```
========================================
How to get printk format specifiers right
========================================

:Author: Randy Dunlap <rdunlap@infradead.org>
:Author: Andrew Murray <amurray@mpc-data.co.uk>

Integer types
=============

::

        If variable is of Type,         use printk format specifier:
        ------------------------------------------------------------
                int                     %d or %x
                unsigned int            %u or %x
                long                    %ld or %lx
                unsigned long           %lu or %lx
                long long               %lld or %llx
                unsigned long long      %llu or %llx
                size_t                  %zu or %zx
                ssize_t                 %zd or %zx
                s32                     %d or %x
                u32                     %u or %x
                s64                     %lld or %llx
                u64                     %llu or %llx

If <type> is dependent on a config option for its size (e.g., ``sector_t``,
``blkcnt_t``) or is architecture-dependent for its size (e.g., ``tcflag_t``),
use a format specifier of its largest possible type and explicitly cast to it.

Example::

        printk("test: sector number/total blocks: %llu/%llu\n",
                (unsigned long long)sector, (unsigned long long)blockcount);

Reminder: ``sizeof()`` result is of type ``size_t``.

The kernel's printf does not support ``%n``. For obvious reasons, floating
point formats (``%e, %f, %g, %a``) are also not recognized. Use of any
unsupported specifier or length qualifier results in a WARN and early
return from vsnprintf.

Raw pointer value SHOULD be printed with %p. The kernel supports
the following extended format specifiers for pointer types:

Pointer Types
=============

Pointers printed without a specifier extension (i.e unadorned %p) are
hashed to give a unique identifier without leaking kernel addresses to user
space. On 64 bit machines the first 32 bits are zeroed. If you _really_
want the address see %px below.

::

        %p      abcdef12 or 00000000abcdef12

Symbols/Function Pointers
=========================

::

        %pF     versatile_init+0x0/0x110
```

```
        %pf     versatile_init
        %pS     versatile_init+0x0/0x110
        %pSR    versatile_init+0x9/0x110
                (with __builtin_extract_return_addr() translation)
        %ps     versatile_init
        %pB     prev_fn_of_versatile_init+0x88/0x88
```

The ``F`` and ``f`` specifiers are for printing function pointers,
for example, f->func, &gettimeofday. They have the same result as
``S`` and ``s`` specifiers. But they do an extra conversion on
ia64, ppc64 and parisc64 architectures where the function pointers
are actually function descriptors.

The ``S`` and ``s`` specifiers can be used for printing symbols
from direct addresses, for example, __builtin_return_address(0),
(void *)regs->ip. They result in the symbol name with (``S``) or
without (``s``) offsets. If KALLSYMS are disabled then the symbol
address is printed instead.

The ``B`` specifier results in the symbol name with offsets and should be
used when printing stack backtraces. The specifier takes into
consideration the effect of compiler optimisations which may occur
when tail-call``s are used and marked with the noreturn GCC attribute.

Examples::

```
        printk("Going to call: %pF\n", gettimeofday);
        printk("Going to call: %pF\n", p->func);
        printk("%s: called from %pS\n", __func__, (void *)_RET_IP_);
        printk("%s: called from %pS\n", __func__,
                                (void *)__builtin_return_address(0));
        printk("Faulted at %pS\n", (void *)regs->ip);
        printk(" %s%pB\n", (reliable ? "" : "? "), (void *)*stack);
```

Kernel Pointers
===============

::

```
        %pK     01234567 or 0123456789abcdef
```

For printing kernel pointers which should be hidden from unprivileged
users. The behaviour of ``%pK`` depends on the ``kptr_restrict sysctl`` - see
Documentation/sysctl/kernel.txt for more details.

Unmodified Addresses
====================

::

```
        %px     01234567 or 0123456789abcdef
```

For printing pointers when you _really_ want to print the address. Please
consider whether or not you are leaking sensitive information about the
Kernel layout in memory before printing pointers with %px. %px is
functionally equivalent to %lx. %px is preferred to %lx because it is more
uniquely grep'able. If, in the future, we need to modify the way the Kernel
handles printing pointers it will be nice to be able to find the call
sites.

Struct Resources
================

::

```
        %pr     [mem 0x60000000-0x6fffffff flags 0x2200] or
                [mem 0x0000000060000000-0x000000006fffffff flags 0x2200]
        %pR     [mem 0x60000000-0x6fffffff pref] or
                [mem 0x0000000060000000-0x000000006fffffff pref]
```

For printing struct resources. The ``R`` and ``r`` specifiers result in a printed resource with (``R``) or without (``r``) a decoded flags member. Passed by reference.

Physical addresses types ``phys_addr_t``
==

::

```
        %pa[p]  0x01234567 or 0x0123456789abcdef
```

For printing a ``phys_addr_t`` type (and its derivatives, such as ``resource_size_t``) which can vary based on build options, regardless of the width of the CPU data path. Passed by reference.

DMA addresses types ``dma_addr_t``
==================================

::

```
        %pad    0x01234567 or 0x0123456789abcdef
```

For printing a ``dma_addr_t`` type which can vary based on build options, regardless of the width of the CPU data path. Passed by reference.

Raw buffer as an escaped string
===============================

::

```
        %*pE[achnops]
```

For printing raw buffer as an escaped string. For the following buffer::

```
                1b 62 20 5c 43 07 22 90 0d 5d
```

few examples show how the conversion would be done (the result string without surrounding quotes)::

```
                %*pE            "\eb \C\a"\220\r]"
                %*pEhp          "\x1bb \C\x07"\x90\x0d]"
                %*pEa           "\e\142\040\\\103\a\042\220\r\135"
```

The conversion rules are applied according to an optional combination of flags (see :c:func:`string_escape_mem` kernel documentation for the details):

- ``a`` - ESCAPE_ANY
- ``c`` - ESCAPE_SPECIAL
- ``h`` - ESCAPE_HEX
- ``n`` - ESCAPE_NULL
- ``o`` - ESCAPE_OCTAL
- ``p`` - ESCAPE_NP
- ``s`` - ESCAPE_SPACE

By default ESCAPE_ANY_NP is used.

ESCAPE_ANY_NP is the sane choice for many cases, in particularly for printing SSIDs.

If field width is omitted the 1 byte only will be escaped.

Raw buffer as a hex string
==========================

::

        %*ph   00 01 02 ... 3f
        %*phC   00:01:02: ... :3f
        %*phD   00-01-02- ... -3f
        %*phN   000102 ... 3f

For printing a small buffers (up to 64 bytes long) as a hex string with
certain separator. For the larger buffers consider to use
:c:func:`print_hex_dump`.

MAC/FDDI addresses
==================

::

        %pM   00:01:02:03:04:05
        %pMR   05:04:03:02:01:00
        %pMF   00-01-02-03-04-05
        %pm   000102030405
        %pmR   050403020100

For printing 6-byte MAC/FDDI addresses in hex notation. The ``M`` and ``m``
specifiers result in a printed address with (``M``) or without (``m``) byte
separators. The default byte separator is the colon (``:``).

Where FDDI addresses are concerned the ``F`` specifier can be used after
the ``M`` specifier to use dash (``-``) separators instead of the default
separator.

For Bluetooth addresses the ``R`` specifier shall be used after the ``M``
specifier to use reversed byte order suitable for visual interpretation
of Bluetooth addresses which are in the little endian order.

Passed by reference.

IPv4 addresses
==============

::

        %pI4   1.2.3.4
        %pi4   001.002.003.004
        %p[Ii]4[hnbl]

For printing IPv4 dot-separated decimal addresses. The ``I4`` and ``i4``
specifiers result in a printed address with (``i4``) or without (``I4``)
leading zeros.

The additional ``h``, ``n``, ``b``, and ``l`` specifiers are used to specify
host, network, big or little endian order addresses respectively. Where
no specifier is provided the default network/big endian order is used.

Passed by reference.

IPv6 addresses
==============

::

```
        %pI6    0001:0002:0003:0004:0005:0006:0007:0008
        %pi6    00010002000300040005000600070008
        %pI6c   1:2:3:4:5:6:7:8

 For printing IPv6 network-order 16-bit hex addresses. The ``I6`` and ``i6``
 specifiers result in a printed address with (``I6``) or without (``i6``)
 colon-separators. Leading zeros are always used.

 The additional ``c`` specifier can be used with the ``I`` specifier to
 print a compressed IPv6 address as described by
 http://tools.ietf.org/html/rfc5952

 Passed by reference.

 IPv4/IPv6 addresses (generic, with port, flowinfo, scope)
 =========================================================

 ::

        %pIS    1.2.3.4         or 0001:0002:0003:0004:0005:0006:0007:0008
        %piS    001.002.003.004 or 00010002000300040005000600070008
        %pISc   1.2.3.4         or 1:2:3:4:5:6:7:8
        %pISpc  1.2.3.4:12345   or [1:2:3:4:5:6:7:8]:12345
        %p[Ii]S[pfschnbl]

 For printing an IP address without the need to distinguish whether it``s
 of type AF_INET or AF_INET6, a pointer to a valid ``struct sockaddr``,
 specified through ``IS`` or ``iS``, can be passed to this format specifier.

 The additional ``p``, ``f``, and ``s`` specifiers are used to specify port
 (IPv4, IPv6), flowinfo (IPv6) and scope (IPv6). Ports have a ``:`` prefix,
 flowinfo a ``/`` and scope a ``%``, each followed by the actual value.

 In case of an IPv6 address the compressed IPv6 address as described by
 http://tools.ietf.org/html/rfc5952 is being used if the additional
 specifier ``c`` is given. The IPv6 address is surrounded by ``[``, ``]`` in
 case of additional specifiers ``p``, ``f`` or ``s`` as suggested by
 https://tools.ietf.org/html/draft-ietf-6man-text-addr-representation-07

 In case of IPv4 addresses, the additional ``h``, ``n``, ``b``, and ``l``
 specifiers can be used as well and are ignored in case of an IPv6
 address.

 Passed by reference.

 Further examples::

        %pISfc          1.2.3.4         or [1:2:3:4:5:6:7:8]/123456789
        %pISsc          1.2.3.4         or [1:2:3:4:5:6:7:8]%1234567890
        %pISpfc         1.2.3.4:12345   or [1:2:3:4:5:6:7:8]:12345/123456789

 UUID/GUID addresses
 ===================

 ::

        %pUb    00010203-0405-0607-0809-0a0b0c0d0e0f
        %pUB    00010203-0405-0607-0809-0A0B0C0D0E0F
        %pUl    03020100-0504-0706-0809-0a0b0c0e0e0f
        %pUL    03020100-0504-0706-0809-0A0B0C0E0E0F

 For printing 16-byte UUID/GUIDs addresses. The additional 'l', 'L',
 'b' and 'B' specifiers are used to specify a little endian order in
 lower ('l') or upper case ('L') hex characters - and big endian order
 in lower ('b') or upper case ('B') hex characters.
```

```
Where no additional specifiers are used the default big endian
order with lower case hex characters will be printed.

Passed by reference.

dentry names
============

::

        %pd{,2,3,4}
        %pD{,2,3,4}

For printing dentry name; if we race with :c:func:`d_move`, the name might be
a mix of old and new ones, but it won't oops.  ``%pd`` dentry is a safer
equivalent of ``%s`` ``dentry->d_name.name`` we used to use, ``%pd<n>`` prints
``n`` last components.  ``%pD`` does the same thing for struct file.

Passed by reference.

block_device names
==================

::

        %pg     sda, sda1 or loop0p1

For printing name of block_device pointers.

struct va_format
================

::

        %pV

For printing struct va_format structures. These contain a format string
and va_list as follows::

        struct va_format {
                const char *fmt;
                va_list *va;
        };

Implements a "recursive vsnprintf".

Do not use this feature without some mechanism to verify the
correctness of the format string and va_list arguments.

Passed by reference.

kobjects
========

::

        %pO

        Base specifier for kobject based structs. Must be followed with
        character for specific type of kobject as listed below:

        Device tree nodes:

        %pOF[fnpPcCF]
```

```
        For printing device tree nodes. The optional arguments are:
            f device node full_name
            n device node name
            p device node phandle
            P device node path spec (name + @unit)
            F device node flags
            c major compatible string
            C full compatible string
        Without any arguments prints full_name (same as %pOFf)
        The separator when using multiple arguments is ':'

        Examples:

        %pOF    /foo/bar@0                      - Node full name
        %pOFf   /foo/bar@0                      - Same as above
        %pOFfp  /foo/bar@0:10                   - Node full name + phandle
        %pOFfcF /foo/bar@0:foo,device:--P-      - Node full name +
                                                  major compatible string +
                                                  node flags
                                                        D - dynamic
                                                        d - detached
                                                        P - Populated
                                                        B - Populated bus

        Passed by reference.


struct clk
==========

::

        %pC     pll1
        %pCn    pll1
        %pCr    1560000000

For printing struct clk structures. ``%pC`` and ``%pCn`` print the name
(Common Clock Framework) or address (legacy clock framework) of the
structure; ``%pCr`` prints the current clock rate.

Passed by reference.

bitmap and its derivatives such as cpumask and nodemask
=======================================================

::

        %*pb    0779
        %*pbl   0,3-6,8-10

For printing bitmap and its derivatives such as cpumask and nodemask,
``%*pb`` output the bitmap with field width as the number of bits and ``%*pbl``
output the bitmap as range list with field width as the number of bits.

Passed by reference.

Flags bitfields such as page flags, gfp_flags
=============================================

::

        %pGp    referenced|uptodate|lru|active|private
        %pGg    GFP_USER|GFP_DMA32|GFP_NOWARN
        %pGv    read|exec|mayread|maywrite|mayexec|denywrite
```

```
For printing flags bitfields as a collection of symbolic constants that
would construct the value. The type of flags is given by the third
character. Currently supported are [p]age flags, [v]ma_flags (both
expect ``unsigned long *``) and [g]fp_flags (expects ``gfp_t *``). The flag
names and print order depends on the particular type.

Note that this format should not be used directly in :c:func:`TP_printk()` part
of a tracepoint. Instead, use the ``show_*_flags()`` functions from
<trace/events/mmflags.h>.

Passed by reference.

Network device features
=======================

::

        %pNF    0x000000000000c000

For printing netdev_features_t.

Passed by reference.

If you add other ``%p`` extensions, please extend lib/test_printf.c with
one or more test cases, if at all feasible.


Thank you for your cooperation and attention.
```