AN-1050
APPLICATON NOTE

One Technology Way • P.O. Box 9106 • Norwood, MA 02062-9106, U.S.A. • Tel: 781.329.4700 • Fax: 781.461.3113 • www.analog.com

A Method for Compressing I²C Scripts for the ADV74xx/ADV75xx/ADV76xx/ADV78xx

by Witold Kaczurba

INTRODUCTION

This application note describes a method for compressing large sets of I²C scripts for microcontroller platforms. The information herein is targeted for applications where the user needs to put over 50 scripts to the memory of one single microcontroller. This method provides excellent results for sets of scripts containing more than 200 scripts for more than six devices on an I²C bus.

Analog Devices, Inc., multiformat decoders allow users to decode various standards of video. Because a variety of standards are supported, these video decoders provide various settings. Each setting contains the bulk of I²C writes for each mode. These writes are collected as scripts.

In some cases, the user may use hundreds of scripts to configure many I²C devices. In such a case, the user requires a lot of memory in a small microcontroller to keep all of these scripts. Note that these scripts may have similar writes as well as unique writes. This application note details how to compress the script on the PC side, as well as how to write an efficient decompressing method on the microcontroller side. The decompressing algorithm utilizes more efficient functions requiring less RAM.

This application note includes a script for an Octave program. Octave is a free (GNU/GPL license) computer program for numerical computations. This Octave script can compress scripts and export the results to the C code with decompressing procedures. The resulting C code is easy to use, and is easily portable to any microcontroller.

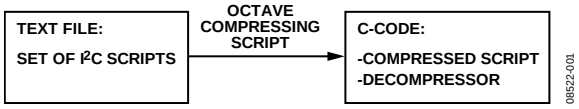

Figure 1. Concept of Compressing Algorithms

TABLE OF CONTENTS

ABOUT THE SCRIPTS

BASIC KNOWLEDGE

The compression method described in this application note is intended to be used with sets of scripts constructed as shown in the Construction of the Script section. One I²C write consists of three values:

- device address (to address the device on the I²C bus)
- register address
- value to be written to the register

Usually scripts provided with evaluation boards by Analog Devices use ascending order by I²C writes. This means that writes to the same device are in ascending order of register addresses, such as:

- 42 00 AB
- 42 01 CD
- 42 02 EF

This not accidential; the order of writes makes compression more efficient because it is more likely to find the same pattern across various scripts. Note that it is unlikely to find the same write to the same register of the same device, such as:

- 42 03 04
- 42 03 08

This increases the probability of the same sequences of writes occurring in different scripts. This is why this application note describes algorithms in which scripts are checked for occurrence of sequences of four identical writes, as shown in Figure 2.

```
##SCRIPT##          ##SCRIPT##
:SCRIPT 1:         :SCRIPT 2:
42  03  04;        42  00  03;
42  04  06;        42  02  27;
42  0A  53;        42  03  09;
42  0C  23;        42  04  FF;
42  0F  45;        42  0A  53;
42  22  39;        42  0C  23;
42  AA  10;        42  0F  45;
42  FF  03;        42  22  39;
...                42  FF  02;
                   ...
```

08522-002

Figure 2. Sequence of Four Identical Writes Occurs in Two Different Scripts

Storing large sequences (of four writes) that are common for various script big blocks is beneficial to the microcontroller side. These common blocks (also called keys) can be stored as a continuous array in a C program. Thus, each key is easily addressable. This ease of addressing the keys eliminates many pointers that would otherwise have to be used in the decompresssing algorithm. Each constant pointer requires memory.

CONSTRUCTION OF THE SCRIPT

The Octave script that is used for compression requires an original set of scripts to be stored in a file in a particular way. Each original script starts with a header (the first line) containing the characters ## at both the beginning and the end of the line. The next line in the script is a small header that must contain colons at both the beginning and at the end of the line. The following line, the third line, contains proper I²C writes, such as:

42 05 02 ; Prim_Mode = 010b for GR

where:
42 indicates the device's 8-bit, I²C address (0x42).
05 is the 8-bit, register address (0x05).
02 is the 8-bit value (0x02).
Prim_Mode = 010b for GR is an optional, user-defined comment.

Note the spaces and semicolon in the equation. The last line of the script is single word End, without any spaces. Scripts are split by the use of empty lines.

The following pages provide an example portion of the set of scripts for the ADV7401 evaluation board (EVAL-ADV7401EBZ).

```
##CP VGA 640x480##
:640x480 _@ 60 Autodetecting sync source  25.175 MHz out through DAC:
42 05 02 ; Prim_Mode = 010b for GR
42 06 08 ; VID_STD = 1000b for 640 × 480 @ 60
42 1D 47 ; Enable 28 MHz crystal
42 3A 11 ; Set latch clock settings to 001b, Power down ADC3
42 3B 80 ; Enable external bias
42 3C 5C ; PLL_QPUMP to 100b
42 6A 00 ; DLL phase adjust
42 6B 82 ; Enable DE output, swap Pr& Pb
42 73 90 ; Set man_gain
42 7B 1D ; TURN OFF EAV & SAV CODES Set BLANK_RGB_SEL
42 85 03 ; Enable DS_OUT
42 86 0B ; Enable stdi_line_count_mode
42 8A 90 ; VCO range to 00b
42 F4 3F ; Max drive strength
42 0E 80 ; Analog Devices recommended setting
42 52 46 ; Analog Devices recommended setting
42 54 00 ; Analog Devices recommended setting
42 0E 00 ; Analog Devices recommended setting
54 00 13 ; Power-down encoder
74 EE EE ; Power-down HDMI
End

##CP VGA 640x480##
:640x480 _@ 72 Autodetecting sync source  31.5 MHz out through DAC:
42 05 02 ; Prim_Mode = 010b for GR
42 06 09 ; VID_STD = 1001b for 640 × 480 @ 72
42 1D 47 ; Enable 28 MHz crystal
42 3A 11 ; set latch clock settings to 001b, Power down ADC3
42 3B 80 ; Enable external bias
42 3C 5C ; PLL_QPUMP to 100b
42 6A 00 ; DLL phase adjust
42 6B 82 ; Enable DE output, swap Pr& Pb
42 73 90 ; Set man_gain
42 7B 1D ; TURN OFF EAV & SAV CODES Set BLANK_RGB_SEL
42 85 03 ; Enable DS_OUT
42 86 0B ; Enable stdi_line_count_mode
42 F4 3F ; Max drive strength
42 0E 80 ; Analog Devices recommended setting
42 52 46 ; Analog Devices recommended setting
42 54 00 ; Analog Devices recommended setting
42 0E 00 ; Analog Devices recommended setting
54 00 13 ; Power down encoder
74 EE EE ; Power down HDMI
End
```

```
##CP VGA 640x480##
:640x480 _@ 75  Autodetecting sync source  31.5 MHz Out through DAC:
42 05 02 ; Prim_Mode = 010b for GR
42 06 0A ; VID_STD =1 010b for 640 × 480 @ 75
42 1D 47 ; Enable 28 MHz crystal
42 3A 11 ; set latch clock settings to 001b, Power down ADC3
42 3B 80 ; Enable external bias
42 3C 5C ; PLL_QPUMP to 100b
42 6A 00 ; DLL phase adjust
42 6B 82 ; Enable DE output, swap Pr& Pb
42 73 90 ; Set man_gain
42 7B 1D ; TURN OFF EAV & SAV CODES Set BLANK_RGB_SEL
42 85 03 ; Enable DS_OUT
42 86 0B ; Enable stdi_line_count_mode
42 F4 3F ; Max drive strength
42 0E 80 ; Analog Devices recommended setting
42 52 46 ; Analog Devices recommended setting
42 54 00 ; Analog Devices recommended setting
42 0E 00 ; Analog Devices recommended setting
54 00 13 ; Power down encoder
74 EE EE ; Power down HDMI
End
```

Note that the device address used in the scripts is always an even number greater than 0. Odd addresses are used for reading back from the device, which does not occur in this case. Thus, Address 0x00 is used as an escape code, a special code used for decompressing.

COMPRESSION/DECOMPRESSION

The Octave script in the Octave Source Code section compresses the script on the PC side. An output of Octave script is C code containing compressed data and the decompression algorithm.

The Octave script algorithm consists of the following steps:

1. The script is loaded into matrices of various dimensions (size is dependent on length of script).
2. The matrices are searched for common blocks or keys where each key consists of four writes.
3. If a common block is found, it is written to a separate block called keys with a key_number index.
4. Where there is a common block, the block is replaced with a single write: (0x00, A, B) where A × 256 + B is a value key_number.
5. The key_number index is increased by one.
6. Finding and replacing the common blocks is repeated until a search from beginning to end produces no results.

This process creates the structure of matrices as shown in Figure 3.

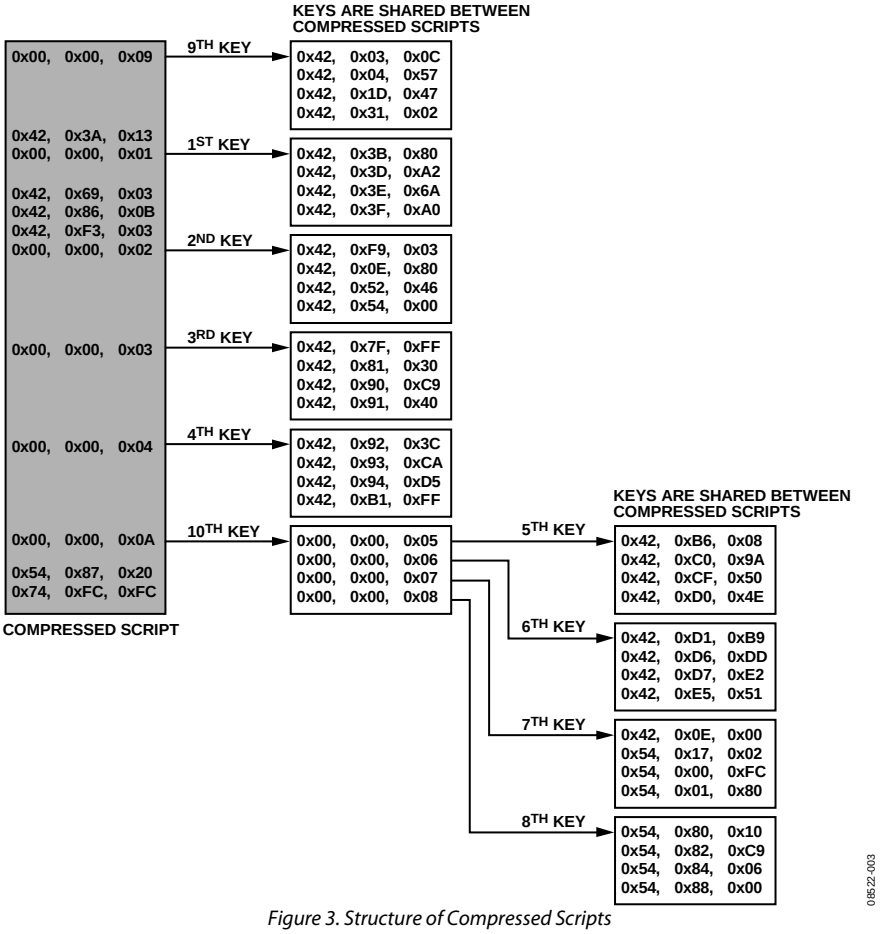

Figure 3. Structure of Compressed Scripts

Once the matrices are formatted and common keys are known, one can start creating C code and perform a small amount of post-optimization (post-compression). This post-compression will:

- write representative end of script code (0xFF), so that the decompressor can find the end of the script.
- determine if the key_number is smaller than 256. If so, its representing code (0x00, A, B) can be shorted to 0x00, B, because A is always 0.
- add C code for the decompressor to the end of the C code.

Finally, generated C code appears as shown in Figure 4.

USAGE

The user can use code given in the Octave Source Code section. It consists of six files that should be put in the same folder. Place the script.txtfile, that is, a set of scripts in a format described in the following section, in this folder. After running the script, create the output.c file containing the compressed script algorithm for decompression and an example of main code.

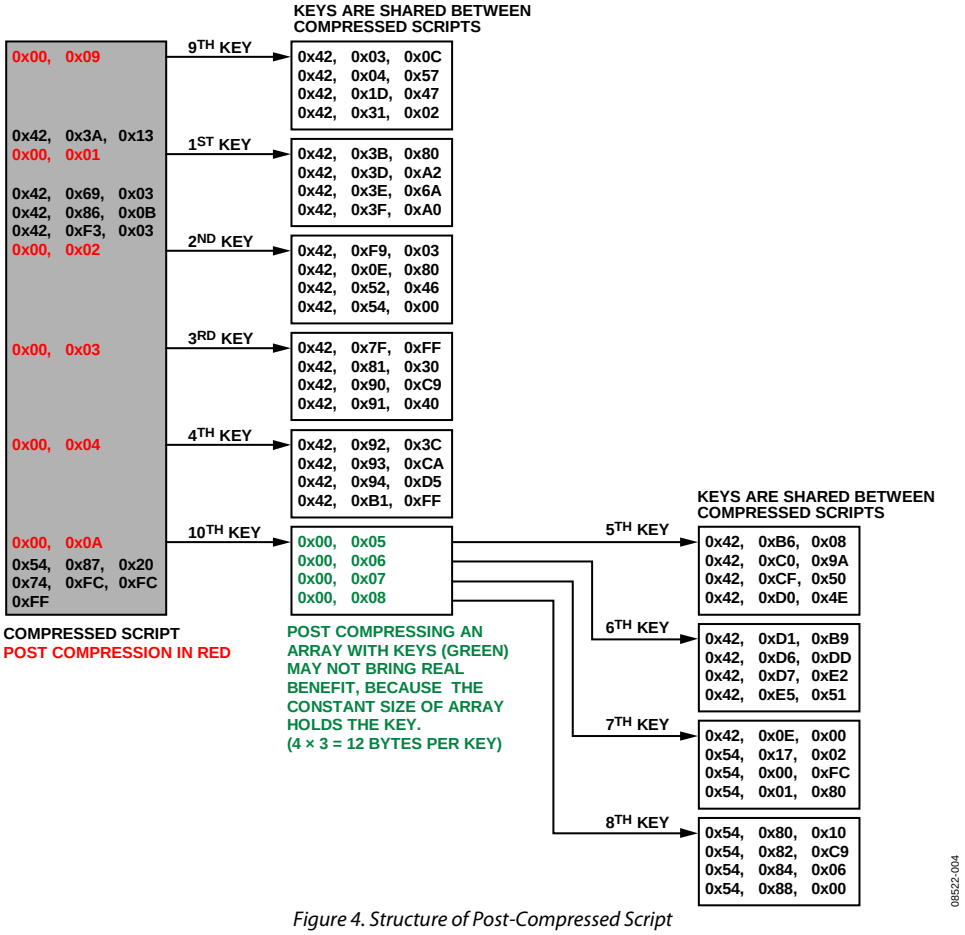

Figure 4. Structure of Post-Compressed Script

OCTAVE SOURCE CODE

All listings should be saved to one folder together with the script.txt file containing the script to be compressed. This code was tested using Octave Version 3.0.3.

MAIN.M

```
clear;

filename = 'script.txt';

%%%%%% loading the script:  %%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  number_of_scripts = number_of_scripts_in_file(filename);

  %%% load all scripts - script by script  %%
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  for i=1:number_of_scripts
    cmd = sprintf("[script_%d] = load_script(\"%s\", %d);", i, filename, i);
    eval(cmd);
  endfor
  printf("Scripts loaded...\r\n");
  fflush(stdout);

  %%%%%%% find same occurances in different scripts %%%%%%%
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  NUMBER_OF_LINES = 4;
  key_number = 0;

  do % master loop - runs the optimization until no improvement is done

    % global_found is a variable to show that any improvement was done in full run
    global_found = 0;

    for i=1:number_of_scripts
      i               % print the iteration
      fflush(stdout);  % update user's screen

      %% load variable current_script with matrix: %%
      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      eval(sprintf("current_script_a = script_%d;", i));
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% check if the script is empty or not: %%
dim = size(current_script_a);
if ((dim==[0,0]) || (dim(1) < NUMBER_OF_LINES))
  found = 0;
  continue;
endif
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[dim1, dim2] = size(current_script_a);  % check get size
start_line = 0;

%%%% looking for the same pattern across matrix %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
while (start_line + NUMBER_OF_LINES - 1 < dim1)
  start_line++;  % move the pointer
  found = 0;     % this pattern (KEY) is not found yet
  key_added = 0; % KEY: variable for maintaining keys (KEY not FOUND - so not added yet)

  % get part of matrix as a pattern to find in other scripts: %
  part_of_script_a = current_script_a(start_line:start_line+NUMBER_OF_LINES-1, :);

  %%%% search in other scripts for pattern %%%%
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  for j=i+1:number_of_scripts

    eval(sprintf("current_script_b = script_%d;", j));

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %% check if the script is empty or not: %%
    dim = size(current_script_b);
    if ((dim==[0,0]) || (dim(1) < NUMBER_OF_LINES))
      continue;
    endif
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    val = find_mat2(current_script_b, part_of_script_a);
    if (val != 0)

      %% AS PATTERN WAS FOUND IN THE OTHER SCRIPT - %
      %%%%%%%  - ADDING THE KEY TO KEY LIST: %%%%%%%%
      if (key_added == 0)
        key_added = 1; % KEY_ADDED
        key(++key_number, :, :) = part_of_script_a';
        this_key = [0, floor(key_number / 256), mod(key_number, 256)]; % MAX: 65535 KEYS!
```

```
            current_script_a = fsubst(current_script_a, this_key, start_line, \
                                  start_line+NUMBER_OF_LINES-1);

            eval(sprintf("script_%d = current_script_a;", i));
            [dim1, dim2] = size(current_script_a);
          end

          %% MODIFICATION OF THE SCRIPT %%
          current_script_b = fsubst(current_script_b, this_key, val, val+NUMBER_OF_LINES-1);
          eval(sprintf("script_%d = current_script_b;", j));

          %% ADDING COUNTER OF FOUND LINES
          found = found + 1;
          global_found = 1;
        endif
      endfor
    endwhile
  endfor
until (!global_found)

%%% export file to c-file %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
script_to_c;
```

LOAD_SCRIPT.M

```
function [matrix] = load_script(filename, script_number)
  % script_number - number of script (starts from 1) to get
  current_script_number = 0;
  matrix = [];

  if ((fhandle = fopen(filename, "r")) == -1)
    return;

  else
    started_script_line = 0;

    while (!feof(fhandle))
      text_line = fgetl(fhandle);

      if (strncmpi(text_line, "end", 3))
        started_script_line = 0;

        if (current_script_number == script_number)
          return; % needed script has been extracted
        endif
```

```
      elseif (started_script_line > 0)
            started_script_line = started_script_line + 1;
      endif

      %%% Is it first line of new script?: %%%
      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      if ((length(text_line) > 1) && (text_line(1) == "#") && (text_line(2) == "#"))

        started_script_line = 1;
        current_script_number = current_script_number + 1;
      endif

      %%% Is it next line of the same script?: %%%
      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      if ((started_script_line > 2) && (script_number == current_script_number))
        new_line = (sscanf(text_line, "%x %x %x"))'; % don't forget about '

        if (started_script_line == 3)
          matrix = new_line;       % creates matrix
        else
          matrix = [matrix; new_line];  % appends to already created matrix
        endif
      endif


    endwhile

    fclose (fhandle);
    return;


  endif
endfunction
```

NUMBER_OF_SCRIPTS_IN_FILE.M

```
function [number_of_scripts] = number_of_scripts_in_file(filename)

  if ((fhandle = fopen(filename, "r")) == -1)
    number_of_scripts = -1;
    fprintf(stdout, "Error while opening file!");
    return;
  else
    number_of_scripts = 0;

    while (!feof(fhandle))
```

```
      text_line = fgetl(fhandle);

      if (length(text_line > 1) && (text_line(1) == "#") && (text_line(2) == "#"))
        number_of_scripts = number_of_scripts + 1;
      endif
    endwhile

    fclose (fhandle);
    return;
  endif
endfunction
```

FSUBST.M

```
function [ret]=fsubst(matrix, sub, first_line, last_line)

  last_line = last_line + 1;
  [dim_a1, dim_a2] = size(matrix);

  ret = matrix(1:first_line-1, :);
  ret = [ret; sub];
  ret = [ret; matrix(last_line:dim_a1, :)];

endfunction
```

FIND_MAT2.M

```
function [res] = find_mat2(mat_a, mat_b)
%find_mat2 returns the first line where mat_b is located in mat_a

occurances_found = 0;

[dim_a1, dim_a2] = size(mat_a);
[dim_b1, dim_b2] = size(mat_b);

  res = 0;
% dim_a2
% dim_b2

  mat_a_expanded = mat_a'(:);
  mat_b_expanded = mat_b'(:);

len_b = length(mat_b_expanded);

% which one is bigger?

for (i=1:dim_a2:(dim_a1*dim_a2-dim_b1*dim_b2+1))

  difference = mat_a_expanded(i:i+len_b-1) - mat_b_expanded;
```

```
  % count number of zero columns
  number_of_matches = length(find(difference == 0));

  if (number_of_matches == len_b)
    %fprintf(stdout, "Offset = %d\r\n", i);
    res(++occurances_found) = ceil(i/dim_a2);
  endif

endfor


endfunction
```

SCRIPT_TO_C.M

```
  filename_output = "output.c"
  termination_characters = "        0xFF }; \r\n\r\n";

  %%% opening file to save %%%
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  if ((fhandle = fopen(filename_output, "w")) == -1)
    return;
  endif

  %%% SIZE OF KEY %%%
  %%%%%%%%%%%%%%%%%%%
  if (key_number > 0)
    [key_dim_a, key_dim_b, key_dim_c] = size(key(1, :, :));
    size_of_key = key_dim_a * key_dim_b * key_dim_c;
  else
    size_of_key = 0;
  end

  %%/* DEVCPP - INCLUDES */       %%
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  fprintf(fhandle, "#include <cstdlib>\r\n");
  fprintf(fhandle, "#include <iostream>\r\n");
  fprintf(fhandle, "using namespace std;\r\n");


  fprintf(fhandle, "#define DECOMPR_SIZE_OF_KEY    %d\r\n", size_of_key);
  fprintf(fhandle, "#define DECOMPR_NUMBER_OF_SCRIPTS %d\r\n", number_of_scripts);
  fprintf(fhandle, "#define DECOMPR_NUMBER_OF_KEYS    %d\r\n\r\n", key_number);

  fprintf(fhandle, "#define DECOMPR_ADDRESS_OR_ESCAPE   0\r\n");
  fprintf(fhandle, "#define DECOMPR_REGADDR             1\r\n");
  fprintf(fhandle, "#define DECOMPR_VALUE               2\r\n\r\n");
```

```
fprintf(fhandle, "#define DECOMPR_ESCAPE_CODE         0\r\n");
fprintf(fhandle, "#define DECOMPR_TERMINATE_VALUE     0xFF\r\n");
fprintf(fhandle, "#define DECOMPR_USES_2_BYTE_CODES   (DECOMPR_NUMBER_OF_KEYS > 254)\r\n");

for (i=1:number_of_scripts)
  eval(sprintf("current_script = script_%d;", i));

  fprintf(fhandle, "const unsigned char script_%d[] = { \r\n", i);
  [dim1, dim2] = size(current_script);

  for (line = 1:dim1)
    %% POST COMPRESSION / OPTIMIZATION: %%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    if ( (current_script(line, 1) == 0) && (key_number < 255) )
      fprintf(fhandle, "        0x%.2X, 0x%.2X,\r\n", \
              current_script(line, 1),                \
              current_script(line, 3));

    else
      fprintf(fhandle, "        0x%.2X, 0x%.2X, 0x%.2X,\r\n",
              current_script(line, 1),                \
              current_script(line, 2),                \
              current_script(line, 3));               \

    endif
  endfor

  fprintf(fhandle, termination_characters);
endfor

%%% LISTING THE KEYS  %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%
if (key_number > 0)
  fprintf(fhandle,                                     \
          "const unsigned char keys[DECOMPR_NUMBER_OF_KEYS][DECOMPR_SIZE_OF_KEY] = { \r\n");

endif
for (n=1:key_number)

  %%%% listing the key   %%%%
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%
  if ((key_number < 255) && (key(n, 1, 1) == 0))
    fprintf(fhandle, "        { 0x%.2X, 0x%.2X, \r\n", \
            key(n, 1, 1),                              \
            key(n, 3, 1));
```

```
  else
    fprintf(fhandle, "{ 0x%.2X, 0x%.2X, 0x%.2X, \r\n",  \
            key(n, 1, 1),                               \
            key(n, 2, 1),                               \
            key(n, 3, 1));

  endif

  for line=2:key_dim_c-1
    if ((key_number < 255) && (key(n, 1, line) == 0))
      fprintf(fhandle, "          0x%.2X, 0x%.2X, \r\n", \
              key(n, 1, line),                           \
              key(n, 3, line));
    else
      fprintf(fhandle, "          0x%.2X, 0x%.2X, 0x%.2X, \r\n", \
              key(n, 1, line),                                    \
              key(n, 2, line),                                    \
              key(n, 3, line));
    endif
  endfor

  if ((key_number < 255) && (key(n, 1, key_dim_c) == 0))
    fprintf(fhandle, "          0x%.2X, 0x%.2X }", \
            key(n, 1, key_dim_c),                  \
            key(n, 3, key_dim_c));
  else
    fprintf(fhandle, "          0x%.2X, 0x%.2X, 0x%.2X }", \
            key(n, 1, key_dim_c),                          \
            key(n, 2, key_dim_c),                          \
            key(n, 3, key_dim_c));
  endif

  if (n != key_number) % this is not the last key:
    fprintf(fhandle, ",\r\n",                       \
            key(n, line, 1),                        \
            key(n, line, 1),                        \
            key(n, line, 1));
  else
    fprintf(fhandle, "};\r\n\r\n",
            key(n, line, 1),                        \
            key(n, line, 1),                        \
            key(n, line, 1));
  endif
endfor


%%% LIST OF POINTERS TO SCRIPTS %%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
fprintf(fhandle,"const unsigned char *script_list[DECOMPR_NUMBER_OF_SCRIPTS] = {\r\n");

for (i=1:number_of_scripts-1)
  fprintf(fhandle, sprintf("      script_%d,\r\n", i));
endfor
i++;

fprintf(fhandle, sprintf("      script_%d\r\n", i)); % last script
fprintf(fhandle,"};\r\n\r\n");
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% EXAMPLE SEND_I2c_COMMAND: %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
fprintf(fhandle, "void send_i2c_command(unsigned char dev_addr, ");
fprintf(fhandle, "unsigned char reg_addr, unsigned char value) {\r\n");

fprintf(fhandle, "     printf(\"%%.2X %%.2X %%.2X ;\\r\\n\", dev_addr, reg_addr, value);\r\n");
fprintf(fhandle, "     return;\r\n");
fprintf(fhandle, "}\r\n\r\n");

%% DECOMPRESSING ALGORITHM: %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
fprintf(fhandle, "void decompress(const unsigned char *scr_addr, unsigned char ");
fprintf(fhandle, "is_key=0)\r\n{\r\n");

fprintf(fhandle, "     unsigned char dev_addr, reg_addr, value; // information ");
fprintf(fhandle, "that has to be passed to I2C function\r\n\r\n");

fprintf(fhandle, "     unsigned int char_type = DECOMPR_ADDRESS_OR_ESCAPE; ");
fprintf(fhandle, " // state machine var.\r\n");

fprintf(fhandle, "     unsigned char *script = (unsigned char *) scr_addr; // pointer\r\n");
fprintf(fhandle, "     unsigned int line_counter = 0; // used to terminate when ");
fprintf(fhandle, "processing of keys (as they do not have END sequence as 0xFF;\r\n\r\n");

fprintf(fhandle, "     do {\r\n");
fprintf(fhandle, "         switch (char_type) {\r\n");
fprintf(fhandle, "                case DECOMPR_ADDRESS_OR_ESCAPE: {\r\n");
fprintf(fhandle, "                       if ((*script == DECOMPR_TERMINATE_VALUE))\r\n");
fprintf(fhandle, "                          return;\r\n\r\n");

if (key_number > 0)
  fprintf(fhandle, "                        if ((*script) == DECOMPR_ESCAPE_CODE) {\r\n\r\n");
  fprintf(fhandle, "                           // two versions of code ");
  fprintf(fhandle, "(below) requires different offset of pointer:\r\n");
```

```
  fprintf(fhandle, "                          //    (0x00, A, B = 16-bit)\r\n");
  fprintf(fhandle, "                          //    (0x00, B = 8-bit code)\r\n");
  fprintf(fhandle, "                          if (DECOMPR_USES_2_BYTE_CODES) {\r\n");
  fprintf(fhandle, "                             decompress(keys[(*(script+1))*256 + ");
  fprintf(fhandle, " (*(script+2))-1], 1);\r\n");

  fprintf(fhandle, "                             script+=3; // shift the pointer\r\n");
  fprintf(fhandle, "                          } else {\r\n");
  fprintf(fhandle, "                             decompress(keys[(*(script+1))-1], 1);\r\n");
  fprintf(fhandle, "                             script+=2; // shift the pointer\r\n");
  fprintf(fhandle, "                          }\r\n\r\n");
  fprintf(fhandle, "                          line_counter++;\r\n");
  fprintf(fhandle, "                       } else {\r\n");
endif

fprintf(fhandle, "                           dev_addr = *script;\r\n");
fprintf(fhandle, "                           script++;\r\n");
fprintf(fhandle, "                           char_type = DECOMPR_REGADDR;\r\n");

if (key_number > 0)
  fprintf(fhandle, "                       }\r\n");
endif

fprintf(fhandle, "                          break;\r\n");
fprintf(fhandle, "                       }\r\n");
fprintf(fhandle, "                  case DECOMPR_REGADDR: {\r\n");
fprintf(fhandle, "                           reg_addr = *script;\r\n");
fprintf(fhandle, "                           script++;\r\n");
fprintf(fhandle, "                           char_type = DECOMPR_VALUE;\r\n");
fprintf(fhandle, "                           break;\r\n");
fprintf(fhandle, "                       }\r\n\r\n");
fprintf(fhandle, "                  case DECOMPR_VALUE: {\r\n");
fprintf(fhandle, "                           value = *script;\r\n");
fprintf(fhandle, "                           script++;\r\n");
fprintf(fhandle, "                           char_type = DECOMPR_ADDRESS_OR_ESCAPE;\r\n");
fprintf(fhandle, "                           \r\n");
fprintf(fhandle, "                           // all data should be ready to send:\r\n");
fprintf(fhandle, "                           send_i2c_command(dev_addr, reg_addr, \r\n");
fprintf(fhandle, "                                            value);\r\n");
fprintf(fhandle, "                           line_counter++;\r\n");
fprintf(fhandle, "                           break;\r\n");
fprintf(fhandle, "                       }\r\n\r\n");
fprintf(fhandle, "                  default: break;\r\n");
fprintf(fhandle, "            }\r\n");
fprintf(fhandle, "     } while (!(is_key && line_counter==4));\r\n}\r\n\r\n");

%% EXAMPLE MAIN %%
```

```
%%%%%%%%%%%%%%%%%%
fprintf(fhandle, "int main(int argc, char *argv[])\r\n{\r\n");
fprintf(fhandle, "    int i;\r\n\r\n");
fprintf(fhandle, "    for (i=0; i<DECOMPR_NUMBER_OF_SCRIPTS ; i++) {\r\n");
fprintf(fhandle, "        printf(\"## Scripts ##\\n:Script no.%%d:\\n\", i);\r\n");
fprintf(fhandle, "        decompress(script_list[i]);\r\n");
fprintf(fhandle, "        printf(\"End\\n\\n\");\r\n");
fprintf(fhandle, "    }\r\n\r\n");
fprintf(fhandle, "    system(\"PAUSE\");\r\n");
fprintf(fhandle, "    return EXIT_SUCCESS;\r\n}\r\n");


fclose(fhandle);
```

CONCLUSION

RESULTS

The compression algorithm discussed in this application note provides a

- 1.858 compression ratio for a set of 51 scripts for the ADV7401 evaluation board (4748 bytes were compressed to 2556)
- 5.056 compression ratio for a set of 251 scripts for the ADV7840 evaluation board (52,088 bytes were compressed to 10,268)

The compression ratio was calculated using the size of the ROM of the compiled program.

A KEIL compiler was used for testing with the Analog Devices ADuC7024 microcontroller.

FURTHER OPTIMIZATIONS

Users may want to optimize a given algorithm. Tips on how to do this are outlined here.

Usually very large scripts use 200 to 700 keys. Thus, instead of using three bytes for decoding the address (0x00, A, B), users may optimize this by simply substituting ESCAPE code so that:

- 0x00, A gives the address of the key between (0 + A)
- 0x01, A gives the address of the key between (256 + A)
- 0x03, A gives the address of the key between (512 + A)
- 0x05, A gives the address of the key between (768 + A)

Because these values (0x01, 0x03, 0x05) are odd, they cannot be used to store the address of the device.

Users may also want to use histograms to check values.

Various key lengths may also be used, however this requires significantly more microcontroller resources during decompression.

REFERENCES AND LICENSING INFORMATION

Octave is available for download free of cost by accessing the official GNU operating system website. This website also contains information about Octave licensing.

Users may want to check generated code with the integrated development environment for C/C++, like Dev-C++. This information, as well as licensing information, can be found on the official Bloodshed Dev-C++ integrated development environment (IDE) website.

NOTES

AN08522-0-12/09(0)