```
/*****************************************************************
 * File:    print_scan.c
 * Purpose: Implementation of debug_printf(), debug_scanf() functions.
 *
 * This is a modified version of the file printf.c, which was distributed
 * by Motorola as part of the M5407C3BOOT.zip package used to initialize
 * the M5407C3 evaluation board.
 *
 * Copyright:
 *      1999-2000 MOTOROLA, INC. All Rights Reserved.
 *  You are hereby granted a copyright license to use, modify, and
 *  distribute the SOFTWARE so long as this entire notice is
 *  retained without alteration in any modified and/or redistributed
 *  versions, and that such modified versions are clearly identified
 *  as such. No licenses are granted by implication, estoppel or
 *  otherwise under any patents or trademarks of Motorola, Inc. This
 *  software is provided on an "AS IS" basis and without warranty.
 *
 *  To the maximum extent permitted by applicable law, MOTOROLA
 *  DISCLAIMS ALL WARRANTIES WHETHER EXPRESS OR IMPLIED, INCLUDING
 *  IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR
 *  PURPOSE AND ANY WARRANTY AGAINST INFRINGEMENT WITH REGARD TO THE
 *  SOFTWARE (INCLUDING ANY MODIFIED VERSIONS THEREOF) AND ANY
 *  ACCOMPANYING WRITTEN MATERIALS.
 *
 *  To the maximum extent permitted by applicable law, IN NO EVENT
 *  SHALL MOTOROLA BE LIABLE FOR ANY DAMAGES WHATSOEVER (INCLUDING
 *  WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS
 *  INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR OTHER PECUNIARY
 *  LOSS) ARISING OF THE USE OR INABILITY TO USE THE SOFTWARE.
 *
 *  Motorola assumes no responsibility for the maintenance and support
 *  of this software
 ******************************************************************/

#include "print_scan.h"
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdint.h>
#include <stdbool.h>
// Keil: suppress ellipsis warning in va_arg usage below
#if defined(__CC_ARM)
#pragma diag_suppress 1256
#endif

#define FLAGS_MINUS     (0x01)
#define FLAGS_PLUS      (0x02)
#define FLAGS_SPACE     (0x04)
#define FLAGS_ZERO      (0x08)
#define FLAGS_POUND     (0x10)

#define IS_FLAG_MINUS(a)    (a & FLAGS_MINUS)
#define IS_FLAG_PLUS(a)     (a & FLAGS_PLUS)
#define IS_FLAG_SPACE(a)    (a & FLAGS_SPACE)
#define IS_FLAG_ZERO(a)     (a & FLAGS_ZERO)
#define IS_FLAG_POUND(a)    (a & FLAGS_POUND)

#define LENMOD_h        (0x01)
#define LENMOD_l        (0x02)
#define LENMOD_L        (0x04)
#define LENMOD_hh       (0x08)
#define LENMOD_ll       (0x10)

#define IS_LENMOD_h(a)  (a & LENMOD_h)
#define IS_LENMOD_hh(a) (a & LENMOD_hh)
#define IS_LENMOD_l(a)  (a & LENMOD_l)
```

```
#define IS_LENMOD_ll(a) (a & LENMOD_ll)
#define IS_LENMOD_L(a)  (a & LENMOD_L)

#define SCAN_SUPPRESS             0x2

#define SCAN_DEST_MASK            0x7c
#define SCAN_DEST_CHAR            0x4
#define SCAN_DEST_STRING          0x8
#define SCAN_DEST_SET             0x10
#define SCAN_DEST_INT             0x20
#define SCAN_DEST_FLOAT           0x30

#define SCAN_LENGTH_MASK          0x1f00
#define SCAN_LENGTH_CHAR          0x100
#define SCAN_LENGTH_SHORT_INT     0x200
#define SCAN_LENGTH_LONG_INT      0x400
#define SCAN_LENGTH_LONG_LONG_INT 0x800
#define SCAN_LENGTH_LONG_DOUBLE   0x1000

#define SCAN_TYPE_SIGNED          0x2000

/*!
 * @brief Scanline function which ignores white spaces.
 *
 * @param[in] s The address of the string pointer to update.
 *
 * @return String without white spaces.
 */
static uint32_t scan_ignore_white_space(const char **s);

#if defined(SCANF_FLOAT_ENABLE)
static double fnum = 0.0;
#endif

/*!
 * @brief Converts a radix number to a string and return its length.
 *
 * @param[in] numstr    Converted string of the number.
 * @param[in] nump      Pointer to the number.
 * @param[in] neg       Polarity of the number.
 * @param[in] radix     The radix to be converted to.
 * @param[in] use_caps  Used to identify %x/X output format.

 * @return Length of the converted string.
 */
static int32_t mknumstr (char *numstr, void *nump, int32_t neg, int32_t radix, bool use_caps);

#if defined(PRINTF_FLOAT_ENABLE)
/*!
 * @brief Converts a floating radix number to a string and return its length.
 *
 * @param[in] numstr          Converted string of the number.
 * @param[in] nump            Pointer to the number.
 * @param[in] radix           The radix to be converted to.
 * @param[in] precision_width   Specify the precision width.

 * @return Length of the converted string.
 */
static int32_t mkfloatnumstr (char *numstr, void *nump, int32_t radix, uint32_t precision_width);
#endif


static void fput_pad(int32_t c, int32_t curlen, int32_t field_width, int32_t *count, PUTCHAR_FUNC func_ptr, void *farg, int
*max_count);

double modf(double input_dbl, double  *intpart_ptr);
```

```
#if !defined(PRINT_MAX_COUNT)
#define n_putchar(func, chacter, p, count)      func(chacter, p)
#else
static int n_putchar(PUTCHAR_FUNC func_ptr, int chacter, void *p, int *max_count)
{
    int result = 0;
    if (*max_count)
    {
        result = func_ptr(chacter, p);
        (*max_count)--;
    }
    return result;
}
#endif

/*FUNCTION*****************************************************************
 *
 * Function Name : _doprint
 * Description   : This function outputs its parameters according to a
 * formatted string. I/O is performed by calling given function pointer
 * using following (*func_ptr)(c,farg);
 *
 *END*********************************************************************/
int _doprint(void *farg, PUTCHAR_FUNC func_ptr, int max_count, char *fmt, va_list ap)
{
    /* va_list ap; */
    char *p;
    int32_t c;

    char vstr[33];
    char *vstrp;
    int32_t vlen;

    int32_t done;
    int32_t count = 0;
    int temp_count = max_count;


    uint32_t flags_used;
    uint32_t field_width;

    int32_t ival;
    int32_t schar, dschar;
    int32_t *ivalp;
    char *sval;
    int32_t cval;
    uint32_t uval;
    bool use_caps;
    uint32_t precision_width;
    //uint32_t length_modifier = 0;
#if defined(PRINTF_FLOAT_ENABLE)
    double fval;
#endif

    if (max_count == -1)
    {
        max_count = INT32_MAX - 1;
    }

    /*
     * Start parsing apart the format string and display appropriate
     * formats and data.
     */
    for (p = (char *)fmt; (c = *p) != 0; p++)
    {
        /*
         * All formats begin with a '%' marker.  Special chars like
```

```
     * '\n' or '\t' are normally converted to the appropriate
     * character by the __compiler__.  Thus, no need for this
     * routine to account for the '\' character.
     */
    if (c != '%')
    {
        n_putchar(func_ptr, c, farg, &max_count);

        count++;

        /*
         * By using 'continue', the next iteration of the loop
         * is used, skipping the code that follows.
         */
        continue;
    }

    /*
     * First check for specification modifier flags.
     */
    use_caps = true;
    flags_used = 0;
    done = false;
    while (!done)
    {
        switch (/* c = */*++p)
        {
            case '-':
                flags_used |= FLAGS_MINUS;
                break;
            case '+':
                flags_used |= FLAGS_PLUS;
                break;
            case ' ':
                flags_used |= FLAGS_SPACE;
                break;
            case '0':
                flags_used |= FLAGS_ZERO;
                break;
            case '#':
                flags_used |= FLAGS_POUND;
                break;
            default:
                /* we've gone one char too far */
                --p;
                done = true;
                break;
        }
    }

    /*
     * Next check for minimum field width.
     */
    field_width = 0;
    done = false;
    while (!done)
    {
        switch (c = *++p)
        {
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
```

```
        case '7':
        case '8':
        case '9':
            field_width = (field_width * 10) + (c - '0');
            break;
        default:
            /* we've gone one char too far */
            --p;
            done = true;
            break;
    }
}

/*
 * Next check for the width and precision field separator.
 */
precision_width = 6;
if (/* (c = *++p) */*++p == '.')
{
    /* precision_used = true; */

    /*
     * Must get precision field width, if present.
     */
    precision_width = 0;
    done = false;
    while (!done)
    {
        switch (c = *++p)
        {
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
                precision_width = (precision_width * 10) + (c - '0');
                break;
            default:
                /* we've gone one char too far */
                --p;
                done = true;
                break;
        }
    }
}
else
{
    /* we've gone one char too far */
    --p;
}

/*
 * Check for the length modifier.
 */
/* length_modifier = 0; */
switch (/* c = */*++p)
{
    case 'h':
        if (*++p != 'h')
        {
```

```
            --p;
        }
        /* length_modifier |= LENMOD_h; */
        break;
    case 'l':
        if (*++p != 'l')
        {
            --p;
        }
        /* length_modifier |= LENMOD_l; */
        break;
    case 'L':
        /* length_modifier |= LENMOD_L; */
        break;
    default:
        /* we've gone one char too far */
        --p;
        break;
}

/*
 * Now we're ready to examine the format.
 */
switch (c = *++p)
{
    case 'd':
    case 'i':
        ival = (int32_t)va_arg(ap, int32_t);
        vlen = mknumstr(vstr,&ival,true,10,use_caps);
        vstrp = &vstr[vlen];

        if (ival < 0)
        {
            schar = '-';
            ++vlen;
        }
        else
        {
            if (IS_FLAG_PLUS(flags_used))
            {
                schar = '+';
                ++vlen;
            }
            else
            {
                if (IS_FLAG_SPACE(flags_used))
                {
                    schar = ' ';
                    ++vlen;
                }
                else
                {
                    schar = 0;
                }
            }
        }
        dschar = false;

        /*
         * do the ZERO pad.
         */
        if (IS_FLAG_ZERO(flags_used))
        {
            if (schar)
            {
                n_putchar(func_ptr, schar, farg, &max_count);
```

```
                count++;
            }
            dschar = true;

            fput_pad('0', vlen, field_width, &count, func_ptr, farg, &max_count);
            vlen = field_width;
        }
        else
        {
            if (!IS_FLAG_MINUS(flags_used))
            {
                fput_pad(' ', vlen, field_width, &count, func_ptr, farg, &max_count);
                if (schar)
                {
                    n_putchar(func_ptr, schar, farg, &max_count);
                    count++;
                }
                dschar = true;
            }
        }

        /* the string was built in reverse order, now display in */
        /* correct order */
        if ((!dschar) && schar)
        {
            n_putchar(func_ptr, schar, farg, &max_count);
            count++;
        }
        goto cont_xd;
#if defined(PRINTF_FLOAT_ENABLE)
    case 'f':
    case 'F':
        fval = (double)va_arg(ap, double);
        vlen = mkfloatnumstr(vstr,&fval,10, precision_width);
        vstrp = &vstr[vlen];

        if (fval < 0)
        {
            schar = '-';
            ++vlen;
        }
        else
        {
            if (IS_FLAG_PLUS(flags_used))
            {
                schar = '+';
                ++vlen;
            }
            else
            {
                if (IS_FLAG_SPACE(flags_used))
                {
                    schar = ' ';
                    ++vlen;
                }
                else
                {
                    schar = 0;
                }
            }
        }
        dschar = false;
        if (IS_FLAG_ZERO(flags_used))
        {
            if (schar)
            {
```

```
                n_putchar(func_ptr, schar, farg, &max_count);
                count++;
            }
            dschar = true;
            fput_pad('0', vlen, field_width, &count, func_ptr, farg, &max_count);
            vlen = field_width;
        }
        else
        {
            if (!IS_FLAG_MINUS(flags_used))
            {
                fput_pad(' ', vlen, field_width, &count, func_ptr, farg, &max_count);
                if (schar)
                {
                    n_putchar(func_ptr, schar, farg, &max_count);
                    count++;
                }
                dschar = true;
            }
        }
        if (!dschar && schar)
        {
            n_putchar(func_ptr, schar, farg, &max_count);
            count++;
        }
        goto cont_xd;
#endif
    case 'x':
        use_caps = false;
    case 'X':
        uval = (uint32_t)va_arg(ap, uint32_t);
        vlen = mknumstr(vstr,&uval,false,16,use_caps);
        vstrp = &vstr[vlen];

        dschar = false;
        if (IS_FLAG_ZERO(flags_used))
        {
            if (IS_FLAG_POUND(flags_used))
            {
                n_putchar(func_ptr, '0', farg, &max_count);
                n_putchar(func_ptr, (use_caps ? 'X' : 'x'), farg, &max_count);
                count += 2;
                /*vlen += 2;*/
                dschar = true;
            }
            fput_pad('0', vlen, field_width, &count, func_ptr, farg, &max_count);
            vlen = field_width;
        }
        else
        {
            if (!IS_FLAG_MINUS(flags_used))
            {
                if (IS_FLAG_POUND(flags_used))
                {
                    vlen += 2;
                }
                fput_pad(' ', vlen, field_width, &count, func_ptr, farg, &max_count);
                if (IS_FLAG_POUND(flags_used))
                {
                    n_putchar(func_ptr, '0', farg, &max_count);
                    n_putchar(func_ptr, (use_caps ? 'X' : 'x'), farg, &max_count);
                    count += 2;

                    dschar = true;
                }
            }
```

```
    }

    if ((IS_FLAG_POUND(flags_used)) && (!dschar))
    {
        n_putchar(func_ptr, '0', farg, &max_count);
        n_putchar(func_ptr, (use_caps ? 'X' : 'x'), farg, &max_count);
        count += 2;
        vlen += 2;
    }
    goto cont_xd;

case 'o':
    uval = (uint32_t)va_arg(ap, uint32_t);
    vlen = mknumstr(vstr,&uval,false,8,use_caps);
    goto cont_u;
case 'b':
    uval = (uint32_t)va_arg(ap, uint32_t);
    vlen = mknumstr(vstr,&uval,false,2,use_caps);
    goto cont_u;
case 'p':
    uval = (uint32_t)va_arg(ap, uint32_t);
    uval = (uint32_t)va_arg(ap, void *);
    vlen = mknumstr(vstr,&uval,false,16,use_caps);
    goto cont_u;
case 'u':
    uval = (uint32_t)va_arg(ap, uint32_t);
    vlen = mknumstr(vstr,&uval,false,10,use_caps);

    cont_u:
        vstrp = &vstr[vlen];

        if (IS_FLAG_ZERO(flags_used))
        {
            fput_pad('0', vlen, field_width, &count, func_ptr, farg, &max_count);
            vlen = field_width;
        }
        else
        {
            if (!IS_FLAG_MINUS(flags_used))
            {
                fput_pad(' ', vlen, field_width, &count, func_ptr, farg, &max_count);
            }
        }

    cont_xd:
        while (*vstrp)
        {
            n_putchar(func_ptr, *vstrp--, farg, &max_count);
            count++;
        }

        if (IS_FLAG_MINUS(flags_used))
        {
            fput_pad(' ', vlen, field_width, &count, func_ptr, farg, &max_count);
        }
    break;

case 'c':
    cval = (char)va_arg(ap, uint32_t);
    n_putchar(func_ptr, cval, farg, &max_count);
    count++;
    break;
case 's':
    sval = (char *)va_arg(ap, char *);
    if (sval)
    {
```

```
                vlen = strlen(sval);
                if (!IS_FLAG_MINUS(flags_used))
                {
                    fput_pad(' ', vlen, field_width, &count, func_ptr, farg, &max_count);
                }
                while (*sval)
                {
                    n_putchar(func_ptr, *sval++, farg, &max_count);
                    count++;
                }
                if (IS_FLAG_MINUS(flags_used))
                {
                    fput_pad(' ', vlen, field_width, &count, func_ptr, farg, &max_count);
                }
            }
            break;
        case 'n':
            ivalp = (int32_t *)va_arg(ap, int32_t *);
            *ivalp = count;
            break;
        default:
            n_putchar(func_ptr, c, farg, &max_count);
            count++;
            break;
      }
   }

   if (max_count)
   {
      return count;
   }
   else
   {
      return temp_count;
   }
}

/*FUNCTION*****************************************************************
 *
 * Function Name : _sputc
 * Description   : Writes the character into the string located by the string
 * pointer and updates the string pointer.
 *
 *END*****************************************************************/
int _sputc(int c, void * input_string)
{
   char **string_ptr = (char  **)input_string;

   *(*string_ptr)++ = (char)c;
   return c;
}

/*FUNCTION*****************************************************************
 *
 * Function Name : mknumstr
 * Description   : Converts a radix number to a string and return its length.
 *
 *END*****************************************************************/
static int32_t mknumstr (char *numstr, void *nump, int32_t neg, int32_t radix, bool use_caps)
{
    int32_t a,b,c;
    uint32_t ua,ub,uc;

    int32_t nlen;
    char *nstrp;
```

```
    nlen = 0;
    nstrp = numstr;
    *nstrp++ = '\0';

    if (neg)
    {
        a = *(int32_t *)nump;
        if (a == 0)
        {
            *nstrp = '0';
            ++nlen;
            goto done;
        }
        while (a != 0)
        {
            b = (int32_t)a / (int32_t)radix;
            c = (int32_t)a - ((int32_t)b * (int32_t)radix);
            if (c < 0)
            {
                c = ~c + 1 + '0';
            }
            else
            {
                c = c + '0';
            }
            a = b;
            *nstrp++ = (char)c;
            ++nlen;
        }
    }
    else
    {
        ua = *(uint32_t *)nump;
        if (ua == 0)
        {
            *nstrp = '0';
            ++nlen;
            goto done;
        }
        while (ua != 0)
        {
            ub = (uint32_t)ua / (uint32_t)radix;
            uc = (uint32_t)ua - ((uint32_t)ub * (uint32_t)radix);
            if (uc < 10)
            {
                uc = uc + '0';
            }
            else
            {
                uc = uc - 10 + (use_caps ? 'A' : 'a');
            }
            ua = ub;
            *nstrp++ = (char)uc;
            ++nlen;
        }
    }
    done:
    return nlen;
}

#if defined(PRINTF_FLOAT_ENABLE)
/*FUNCTION**********************************************************************
 *
 * Function Name : mkfloatnumstr
 * Description   : Converts a floating radix number to a string and return
 * its length, user can specify output precision width.
```

```
 *
 *END**************************************************************************/
static int32_t mkfloatnumstr (char *numstr, void *nump, int32_t radix, uint32_t precision_width)
{
    int32_t a,b,c,i;
    double fa,fb;
    double r, fractpart, intpart;

    int32_t nlen;
    char *nstrp;
    nlen = 0;
    nstrp = numstr;
    *nstrp++ = '\0';
    r = *(double *)nump;
    if (r == 0)
    {
        *nstrp = '0';
        ++nlen;
        goto done;
    }
    fractpart = modf((double)r , (double *)&intpart);
    /* Process fractional part */
    for (i = 0; i < precision_width; i++)
    {
        fractpart *= radix;
    }
    //a = (int32_t)floor(fractpart + (double)0.5);
    fa = fractpart + (double)0.5;
    for (i = 0; i < precision_width; i++)
    {
        fb = fa / (int32_t)radix;
        c = (int32_t)(fa - (uint64_t)fb * (int32_t)radix);
        if (c < 0)
        {
            c = ~c + 1 + '0';
        }else
        {
            c = c + '0';
        }
        fa = fb;
        *nstrp++ = (char)c;
        ++nlen;
    }
    *nstrp++ = (char)'.';
    ++nlen;
    a = (int32_t)intpart;
    while (a != 0)
    {
        b = (int32_t)a / (int32_t)radix;
        c = (int32_t)a - ((int32_t)b * (int32_t)radix);
        if (c < 0)
        {
            c = ~c + 1 + '0';
        }else
        {
            c = c + '0';
        }
        a = b;
        *nstrp++ = (char)c;
        ++nlen;
    }
    done:
    return nlen;
}
#endif
```

```
static void fput_pad(int32_t c, int32_t curlen, int32_t field_width, int32_t *count, PUTCHAR_FUNC func_ptr, void *farg, int
*max_count)
{
   int32_t i;

   for (i = curlen; i < field_width; i++)
   {
      func_ptr((char)c, farg);
      (*count)++;
   }
}

/*FUNCTION*****************************************************************
 *
 * Function Name : scan_prv
 * Description   : Converts an input line of ASCII characters based upon a
 * provided string format.
 *
 *END*********************************************************************/
int scan_prv(const char *line_ptr, char *format, va_list args_ptr)
{
   uint8_t base;
   /* Identifier for the format string */
   char *c = format;
   const char *s;
   char temp;
   /* Identifier for the input string */
   const char *p = line_ptr;
   /* flag telling the conversion specification */
   uint32_t flag = 0 ;
   /* filed width for the matching input streams */
   uint32_t field_width;
   /* how many arguments are assigned except the suppress */
   uint32_t nassigned = 0;
   /* how many characters are read from the input streams */
   uint32_t n_decode = 0;

   int32_t val;
   char *buf;
   int8_t neg;

   /* return EOF error before any convernsion */
   if (*p == '\0')
   {
      return EOF;
   }

   /* decode directives */
   while ((*c) && (*p))
   {
      /* ignore all white-spaces in the format strings */
      if (scan_ignore_white_space((const char **)&c))
      {
         n_decode += scan_ignore_white_space(&p);
      }
      else if (*c != '%')
      {
         /* Ordinary characters */
         c++;
ordinary:   if (*p == *c)
         {
            n_decode++;
            p++;
            c++;
         }
         else
```

```
        {
            /* Match failure. Misalignment with C99, the unmatched
             * characters need to be pushed back to stream. HOwever
             * , it is deserted now. */
            break;
        }
    }
    else
    {
        /* convernsion specification */
        c++;
        if (*c == '%')
        {
            goto ordinary;
        }

        /* Reset */
        flag = 0;
        field_width = 0;
        base = 0;

        /* Loop to get full conversion specification */
        while ((*c) && (!(flag & SCAN_DEST_MASK)))
        {
            switch (*c)
            {
                case '*':
                    if (flag & SCAN_SUPPRESS)
                    {
                        /* Match failure*/
                        return nassigned;
                    }
                    flag |= SCAN_SUPPRESS;
                    c++;
                    break;
                case 'h':
                    if (flag & SCAN_LENGTH_MASK)
                    {
                        /* Match failure*/
                        return nassigned;
                    }
                    flag |= SCAN_LENGTH_SHORT_INT;

                    if (c[1] == 'h')
                    {
                        flag |= SCAN_LENGTH_CHAR;
                        c++;
                    }
                    c++;
                    break;
                case 'l':
                    if (flag & SCAN_LENGTH_MASK)
                    {
                        /* Match failure*/
                        return nassigned;
                    }
                    flag |= SCAN_LENGTH_LONG_INT;

                    if (c[1] == 'l')
                    {
                        flag |= SCAN_LENGTH_LONG_LONG_INT;
                        c++;
                    }
                    c++;
                    break;
#if defined(ADVANCE)
```

```
                case 'j':
                    if (flag & SCAN_LENGTH_MASK)
                    {
                        /* Match failure*/
                        return nassigned;
                    }
                    flag |= SCAN_LENGTH_INTMAX;
                    c++
                case 'z'
                    if (flag & SCAN_LENGTH_MASK)
                    {
                        /* Match failure*/
                        return nassigned;
                    }
                    flag |= SCAN_LENGTH_SIZE_T;
                    c++;
                    break;
                case 't':
                    if (flag & SCAN_LENGTH_MASK)
                    {
                        /* Match failure*/
                        return nassigned;
                    }
                    flag |= SCAN_LENGTH_PTRDIFF_T;
                    c++;
                    break;
#endif
#if defined(SCANF_FLOAT_ENABLE)
                case 'L':
                    if (flag & SCAN_LENGTH_MASK)
                    {
                        /* Match failure*/
                        return nassigned;
                    }
                    flag |= SCAN_LENGTH_LONG_DOUBLE;
                    c++;
                    break;
#endif
                case '0':
                case '1':
                case '2':
                case '3':
                case '4':
                case '5':
                case '6':
                case '7':
                case '8':
                case '9':
                    if (field_width)
                    {
                        /* Match failure*/
                        return nassigned;
                    }
                    do {
                        field_width = field_width * 10 + *c - '0';
                        c++;
                    } while ((*c >= '0') && (*c <= '9'));
                    break;
                case 'd':
                    flag |= SCAN_TYPE_SIGNED;
                case 'u':
                    base = 10;
                    flag |= SCAN_DEST_INT;
                    c++;
                    break;
```

```
                case 'o':
                    base = 8;
                    flag |= SCAN_DEST_INT;
                    c++;
                    break;
                case 'x':
                case 'X':
                    base = 16;
                    flag |= SCAN_DEST_INT;
                    c++;
                    break;
                case 'i':
                    base = 0;
                    flag |= SCAN_DEST_INT;
                    c++;
                    break;
#if defined(SCANF_FLOAT_ENABLE)
                case 'a':
                case 'A':
                case 'e':
                case 'E':
                case 'f':
                case 'F':
                case 'g':
                case 'G':
                    flag |= SCAN_DEST_FLOAT;
                    c++;
                    break;
#endif
                case 'c':
                    flag |= SCAN_DEST_CHAR;
                    if (!field_width)
                    {
                        field_width = 1;
                    }
                    c++;
                    break;
                case 's':
                    flag |= SCAN_DEST_STRING;
                    c++;
                    break;
#if defined(ADVANCE) /* [x]*/
                case '[':
                    flag |= SCAN_DEST_SET;
                    /*Add Set functionality */
                    break;
#endif
                default:
#if defined(SCAN_DEBUG)
                    printf("Unrecognized expression specifier: %c format: %s, number is: %d\r\n", c, format, nassigned);
#endif
                    return nassigned;
            }
        }

        if (!(flag & SCAN_DEST_MASK))
        {
            /* Format strings are exausted */
            return nassigned;
        }

        if (!field_width)
        {
            /* Larget then length of a line */
            field_width = 99;
```

```
    }

    /* Matching strings in input streams and assign to argument */
    switch (flag & SCAN_DEST_MASK)
    {
        case SCAN_DEST_CHAR:
            s = (const char *)p;
            buf = va_arg(args_ptr, char *);
            while ((field_width--) && (*p))
            {
                if (!(flag & SCAN_SUPPRESS))
                {
                    *buf++ = *p++;
                }
                else
                {
                    p++;
                }
                n_decode++;
            }

            if (((!(flag)) & SCAN_SUPPRESS) && (s != p))
            {
                nassigned++;
            }
            break;
        case SCAN_DEST_STRING:
            n_decode += scan_ignore_white_space(&p);
            s = p;
            buf = va_arg(args_ptr, char *);
            while ((field_width--) && (*p != '\0') && (*p != ' ') &&
                   (*p != '\t') && (*p != '\n') && (*p != '\r') && (*p != '\v') && (*p != '\f'))
            {
                if (flag & SCAN_SUPPRESS)
                {
                    p++;
                }
                else
                {
                    *buf++ = *p++;
                }
                n_decode++;
            }

            if ((!(flag & SCAN_SUPPRESS)) && (s != p))
            {
                /* Add NULL to end of string */
                *buf = '\0';
                nassigned++;
            }
            break;
        case SCAN_DEST_INT:
            n_decode += scan_ignore_white_space(&p);
            s = p;
            val = 0;
            /*TODO: scope is not testsed */
            if ((base == 0) || (base == 16))
            {
                if ((s[0] == '0') && ((s[1] == 'x') || (s[1] == 'X')))
                {
                    base = 16;
                    if (field_width >= 1)
                    {
                        p += 2;
                        n_decode += 2;
                        field_width -= 2;
```

```
        }
    }
}

if (base == 0)
{
    if (s[0] == '0')
    {
        base = 8;
    }
    else
    {
        base = 10;
    }
}

neg = 1;
switch (*p)
{
    case '-':
        neg = -1;
        n_decode++;
        p++;
        field_width--;
        break;
    case '+':
        neg = 1;
        n_decode++;
        p++;
        field_width--;
        break;
    default:
        break;
}

while ((*p) && (field_width--))
{
    if ((*p <= '9') && (*p >= '0'))
    {
        temp = *p - '0';
    }
    else if((*p <= 'f') && (*p >= 'a'))
    {
        temp = *p - 'a' + 10;
    }
    else if((*p <= 'F') && (*p >= 'A'))
    {
        temp = *p - 'A' + 10;
    }
    else
    {
        break;
    }

    if (temp >= base)
    {
        break;
    }
    else
    {
        val = base * val + temp;
    }
    p++;
    n_decode++;
}
```

```
            val *= neg;
            if (!(flag & SCAN_SUPPRESS))
            {
               switch (flag & SCAN_LENGTH_MASK)
               {
                  case SCAN_LENGTH_CHAR:
                     if (flag & SCAN_TYPE_SIGNED)
                     {
                        *va_arg(args_ptr, signed char *) = (signed char)val;
                     }
                     else
                     {
                        *va_arg(args_ptr, unsigned char *) = (unsigned char)val;
                     }
                     break;
                  case SCAN_LENGTH_SHORT_INT:
                     if (flag & SCAN_TYPE_SIGNED)
                     {
                        *va_arg(args_ptr, signed short *) = (signed short)val;
                     }
                     else
                     {
                        *va_arg(args_ptr, unsigned short *) = (unsigned short)val;
                     }
                     break;
                  case SCAN_LENGTH_LONG_INT:
                     if (flag & SCAN_TYPE_SIGNED)
                     {
                        *va_arg(args_ptr, signed long int *) = (signed long int)val;
                     }
                     else
                     {
                        *va_arg(args_ptr, unsigned long int *) = (unsigned long int)val;
                     }
                     break;
                  case SCAN_LENGTH_LONG_LONG_INT:
                     if (flag & SCAN_TYPE_SIGNED)
                     {
                        *va_arg(args_ptr, signed long long int *) = (signed long long int)val;
                     }
                     else
                     {
                        *va_arg(args_ptr, unsigned long long int *) = (unsigned long long int)val;
                     }
                     break;
                  default:
                     /* The default type is the type int */
                     if (flag & SCAN_TYPE_SIGNED)
                     {
                        *va_arg(args_ptr, signed int *) = (signed int)val;
                     }
                     else
                     {
                        *va_arg(args_ptr, unsigned int *) = (unsigned int)val;
                     }
                     break;
               }
               nassigned++;
            }
            break;
#if defined(SCANF_FLOAT_ENABLE)
         case SCAN_DEST_FLOAT:
            n_decode += scan_ignore_white_space(&p);
            fnum = strtod(p, (char **)&s);

            if ((fnum == HUGE_VAL) || (fnum == -HUGE_VAL))
```

```
                {
                    break;
                }

                n_decode += (int)(s) - (int)(p);
                p = s;
                if (!(flag & SCAN_SUPPRESS))
                {
                    if (flag & SCAN_LENGTH_LONG_DOUBLE)
                    {
                        *va_arg(args_ptr, double *) = fnum;
                    }
                    else
                    {
                        *va_arg(args_ptr, float *) = (float)fnum;
                    }
                    nassigned++;
                }
                break;
#endif
#if defined(ADVANCE)
            case SCAN_DEST_SET:
                break;
#endif
            default:
#if defined(SCAN_DEBUG)
                printf("ERROR: File %s  line: %d\r\n", __FILE__, __LINE__);
#endif
                return nassigned;
        }
    }
  }
  return nassigned;
}

/*FUNCTION*****************************************************************
 *
 * Function Name : scan_ignore_white_space
 * Description   : Scanline function which ignores white spaces.
 *
 *END*********************************************************************/
static uint32_t scan_ignore_white_space(const char **s)
{
  uint8_t count = 0;
  uint8_t c;

  c = **s;
  while ((c == ' ') || (c == '\t') || (c == '\n') || (c == '\r') || (c == '\v') || (c == '\f'))
  {
    count++;
    (*s)++;
    c = **s;
  }
  return count;
}
```