Mawk Printf
rcs-version `1.5  2016/07/25`

Table of Contents

1. Introduction. This is the source and documentation for a new `mawk` implementation of `s?printf()`. The purpose of the reimplementation is,

(1) Remove the use of a fixed buffer for `sprintf()`.

(2) Handle null character, `'\0'`, in the format string and the argument strings.

(3) For integer formats, e.g. `%d`, handle bigger integers. The original implementation was restricted to 32-bit integers. This implementation uses 64-bit integers.

(4) When the format string is a constant, which is the usual case, parse it at compile-time as opposed to run-time. This removes some redundant computation at run-time.

2. The Interface. To execute `printf(form,...)` or `sprintf(form,...)`, a call is made to `parse_form(form)`, which returns a `const Form*`. To the caller, this is an abstract type.

⟨ `printf` definitions 1a ⟩≡
```
typedef struct form Form ;
const Form* parse_form(const STRING*) ;
```
See also 1b, 11b and 21d.

This code is used in 24c.

To execute `s?printf(form,arg1,...argn)`, a call is made to `do_xprint(fp,form,cp)`, where

`fp` is a `FILE*` for output or zero for `sprintf`.

`form` is the `const Form*` returned from `parse_form()`.

`cp` is a pointer to `arg1..argn` in an array of `CELL`.

If doing `printf`, the return is zero. If doing `sprintf`, the return is the resulting `STRING*`.

3. Form Data. This section handles the parsing and storage of `form` in the function calls `printf(form,...)` and `sprintf(form,...)`.

⟨ `printf` definitions 1b ⟩+≡
```
typedef struct spec Spec ;
struct form {
    Form* link ;
    STRING* form ;
    unsigned num_args ;  /* number of args to satisfy form */
    Spec* specs ;  /* each  %..C  is one Spec */
    STRING* ending ;   /* after last Spec */
} ;
```
See also 1a, 11b and 21d.

This code is used in 24c.

All the `Form`s are kept on a singly linked list, `the_forms`.

⟨ `form` definitions and data 1c ⟩≡
```
static Form* the_forms ;
```
See also 5b, 5c, 6c and 9b.

This code is used in 24d.

This function finds a `form` on the list. Successful search causes move to the front.

⟨ `printf` functions 2a ⟩≡
```
static Form* find_form(const STRING* form)
{
    Form* p = the_forms ;
    Form* q = 0 ;
    while(p) {
        if (STRING_eq(form,p->form)) {
            if (q) { /* move to the front */
                q->link = p->link ;
                p->link = the_forms ;
                the_forms = p ;
            }
            return p ;
        }
        else {
            q = p ;
            p = p->link ;
        }
    }
    /* not found */
    return 0 ;
}
```

See also 2c, 3a, 6a, 11a, 12a, 13a, 15a, 17a, 18a, 19a, 20a, 21a, 21c, 22a, 23a and 24a.

This code is used in 24d.

⟨ static prototypes 2b ⟩≡
```
static Form* find_form(const STRING*) ;
```

See also 3b, 6b, 12b, 13b, 15b, 17b, 18b, 19b, 20b, 21b and 24b.

This code is used in 24d.

⟨ `printf` functions 2c ⟩+≡
```
const Form* parse_form(const STRING* form)
{
    Form* ret = find_form(form) ;
    if (ret) return ret ;
    ret = parse_form0(form) ;
    /* zero if an error */
    if (ret) {
        ret->link = the_forms ;
        the_forms = ret ;
    }
    return ret ;
}
```

See also 2a, 3a, 6a, 11a, 12a, 13a, 15a, 17a, 18a, 19a, 20a, 21a, 21c, 22a, 23a and 24a.

This code is used in 24d.

The real work of parsing a format happens here.

⟨ printf functions 3a ⟩+≡

```
static Form* parse_form0(const STRING* form)
{
    STRING* prefix ;
    Spec tmp_head ;
    Spec* tail = &tmp_head ;
    char xbuffer[1024] ;
    char* buffer = form->len <= 1024 ? xbuffer : (char*)emalloc(form->len) ;
    const char* str = form->str ; /* walks form */
    const char* end = str + form->len ;

    tmp_head.link = 0 ;
    Form* ret = (Form*)zmalloc(sizeof(Form)) ;

    while(1) {
        ⟨ get the prefix 4a ⟩
        if (str == end)
            ⟨ done, so finish up and return 5a ⟩
        else {
            str = parse_spec(str+1,end,tail) ;
            if (str == 0) {
                /* error, msg already done */
                return 0 ;
            }
            tail = tail->link ;
            tail->prefix = prefix ;
        }
    }
}
```

See also 2a, 2c, 6a, 11a, 12a, 13a, 15a, 17a, 18a, 19a, 20a, 21a, 21c, 22a, 23a and 24a.

This code is used in 24d.

⟨ static prototypes 3b ⟩+≡

```
static Form* parse_form0(const STRING*) ;
```

See also 2b, 6b, 12b, 13b, 15b, 17b, 18b, 19b, 20b, 21b and 24b.

This code is used in 24d.

Everything up to a `%` goes into the `prefix`, which is collected in `buffer`. `%%` becomes `%` in the `prefix`.

⟨ get the `prefix` 4a ⟩≡

```
{
    char* bp = buffer ;
    prefix = 0 ;
    while(str < end) {
        if (*str == '%') {
            if (str[1] == '%') {
                *bp++ = '%' ;
                str += 2 ;
                continue ;
            }
            else { /* prefix is done */
                if (bp > buffer) {
                    prefix = new_STRING2(buffer, bp - buffer) ;
                    bp = buffer ;
                }
                break ;
            }
        }
        else *bp++ = *str++ ;
    }
    if (bp > buffer) {
        /* prefix is really the ending */
        if (bp == buffer+1 && buffer[0] == '\n') {
            prefix = STRING_dup(the_nl_str) ;
        }
        else {
            prefix = new_STRING2(buffer, bp - buffer) ;
        }
    }
}
```

This code is used in 3a.

When we get here, the collected `prefix` is actually the `ending`, everything after the last control character. This code finishes the construction of the `Form`, returning a pointer to it.

⟨ done, so finish up and return 5a ⟩≡
```
{
    ret->ending = prefix ;
    ret->form = STRING_dup(form) ;
    ret->link = 0 ;
    ret->specs = tmp_head.link ;
    ret->num_args = 0 ;
    {
        const Spec* sp = ret->specs ;
        while(sp) {
            ret->num_args += 1 + sp->ast_cnt ;
            sp = sp->link ;
        }
    }
    if (buffer != xbuffer) free(buffer) ;
    return ret ;
}
```

This code is used in 3a.

Often the `ending` is only a new-line, here is a `STRING` to handle that case.

⟨ `form` definitions and data 5b ⟩+≡
```
static STRING the_nl = {1,1,"\n"} ;
static STRING* const the_nl_str = &the_nl ;
```

See also 1c, 5c, 6c and 9b.

This code is used in 24d.

Function `parse_spec()` analyzes `"%...C"`, where `C` is the terminating control character. On success, a pointer to past the end of the `Spec` is returned and the `Spec` is placed at the back of a singly-linked list at `tail`.

⟨ `form` definitions and data 5c ⟩+≡
```
struct spec {
    Spec* link ;
    unsigned type ;
    const STRING* prefix ;
    int width ;
    int prec ;
    int minus_flag ;
    unsigned ast_cnt ;
    const char* form ;
} ;
```

See also 1c, 5b, 6c and 9b.

This code is used in 24d.

⟨ printf functions 6a ⟩ +≡

```
const char* parse_spec(const char* str, const char* end, Spec* tail)
{
    const char* const start = str-1 ;
    Spec* spec = (Spec*)zmalloc(sizeof(Spec)) ;
    spec->link = 0 ;
    spec->width = WP_NONE ;
    spec->prec  = WP_NONE ;
    spec->minus_flag = 0 ;
    spec->ast_cnt = 0 ;
    spec->form = 0 ;
    unsigned l_cnt = 0 ;

    ⟨ do flags 7a ⟩
    ⟨ do width and precision 8a ⟩
    ⟨ count 'l' 9a ⟩
    ⟨ finish with the control character 9c ⟩
    tail->link = spec ;
    return str+1 ;
}
```

See also 2a, 2c, 3a, 11a, 12a, 13a, 15a, 17a, 18a, 19a, 20a, 21a, 21c, 22a, 23a and 24a.

This code is used in 24d.

⟨ static prototypes 6b ⟩ +≡

```
static const char* parse_spec(const char*, const char*, Spec*) ;
```

See also 2b, 3b, 12b, 13b, 15b, 17b, 18b, 19b, 20b, 21b and 24b.

This code is used in 24d.

The `width` and `prec` can be an `*`, `WP_AST`, or nothing, `WP_NONE`, or a non-negative integer.

⟨ `form` definitions and data 6c ⟩ +≡

```
enum { WP_AST = -1, WP_NONE = -2 } ;
```

See also 1c, 5b, 5c and 9b.

This code is used in 24d.

⟨ do flags 7a ⟩≡

```
{
    while(str < end) {
        switch(*str) {
            case '-' :
                spec->minus_flag = 1 ;
                /* fall thru */
            case '+' : case ' ' : case '#' : case '0' :
                str++ ;
                continue ;
            default:
                break ;
        }
        break ; /* while */
    }
    if (str == end) {
        c_or_rt_error_or_silence(
            "incomplete format specification \"%s\"", start) ;
        ⟨ return error 7b ⟩
    }
}
```

This code is used in 6a.

A null return indicates a misformed spec. We do not attempt to recover allocated memory as the program will exit soon.

⟨ return error 7b ⟩≡

```
{
    return 0 ;
}
```

This code is used in 7a and 9c.

⟨ do width and precision 8a ⟩≡

```
{
    if (*str == '*') {
        spec->width = WP_AST ;
        spec->ast_cnt = 1 ;
        str++ ;
    }
    else {
        unsigned char c = *(unsigned char*) str ;

        if (scan_code[c] == SC_DIGIT) {
            spec->width = c - '0' ;
            str++ ;
            while(1) {
                c = *(unsigned char*) str ;
                if (scan_code[c] != SC_DIGIT) break ;
                spec->width = spec->width * 10 + c - '0' ;
                str++ ;
            }
        }
    }
    /* width is done */
    if (*str == '.') {
        /* have precision , no numbers will be prec 0 */
        str++ ;
        if (*str == '*') {
            spec->prec = WP_AST ;
            spec->ast_cnt++ ;
            str++ ;
        }
        else {
            unsigned char c ;
            spec->prec = 0 ;
            while(1) {
                c = * (unsigned char*) str ;
                if (scan_code[c] != SC_DIGIT) break ;
                spec->prec = spec->prec * 10 + c - '0' ;
                str++ ;
            }
        }
    }
}
```

This code is used in 6a.

An 'l' modifier is not necessary in awk, but some awk's accept it and earlier mawk did, so we accept it as long as followed by an integer control character, d, i, etc.

⟨ count 'l' 9a ⟩≡
```
{
    while(*str == 'l') {
        str++ ;
        l_cnt++ ;
    }
}
```
This code is used in 6a.

The next expected character is the control character and it determines the type of the Spec. The types are,

PF_C print a character, %c.

PF_I print a signed integer, %d or %i.

PF_U print an unsigned integer, %u, %x, %X or %o.

PF_F print a floating point number, %f, %g, %G, %e or %E.

PF_S print a string, %s.

⟨ form definitions and data 9b ⟩+≡
```
enum { PF_C = 0, PF_I, PF_U, PF_F, PF_S } ;
```
See also 1c, 5b, 5c and 6c.

This code is used in 24d.

⟨ finish with the control character 9c ⟩≡
```
{
    switch(*str) {
        case 's':
            spec->type = PF_S ;
            break ;
        case 'c':
        case 'e': case 'g': case 'f': case 'E': case 'G':
            ⟨ make the spec type PF_F or type PF_C 10a ⟩
            break ;
        case 'd': case 'i':
        case 'o': case 'x': case 'X': case 'u':
            ⟨ make the spec type PF_I or type PF_U 10b ⟩
            break ;
        default:
           c_or_rt_error_or_silence(
  "invalid control character '%c' in [s]printf format (\"%s\")",
           *(unsigned char*) str, start) ;
           ⟨ return error 7b ⟩
           break ;
    }
}
```
This code is used in 6a.

An 'l' modifier does not make sense here, but rather than make it an error, we silently remove it. This is a minor change in mawk behavior.

⟨ make the spec type PF_F or type PF_C 10a ⟩ ≡

```
{
    size_t len = str+1 - start - l_cnt ;
    char* form = (char*) zmalloc(len+1) ;
    spec->type = *str == 'c' ? PF_C : PF_F ;
    memcpy(form, start, len-1) ;
    form[len] = 0 ;
    form[len-1] = *str ;
    spec->form = form ;
}
```

This code is used in 9c.

A printf form such as "%2x" in an awk program needs to be converted to "%2lx" for a call to the 64 bit long C-library or "%2llx" for the 32 bit long C-library. The user is allowed to use 'l' in an awk program, but it has no real effect.

⟨ make the spec type PF_I or type PF_U 10b ⟩ ≡

```
{
    size_t len = str+1 - start ;
    int delta = (have_long64? 1 : 2) - l_cnt ;
        /* delta is number of 'l' to add or remove */
    char* form ;

    len += delta ;
    form = (char*) zmalloc(len+1) ;
    if (*str == 'd' || *str == 'i') {
        spec->type = PF_I ;
    }
    else {
        spec->type = PF_U ;
    }
    form[len] = 0 ;
    form[len-1] = *str ;
    form[len-2] = 'l' ;
    if (!have_long64) {
        form[len-3] = 'l' ;
    }
    memcpy(form, start, len - (have_long64?2:3)) ;
    spec->form = form ;
}
```

This code is used in 9c.

4. Output. Up to the final output, to a file or copy to a string, the execution of `printf()` and `sprintf()` is the same. Both start at `do_xprintf(fp,form,cp)` with arguments,

`fp` is `FILE*` to the output file, if doing `printf()`. If doing `sprintf()` is zero.

`form` is the `Form*` parsed from the format string.

`cp` points into an array of arguments. The number of arguments is `form->num_args`. The `CELL` arguments can be modified, but the caller owns the cells.

If doing `sprintf()`, the return is the `STRING*` output, else zero.

⟨ printf functions 11a ⟩ +≡

```
STRING*  do_xprintf(
    FILE* fp,
    const Form* form,
    CELL* cp)
{
    const Spec* spec = form->specs ;

    sprintf_ptr = string_buff ;
    while(spec) {
        if (spec->prefix) {
            xprint_string(spec->prefix->str,spec->prefix->len,fp) ;
        }
        if (spec->type == PF_S) {
            xprint_string_spec(fp,spec,cp) ;
        }
        else {
            xprint_num_spec(fp,spec,cp) ;
        }
        cp +=  1 + spec->ast_cnt ;
        spec = spec->link ;
    }
    if (form->ending) {
        xprint_string(form->ending->str, form->ending->len, fp) ;
    }
    if (fp) {
        if (ferror(fp)) {
            write_error(fp) ;
            mawk_exit(2) ;
        }
        return 0 ;
    }
    return new_STRING2(string_buff, sprintf_ptr - string_buff) ;
}
```

See also 2a, 2c, 3a, 6a, 12a, 13a, 15a, 17a, 18a, 19a, 20a, 21a, 21c, 22a, 23a and 24a.

This code is used in 24d.

⟨ printf definitions 11b ⟩ +≡

```
STRING* do_xprintf(FILE*, const Form*, CELL*) ;
```

See also 1a, 1b and 21d.

This code is used in 24c.

Here are two helper functions. The first outputs an array of characters and the second outputs an array of spaces.

⟨ printf functions 12a ⟩ +≡

```
static void xprint_string(const char* str, size_t len,  FILE* fp)
{
    if (fp) {
        fwrite(str, 1, len, fp) ;
    }
    else {
        size_t room = string_buff_end - sprintf_ptr ;
        while(room < len) {
            sprintf_ptr = enlarge_string_buff(sprintf_ptr) ;
            room = string_buff_end - sprintf_ptr ;
        }
        memcpy(sprintf_ptr, str, len) ;
        sprintf_ptr += len ;
    }
}

static void xprint_spaces(size_t len, FILE* fp)
{
    char buffer[1024] ;
    memset(buffer, ' ', len > 1024 ? 1024 : len) ;
    while(len > 1024) {
        xprint_string(buffer, 1024, fp) ;
        len -= 1024 ;
    }
    xprint_string(buffer, len, fp) ;
}
```

See also 2a, 2c, 3a, 6a, 11a, 13a, 15a, 17a, 18a, 19a, 20a, 21a, 21c, 22a, 23a and 24a.

This code is used in 24d.

⟨ static prototypes 12b ⟩ +≡

```
static void xprint_string(const char*,size_t,FILE*) ;
static void xprint_spaces(size_t,FILE*) ;
```

See also 2b, 3b, 6b, 13b, 15b, 17b, 18b, 19b, 20b, 21b and 24b.

This code is used in 24d.

When outputting for `sprintf()` to `string_buff`, the place to write is `sprintf_ptr` ;

⟨ static data 12c ⟩ ≡

```
char* sprintf_ptr ;
```

See also 23b.

This code is used in 24d.

4.1. Output for a %..s Specification. This function outputs the STRING at cp using a PF_S spec. Note that since the STRING might have \0, we do the implementation instead of passing it to a C-library function.

⟨ printf functions 13a ⟩ +≡

```
static
void xprint_string_spec(FILE* fp, const Spec* spec, CELL* cp)
{
    int width = spec->width ;
    int prec = spec->prec ;
    const char* str ;
    size_t len ;
    int minus_flag = spec->minus_flag ;

    ⟨ get width and prec for * 14a ⟩
    cast1_to_s(cp) ;
    str = string(cp)->str ;
    len = string(cp)->len ;

    if (width < 0) width = 0 ; /* effectively ignore */
    if (prec < 0)  prec = len ; /* ditto */
    if (prec < len) len = prec ;

    /* output str and spaces if needed */
    if (len >= width) {
       xprint_string(str,len,fp) ;
    }
    else if (minus_flag) {
        /* left justify */
        xprint_string(str,len,fp) ;
        xprint_spaces(width-len,fp) ;
    }
    else {
        xprint_spaces(width-len,fp) ;
        xprint_string(str,len,fp) ;
    }
}
```

See also 2a, 2c, 3a, 6a, 11a, 12a, 15a, 17a, 18a, 19a, 20a, 21a, 21c, 22a, 23a and 24a.

This code is used in 24d.

⟨ static prototypes 13b ⟩ +≡

```
static void xprint_string_spec(FILE*,const Spec*,CELL*) ;
```

See also 2b, 3b, 6b, 12b, 15b, 17b, 18b, 19b, 20b, 21b and 24b.

This code is used in 24d.

Here is a tricky point: for `%*s`, if the value passed for `*` is negative, e.g., `-5`, then it is the same as passing `5` to `%-*s`.

⟨ get `width` and `prec` for `*` 14a ⟩ ≡

```
{
    if (width == WP_AST) {
        cast1_to_d(cp) ;
        width = d_to_int(cp->dval) ;
        if (width < 0) {
            minus_flag = 1 ;
            width = -width ;
        }
        cp++ ;
    }
    if (prec == WP_AST) {
        cast1_to_d(cp) ;
        prec = d_to_int(cp->dval) ;
        cp++ ;
    }
}
```

This code is used in 13a.

4.2. Output for Numeric Specifications. There are four numeric specifications. The function, `xprint_num_spec(fp,spec,cp)` breaks the output into the four cases, converting the data in `*cp` to the appropriate type.

⟨ `printf` functions 15a ⟩ +≡

```
static
void xprint_num_spec(FILE* fp, const Spec* spec, CELL* cp)
{
    unsigned ast_cnt = spec->ast_cnt ;
    int ast[2] ;

    ⟨ set ast[2], incrementing cp as needed 16a ⟩

    switch(spec->type) {
        case PF_F:
            if (cp->type != C_DOUBLE) cast1_to_d(cp) ;
            xprint_pf_f(fp, spec->form, ast_cnt, ast[0],ast[1],cp->dval) ;
            break ;

        case PF_I:
            if (cp->type != C_DOUBLE) cast1_to_d(cp) ;
            xprint_pf_i(fp, spec->form, ast_cnt, ast[0],ast[1],
                        d_to_i64(cp->dval)) ;
            break ;

        case PF_U:
            if (cp->type != C_DOUBLE) cast1_to_d(cp) ;
            xprint_pf_u(fp, spec->form, ast_cnt, ast[0],ast[1],
                        d_to_u64(cp->dval)) ;
            break ;

        case PF_C:
            {
                unsigned c ;
                ⟨ get the value of *cp into c 16b ⟩
                xprint_pf_c(fp, spec->form, ast_cnt, ast[0],ast[1],c) ;
            }
            break ;
    }
}
```

See also 2a, 2c, 3a, 6a, 11a, 12a, 13a, 17a, 18a, 19a, 20a, 21a, 21c, 22a, 23a and 24a.

This code is used in 24d.

⟨ static prototypes 15b ⟩ +≡

```
static void xprint_num_spec(FILE*, const Spec*, CELL*) ;
```

See also 2b, 3b, 6b, 12b, 13b, 17b, 18b, 19b, 20b, 21b and 24b.

This code is used in 24d.

Note the value assigned to * is supposed to be non-negative, but we do not check as the C-library function will handle it.

⟨ set ast[2], incrementing cp as needed 16a ⟩ ≡

```
{
    if (ast_cnt >= 1)  {
        cast1_to_d(cp) ;
        ast[0] = d_to_int(cp->dval) ;
        cp++ ;
    }
    if (ast_cnt == 2) {
        cast1_to_d(cp) ;
        ast[1] = d_to_int(cp->dval) ;
        cp++ ;
    }
}
```

This code is used in 15a.

%c is a little weird in awk, for numeric values it uses the value, but for string values, it uses the first character in the string or zero for the empty string.

⟨ get the value of *cp into c 16b ⟩ ≡

```
{
    int64_t val ;

    if (cp->type == C_MBSTRN) {
        check_strnum(cp) ;
    }
    if (cp->type == C_STRING) {
        c = string(cp)->str[0] ;
    }
    else {
        cast1_to_d(cp) ;
        val = d_to_i64(cp->dval) ;
        c = (unsigned) (val & 0xff) ;
    }
}
```

This code is used in 15a.

If working in C++, to get the four functions, xprint_fp_{f,i,u,c}(), we would write one template function and instantiate it four times. Since we are working in ansi C, the following sub-sections are instantiation by cut and paste.

4.2.1. Floating Point Format.

⟨ printf functions 17a ⟩ +≡

```
static void xprint_pf_f(FILE* fp, const char* form,
    unsigned cnt, int x, int y, double d)
{
    if (fp) {
        switch(cnt) {
            case 0:
                fprintf(fp, form, d) ;
                break ;
            case 1:
                fprintf(fp, form, x, d) ;
                break ;
            case 2:
                fprintf(fp, form, x, y, d) ;
                break ;
        }
    }
    else {
        size_t room = string_buff_end - sprintf_ptr ;
        size_t used = (size_t)-1 ;

        while(1) {
            switch(cnt) {
                case 0:
                    used = snprintf(sprintf_ptr, room, form, d) ;
                    break ;
                case 1:
                    used = snprintf(sprintf_ptr, room, form, x ,d) ;
                    break ;
                case 2:
                    used = snprintf(sprintf_ptr, room, form, x,y,d) ;
                    break ;
            }
            if ((int) used < 0) snprintf_bozo() ;

            if (used <= room) break ; /* while */
            do {
                sprintf_ptr = enlarge_string_buff(sprintf_ptr) ;
                room = string_buff_end - sprintf_ptr ;
            }
            while(room < used) ;
        }
        sprintf_ptr += used ;
    }
}
```

See also 2a, 2c, 3a, 6a, 11a, 12a, 13a, 15a, 18a, 19a, 20a, 21a, 21c, 22a, 23a and 24a.

This code is used in 24d.

⟨ static prototypes 17b ⟩ +≡

```
static void xprint_pf_f(FILE*,const char*,unsigned,int,int,double) ;
```

See also 2b, 3b, 6b, 12b, 13b, 15b, 18b, 19b, 20b, 21b and 24b.

This code is used in 24d.

4.2.2. Integer Format.

⟨ printf functions 18a ⟩ +≡

```
static void xprint_pf_i(FILE* fp, const char* form,
    unsigned cnt, int x, int y, int64_t val)
{
    if (fp) {
        switch(cnt) {
            case 0:
                fprintf(fp, form, val) ;
                break ;
            case 1:
                fprintf(fp, form, x, val) ;
                break ;
            case 2:
                fprintf(fp, form, x, y, val) ;
                break ;
        }
    }
    else {
        size_t room = string_buff_end - sprintf_ptr ;
        size_t used = (size_t)-1 ;

        while(1) {
            switch(cnt) {
                case 0:
                    used = snprintf(sprintf_ptr, room, form, val) ;
                    break ;
                case 1:
                    used = snprintf(sprintf_ptr, room, form, x ,val) ;
                    break ;
                case 2:
                    used = snprintf(sprintf_ptr, room, form, x,y,val) ;
                    break ;
            }
            if ((int) used < 0) snprintf_bozo() ;

            if (used <= room) break ; /* while */
            do {
                sprintf_ptr = enlarge_string_buff(sprintf_ptr) ;
                room = string_buff_end - sprintf_ptr ;
            }
            while(room < used) ;
        }
        sprintf_ptr += used ;
    }
}
```

See also 2a, 2c, 3a, 6a, 11a, 12a, 13a, 15a, 17a, 19a, 20a, 21a, 21c, 22a, 23a and 24a.

This code is used in 24d.

⟨ static prototypes 18b ⟩ +≡

```
static void xprint_pf_i(FILE*,const char*,unsigned,int,int, int64_t) ;
```

See also 2b, 3b, 6b, 12b, 13b, 15b, 17b, 19b, 20b, 21b and 24b.

This code is used in 24d.

4.2.3. Unsigned Integer Format.

⟨ printf functions 19a ⟩ +≡

```
static void xprint_pf_u(FILE* fp, const char* form,
    unsigned cnt, int x, int y, uint64_t val)
{
    if (fp) {
        switch(cnt) {
            case 0:
                fprintf(fp, form, val) ;
                break ;
            case 1:
                fprintf(fp, form, x, val) ;
                break ;
            case 2:
                fprintf(fp, form, x, y, val) ;
                break ;
        }
    }
    else {
        size_t room = string_buff_end - sprintf_ptr ;
        size_t used = (size_t)-1 ;

        while(1) {
            switch(cnt) {
                case 0:
                    used = snprintf(sprintf_ptr, room, form, val) ;
                    break ;
                case 1:
                    used = snprintf(sprintf_ptr, room, form, x ,val) ;
                    break ;
                case 2:
                    used = snprintf(sprintf_ptr, room, form, x,y,val) ;
                    break ;
            }
            if ((int) used < 0) snprintf_bozo() ;

            if (used <= room) break ; /* while */
            do {
                sprintf_ptr = enlarge_string_buff(sprintf_ptr) ;
                room = string_buff_end - sprintf_ptr ;
            }
            while(room < used) ;
        }
        sprintf_ptr += used ;
    }
}
```

See also 2a, 2c, 3a, 6a, 11a, 12a, 13a, 15a, 17a, 18a, 20a, 21a, 21c, 22a, 23a and 24a.

This code is used in 24d.

⟨ static prototypes 19b ⟩ +≡

```
static void xprint_pf_u(FILE*,const char*,unsigned,int,int, uint64_t) ;
```

See also 2b, 3b, 6b, 12b, 13b, 15b, 17b, 18b, 20b, 21b and 24b.

This code is used in 24d.

4.2.4. Single Character Format, %c. Here the input, `val`, is known to be in the range, `[0,255)`.

⟨ `printf` functions 20a ⟩ +≡

```
static void xprint_pf_c(FILE* fp, const char* form,
    unsigned cnt, int x, int y, unsigned val)
{
    if (fp) {
        switch(cnt) {
            case 0:
                fprintf(fp, form, val) ;
                break ;
            case 1:
                fprintf(fp, form, x, val) ;
                break ;
            case 2:
                fprintf(fp, form, x, y, val) ;
                break ;
        }
    }
    else {
        size_t room = string_buff_end - sprintf_ptr ;
        size_t used = (size_t)-1 ;

        while(1) {
            switch(cnt) {
                case 0:
                    used = snprintf(sprintf_ptr, room, form, val) ;
                    break ;
                case 1:
                    used = snprintf(sprintf_ptr, room, form, x ,val) ;
                    break ;
                case 2:
                    used = snprintf(sprintf_ptr, room, form, x,y,val) ;
                    break ;
            }
            if ((int) used < 0) snprintf_bozo() ;

            if (used <= room) break ; /* while */
            do {
                sprintf_ptr = enlarge_string_buff(sprintf_ptr) ;
                room = string_buff_end - sprintf_ptr ;
            }
            while(room < used) ;
        }
        sprintf_ptr += used ;
    }
}
```

See also 2a, 2c, 3a, 6a, 11a, 12a, 13a, 15a, 17a, 18a, 19a, 21a, 21c, 22a, 23a and 24a.

This code is used in 24d.

⟨ static prototypes 20b ⟩ +≡

```
static void xprint_pf_c(FILE*,const char*,unsigned,int,int, unsigned) ;
```

See also 2b, 3b, 6b, 12b, 13b, 15b, 17b, 18b, 19b, 21b and 24b.

This code is used in 24d.

⟨ printf functions 21a ⟩ +≡

```
static void snprintf_bozo() {
    rt_error("unexpected error (%d) in internal call to snprintf", errno) ;
    /* no return */
}
```

See also 2a, 2c, 3a, 6a, 11a, 12a, 13a, 15a, 17a, 18a, 19a, 20a, 21c, 22a, 23a and 24a.

This code is used in 24d.

⟨ static prototypes 21b ⟩ +≡

```
static void snprintf_bozo(void) ;
```

See also 2b, 3b, 6b, 12b, 13b, 15b, 17b, 18b, 19b, 20b and 24b.

This code is used in 24d.

5. Loose Ends.

5.1. `Form` Disassembly.

Here is support for `-Wd`.

⟨ printf functions 21c ⟩ +≡

```
void visible_string(FILE*, const STRING*, int) ; /* in da.c */
static void da_Spec(FILE*, const Spec*) ;
void da_Form(FILE* fp, const Form* form)
{
    const Spec* spec = form->specs ;
    while(spec) {
        da_Spec(fp,spec) ;
        spec = spec->link ;
    }
    if (form->ending) {
        visible_string(fp, form->ending,'\"') ;
    }
}
```

See also 2a, 2c, 3a, 6a, 11a, 12a, 13a, 15a, 17a, 18a, 19a, 20a, 21a, 22a, 23a and 24a.

This code is used in 24d.

⟨ printf definitions 21d ⟩ +≡

```
void da_Form(FILE*,const Form*) ;
```

See also 1a, 1b and 11b.

This code is used in 24c.

⟨ printf functions 22a ⟩ +≡

```
static void da_Spec(FILE* fp, const Spec* spec)
{
    if (spec->prefix) visible_string(fp, spec->prefix, '\"') ;
    if (spec->type != PF_S) {
        fprintf(fp, "%s", spec->form) ;
    }
    else {
        fputc('%', fp) ;
        if (spec->minus_flag) fputc('-',fp) ;
        if (spec->width == WP_AST) {
            fputc('*',fp) ;
        }
        else if (spec->width >= 0) {
            fprintf(fp, "%d", spec->width) ;
        }
        if (spec->prec >= 0) {
            fprintf(fp, ".%d", spec->prec) ;
        }
        else if (spec->prec == WP_AST) {
            fputs(".*", fp) ;
        }
        fputc('s', fp) ;
    }
}
```

See also 2a, 2c, 3a, 6a, 11a, 12a, 13a, 15a, 17a, 18a, 19a, 20a, 21a, 21c, 23a and 24a.
This code is used in 24d.

5.2. Checking `CONVFMT`. A bug report, against `mawk`, reported a core-dump on bad assignment to `CONVFMT`. The user is either careless or malicious, but we will protect him/her by no longer allowing weird assignment to `CONVFMT` or `OFMT`. This is a minor change in `mawk`'s behavior.

If `conv` is a valid string for assignment to `CONVFMT`, then `conv` is returned else zero. Valid means it parses to a `Form*` that takes one argument of type `PF_F`. Any prefix or ending does not contain `'\0'`.

⟨ `printf` functions 23a ⟩ +≡

```
STRING* check_convfmt(const STRING* conv)
{
    const Form* form ;
    checking_convfmt = 1 ;
    form = parse_form(conv) ;
    checking_convfmt = 0 ;
    if (form == 0 || form->num_args != 1 || form->specs->type != PF_F) {
        return 0 ;
    }
    /* no \0 in prefix or ending */
    if (form->ending && memchr(form->ending->str, 0, form->ending->len)) {
        return 0 ;
    }
    {
        const STRING* prefix = form->specs->prefix ;
        if (prefix && memchr(prefix->str, 0, prefix->len)) {
            return 0 ;
        }
    }
    return  new_STRING(form->specs->form) ;
}
```

See also 2a, 2c, 3a, 6a, 11a, 12a, 13a, 15a, 17a, 18a, 19a, 20a, 21a, 21c, 22a and 24a.

This code is used in 24d.

Setting this to `1`, turns off error messages in `parse_form()`.

⟨ static data 23b ⟩ +≡

```
static int checking_convfmt = 0 ;
```

See also 12c.

This code is used in 24d.

⟨ printf functions 24a ⟩ +≡

```
static void c_or_rt_error_or_silence(const char* format, ...)
{
    if (! checking_convfmt) {
        /* up to caller not to exceed this buffer */
        char buffer[1024] ;
        va_list args ;

        va_start(args,format) ;
        vsprintf(buffer, format, args) ;
        if (mawk_state == EXECUTION) {
            rt_error(buffer) ;
        }
        else {
            compile_error(buffer) ;
        }
    }
}
```

See also 2a, 2c, 3a, 6a, 11a, 12a, 13a, 15a, 17a, 18a, 19a, 20a, 21a, 21c, 22a and 23a.

This code is used in 24d.

⟨ static prototypes 24b ⟩ +≡

```
static void c_or_rt_error_or_silence(const char*,...) ;
```

See also 2b, 3b, 6b, 12b, 13b, 15b, 17b, 18b, 19b, 20b and 21b.

This code is used in 24d.

6. Source Files.

⟨ "printf.h" 24c ⟩ ≡

```
/* printf.h */
⟨ blurb 25a ⟩
#ifndef PRINTF_H
#define PRINTF_H 1
#include "mawk.h"
#include "types.h"
#include "memory.h"
#include "files.h"

⟨ printf definitions 1a, ... ⟩
#endif /* PRINTF_H */
```

⟨ "printf.c" 24d ⟩ ≡

```
/* printf.c */
⟨ blurb 25a ⟩

#include <stdarg.h>
#include "mawk.h"
#include "scan.h"
#include "printf.h"
#include "int.h"

⟨ form definitions and data 1c, ... ⟩
⟨ static prototypes 2b, ... ⟩
⟨ static data 12c, ... ⟩
⟨ printf functions 2a, ... ⟩
```

⟨ blurb 25a ⟩≡

```
/*
copyright 2016 Michael D. Brennan

This is a source file for mawk, an implementation of
the AWK programming language.

Mawk is distributed without warranty under the terms of
the GNU General Public License, version 3, 2007.

printf.c and printf.h were generated with the commands

   notangle -R'"printf.c"' printf.w > printf.c
   notangle -R'"printf.h"' printf.w > printf.h

Notangle is part of Norman Ramsey's noweb literate programming package.
Noweb home page: http://www.cs.tufts.edu/~nr/noweb/

It's easiest to read or modify this file by working with printf.w.
*/
```

This code is used in 24c and 24d.

7. Identifier Index. Underlined code chunks are identifier definitions; other chunks are identifier uses.