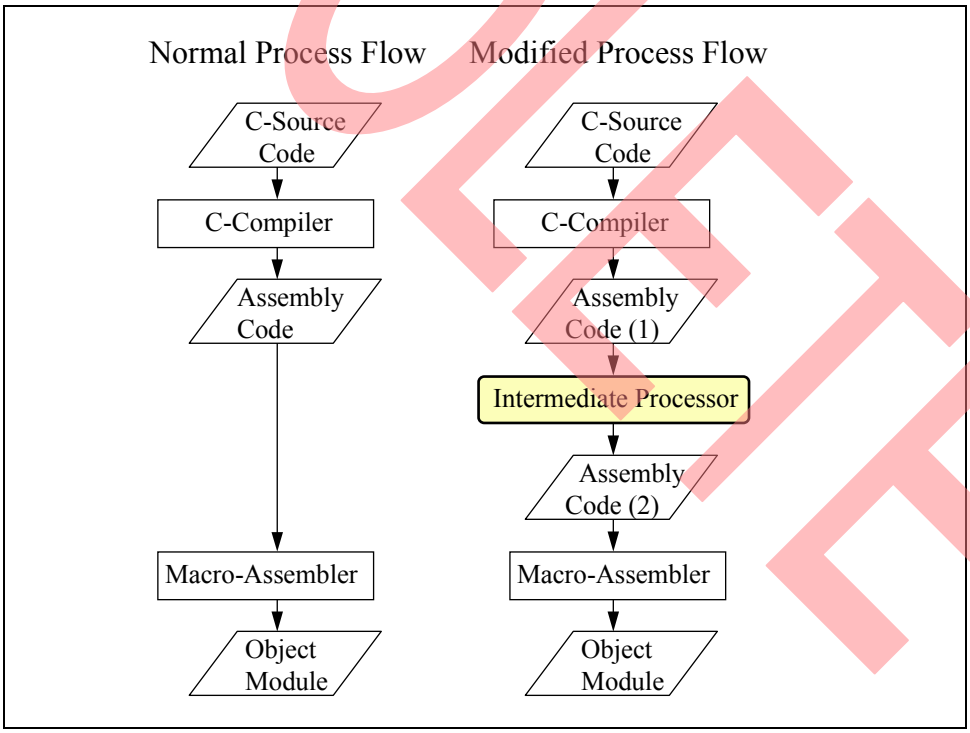

Fig. 2, C-Compiler Process Flow

Of course, the assembly source-code lines which shall be modified be the compressor must be marked by a special header string. In the example here I used the following scheme:

```
#define HeadSequ "]{C}["

const char CSTR Msg[]  =  HeadSequ "HELLO, GOOD MORNING";
```

in the C-Compiler assembly language output, this like looks like

```
_CompInitMsg DB   "]{C}[HELLO, GOOD MORNING",H'00
```

and after compression, assembly line looks like

```
_CompInitMsg DB   H'5F,H'C9,H'2E,H'7C,H'53,H'E,H'70,H'F5
             DB   H'5,H'4,H'D,H'39,H'2
```

which means a reduction from 20 to 13 bytes.

The compression algorithm used in this example is a rather simple one:
In a first pass, all text messages are analyzed to find out which characters out of the complete ASCII set occur and the ranking of occurrences.
With this information, a conversion table is generated.
In a 2nd pass, the ASCII characters are converted into a 4-bit code, which is stored as a binary bit stream.
Since 4-bits can only store 16-different characters, the total number of required characters is divided into some pages, each containing 16 different characters.
One of the characters (in the 1st page) is defined as a special escape character, which is followed by a page value and the index in a different page to be able to address characters in different pages.
So without the overhead of the escape character, the compression rate would be 50%.
In practice, the compression rate is about 40%.
For a better understanding, the de-compression algorithm is shown in figure 3.

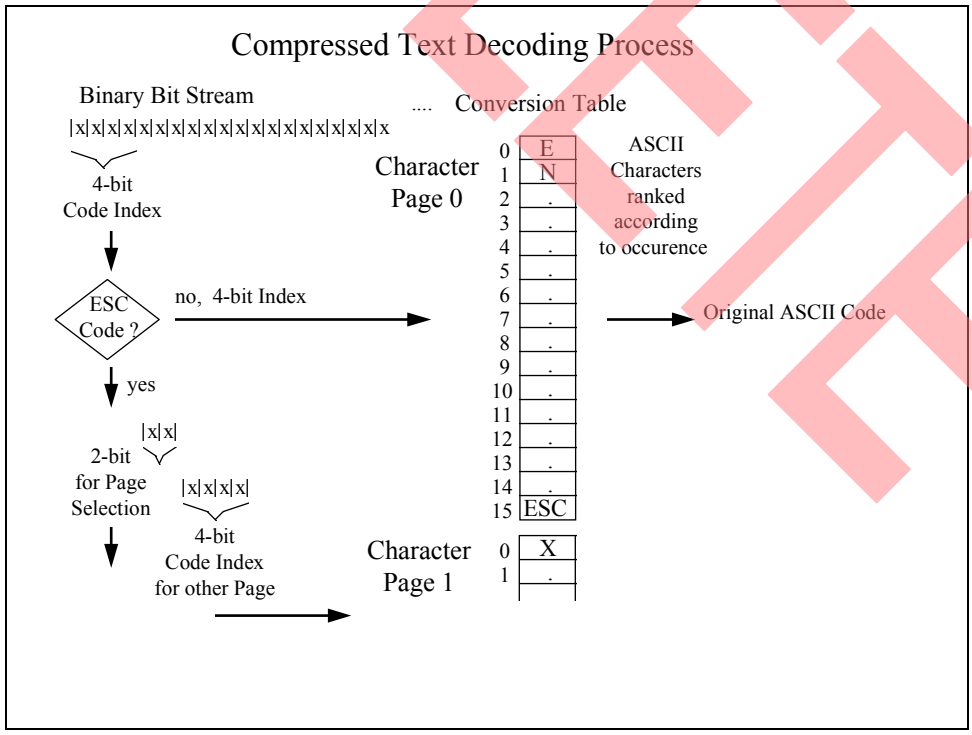

Fig. 3, Decoding Process

Note that this compression scheme is just a simple example and that different compression schemes might yield better compression rates.

For the decompression process, it is of course necessary to have a conversion table and a decompression function on the microcontroller, which adds some additional code-overhead. Note, that the conversion table is generated by the compressor function and is also put into the assembly source by searching and replacing a special header-line.

A nice thing is, that in case of the former mentioned LCD-display application, the conversion table is not really needed if one can organize the bit-map table in the same manner.

Finally the following listing shows an example program for our 8-bit board, where the text messages were compressed during the compile / assemble process, yielding a text-data reduction of about 40%. The microcontroller program also contains a decompression function to convert and list the text messages as ASCII characters again.

Eddie Bendels

```
/*+-------------------------------------------------------------------------+*/
/*|                          F U J I T S U                                  |*/
/*|                                                                         |*/
/*|              M i k r o e l e k t r o n i k   G m b H                    |*/
/*|                                                                         |*/
/*|  Filename:  CT.C                                                        |*/
/*|  Function:  Demo on how to Crompress Text in C-Programs                 |*/
/*|             using a special Post-Porcessor in-between                  |*/
/*|             the Compiler and Assembler Pass                             |*/
/*|  Series:    not specific                                                |*/
/*|  Version:   V01.00                                                      |*/
/*|  Design:    Edmund Bendels 15.07.96                                     |*/
/*|  Note:                                                                  |*/
/*|                                                                         |*/
/*+-------------------------------------------------------------------------+*/

#include <TYPEDEFS.H>                    /* Usefull Type Definitions */
#include "evabios.h"              /* Include EVAKIT BiosInterfaceFunctions */

#define HeadSequ "]{C}["
#define TabSequ  "]{T}["
#define EOSTRCH '^'
#define PAGEESC 15
#define CharPerPage 15

BYTE dyDVAR; direct BYTE dyDIRVAR; /* Dummy Variables avoid LinkerWarnings */
BYTE dyDINT = 0x12; direct BYTE dyDIRINIT = 0x34;


typedef const struct { const char *pStr;} STR_ARY;

CSTR   ver[]      = "TextComp Example V 0.1, FUJITSU Mikroelektronik";
CSTR   date[]     = __DATE__;
CSTR   time[]     = __TIME__;

CSTR NormalMsg[]  = "Text Compressor Demonstration\n";
CSTR CrLfMsg[]    = "\n";


/*
   The following Strings contain a special Header
   which mark them for the intermediate-processor
*/

CSTR  ConvTab[]  = TabSequ;              /* replace by actual Conv.Table */

const char CompInitMsg[]  =  HeadSequ "HELLO, GOOD MORNING";

#define nOpStr 18
STR_ARY OptionStr[] = {
 { HeadSequ "SELECT OPTIONS" },
 { HeadSequ "CHANNEL" },
 { HeadSequ "SET TIME" },
 { HeadSequ "PROG RECORD" },
 { HeadSequ "SHOW RECORD" },
 { HeadSequ "STATUS" },
 { HeadSequ "PLAY" },
 { HeadSequ "RECORD" },
 { HeadSequ "REWIND" },
 { HeadSequ "FOREWARD" },
 { HeadSequ "SLOW PLAY" },
 { HeadSequ "STOP" },
 { HeadSequ "WARMING UP" },
 { HeadSequ "TAPE ERROR" },
 { HeadSequ "ACESS DENIED" },
 { HeadSequ "INSERT TAPE" },
 { HeadSequ "NO TAPE" },
 { HeadSequ "PLEASE WAIT" }
```

```
};

/* CSTR TestStr[] = { 0x89, 0x10, 0x04  }; */

BYTE *pGetData;
BYTE AcData;
BYTE nAcBits;

/*--------------------------------*/
/*-- Convert a number of Bits   --*/
/*-- from a Stream to a Byte    --*/
/*--------------------------------*/
BYTE GetBits(BYTE nBits)
{
  BYTE Mask, j;
  BYTE Val, Tmp;

        if (!nAcBits) {
          AcData  = *pGetData++;
          nAcBits = 8; }

        Mask = 0;
        for (j=0;j<nBits;j++) {               /* Create Bits Mask */
          Mask = (Mask << 1) | 0x01; }

        /*-- Get 4-Bit Nibble Data --*/
        Val = AcData & Mask;                  /* get DataBits */
        if (nAcBits < nBits) {                /*- if not all Bits present -*/
          Tmp = AcData = *pGetData++; /* take some bits from following Data */
          for (j=0;j<nAcBits;j++) {           /* align to available bits */
            Tmp = Tmp << 1; }
          Val = (Val | Tmp) & Mask;
          nAcBits = nAcBits + 8 - nBits;      /* calculate remaining bits */
          AcData = AcData >> (8-nAcBits);     /* new remaining data */
        }
        else {                                /*- all bits were available -*/
          AcData = AcData >> nBits;           /* new remaining data */
          nAcBits -= nBits;
        }
        return Val;
}

/*--------------------------------*/
/*--  Decompress a Text String  --*/
/*--------------------------------*/
void PrintCompStr(BYTE *pS)
{
   BYTE Ix, Ch;
   WORD Ofs;

        pGetData = pS;
        nAcBits = 0;

        do {
          Ofs = 0;
          Ix = GetBits(4);                /* Get 4/bit Nibble */
          if (Ix==PAGEESC) {              /* if switch to Other Page */
            Ch = GetBits(2);
            Ofs = CharPerPage * Ch;       /* calculate Page Offset */
            Ix = GetBits(4);              /* Get 4/bit Nibble of character Code */
          }
          Ch = ConvTab[Ix+Ofs];           /* Convert to ASCII */
          if (Ch != EOSTRCH) putch(Ch); /* if not Termination Character */
        } while (Ch != EOSTRCH);
}

/*--------------------------------*/
/*      Test Program              */
```

```
/*--------------------------------*/
void main ()
{
   char *pStr;
   WORD ix;

        puts(NormalMsg);                /* Print Initial Message */

        PrintCompStr(CompInitMsg);      /* Print One Compressed Message*/
        puts(CrLfMsg);

        for (ix=0;ix<nOpStr;ix++) {
          PrintCompStr( OptionStr[ix].pStr );
          puts(CrLfMsg);
        }
}
```