```
**********************************************************
   79336 Thu Oct 16 18:42:19 2014
new/usr/src/cmd/mdb/common/mdb/mdb_print.c
5231 ::printf doesn't handle enums
Reviewed by: Robert Mustacchi <rm@joyent.com>
Reviewed by: Richard Lowe <richlowe@richlowe.net>
Reviewed by: Alex Reece <alex@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2012, 2014 by Delphix. All rights reserved.
 * Copyright (c) 2012 Joyent, Inc. All rights reserved.
 * Copyright (c) 2014 Nexenta Systems, Inc. All rights reserved.
#endif /* ! codereview */
 */

#include <mdb/mdb_modapi.h>
#include <mdb/mdb_target.h>
#include <mdb/mdb_argvec.h>
#include <mdb/mdb_string.h>
#include <mdb/mdb_stdlib.h>
#include <mdb/mdb_err.h>
#include <mdb/mdb_debug.h>
#include <mdb/mdb_fmt.h>
#include <mdb/mdb_ctf.h>
#include <mdb/mdb_ctf_impl.h>
#include <mdb/mdb.h>
#include <mdb/mdb_tab.h>

#include <sys/isa_defs.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <netinet/in.h>
#include <strings.h>
#include <libctf.h>
#include <ctype.h>

typedef struct holeinfo {
        ulong_t hi_offset;              /* expected offset */
        uchar_t hi_isunion;             /* represents a union */
} holeinfo_t;
```



```
typedef struct printarg {
        mdb_tgt_t *pa_tgt;              /* current target */
        mdb_tgt_t *pa_realtgt;          /* real target (for -i) */
        mdb_tgt_t *pa_immtgt;           /* immediate target (for -i) */
        mdb_tgt_as_t pa_as;             /* address space to use for i/o */
        mdb_tgt_addr_t pa_addr;         /* base address for i/o */
        ulong_t pa_armemlim;            /* limit on array elements to print */
        ulong_t pa_arstrlim;            /* limit on array chars to print */
        const char *pa_delim;           /* element delimiter string */
        const char *pa_prefix;          /* element prefix string */
        const char *pa_suffix;          /* element suffix string */
        holeinfo_t *pa_holes;           /* hole detection information */
        int pa_nholes;                  /* size of holes array */
        int pa_flags;                   /* formatting flags (see below) */
        int pa_depth;                   /* previous depth */
        int pa_nest;                    /* array nesting depth */
        int pa_tab;                     /* tabstop width */
        uint_t pa_maxdepth;             /* Limit max depth */
        uint_t pa_nooutdepth;           /* don't print output past this depth */
} printarg_t;

#define PA_SHOWTYPE     0x001           /* print type name */
#define PA_SHOWBASETYPE 0x002           /* print base type name */
#define PA_SHOWNAME     0x004           /* print member name */
#define PA_SHOWADDR     0x008           /* print address */
#define PA_SHOWVAL      0x010           /* print value */
#define PA_SHOWHOLES    0x020           /* print holes in structs */
#define PA_INTHEX       0x040           /* print integer values in hex */
#define PA_INTDEC       0x080           /* print integer values in decimal */
#define PA_NOSYMBOLIC   0x100           /* don't print ptrs as func+offset */

#define IS_CHAR(e) \
        (((e).cte_format & (CTF_INT_CHAR | CTF_INT_SIGNED)) == \
        (CTF_INT_CHAR | CTF_INT_SIGNED) && (e).cte_bits == NBBY)

#define COMPOSITE_MASK  ((1 << CTF_K_STRUCT) | \
                         (1 << CTF_K_UNION) | (1 << CTF_K_ARRAY))
#define IS_COMPOSITE(k) (((1 << k) & COMPOSITE_MASK) != 0)

#define SOU_MASK        ((1 << CTF_K_STRUCT) | (1 << CTF_K_UNION))
#define IS_SOU(k)       (((1 << k) & SOU_MASK) != 0)

#define MEMBER_DELIM_ERR        -1
#define MEMBER_DELIM_DONE       0
#define MEMBER_DELIM_PTR        1
#define MEMBER_DELIM_DOT        2
#define MEMBER_DELIM_LBR        3

typedef int printarg_f(const char *, const char *,
    mdb_ctf_id_t, mdb_ctf_id_t, ulong_t, printarg_t *);

static int elt_print(const char *, mdb_ctf_id_t, mdb_ctf_id_t, ulong_t, int,
    void *);
static void print_close_sou(printarg_t *, int);

/*
 * Given an address, look up the symbol ID of the specified symbol in its
 * containing module.  We only support lookups for exact matches.
 */
static const char *
addr_to_sym(mdb_tgt_t *t, uintptr_t addr, char *name, size_t namelen,
    GElf_Sym *symp, mdb_syminfo_t *sip)
{
        const mdb_map_t *mp;
        const char *p;
```

```
125         if (mdb_tgt_lookup_by_addr(t, addr, MDB_TGT_SYM_EXACT, name,
126             namelen, NULL, NULL) == -1)
127                 return (NULL); /* address does not exactly match a symbol */
128
129         if ((p = strrsplit(name, '`')) != NULL) {
130                 if (mdb_tgt_lookup_by_name(t, name, p, symp, sip) == -1)
131                         return (NULL);
132                 return (p);
133         }
134
135         if ((mp = mdb_tgt_addr_to_map(t, addr)) == NULL)
136                 return (NULL); /* address does not fall within a mapping */
137
138         if (mdb_tgt_lookup_by_name(t, mp->map_name, name, symp, sip) == -1)
139                 return (NULL);
140
141         return (name);
142 }
143
144 /*
145  * This lets dcmds be a little fancy with their processing of type arguments
146  * while still treating them more or less as a single argument.
147  * For example, if a command is invokes like this:
148  *
149  *   ::<dcmd> proc_t ...
150  *
151  * this function will just copy "proc_t" into the provided buffer. If the
152  * command is instead invoked like this:
153  *
154  *   ::<dcmd> struct proc ...
155  *
156  * this function will place the string "struct proc" into the provided buffer
157  * and increment the caller's argv and argc. This allows the caller to still
158  * treat the type argument logically as it would an other atomic argument.
159  */
160 int
161 args_to_typename(int *argcp, const mdb_arg_t **argvp, char *buf, size_t len)
162 {
163         int argc = *argcp;
164         const mdb_arg_t *argv = *argvp;
165
166         if (argc < 1 || argv->a_type != MDB_TYPE_STRING)
167                 return (DCMD_USAGE);
168
169         if (strcmp(argv->a_un.a_str, "struct") == 0 ||
170             strcmp(argv->a_un.a_str, "enum") == 0 ||
171             strcmp(argv->a_un.a_str, "union") == 0) {
172                 if (argc <= 1) {
173                         mdb_warn("%s is not a valid type\n", argv->a_un.a_str);
174                         return (DCMD_ABORT);
175                 }
176
177                 if (argv[1].a_type != MDB_TYPE_STRING)
178                         return (DCMD_USAGE);
179
180                 (void) mdb_snprintf(buf, len, "%s %s",
181                     argv[0].a_un.a_str, argv[1].a_un.a_str);
182
183                 *argcp = argc - 1;
184                 *argvp = argv + 1;
185         } else {
186                 (void) mdb_snprintf(buf, len, "%s", argv[0].a_un.a_str);
187         }
188
189         return (0);
190 }
```



```
192 /*ARGSUSED*/
193 int
194 cmd_sizeof(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
195 {
196         mdb_ctf_id_t id;
197         char tn[MDB_SYM_NAMLEN];
198         int ret;
199
200         if (flags & DCMD_ADDRSPEC)
201                 return (DCMD_USAGE);
202
203         if ((ret = args_to_typename(&argc, &argv, tn, sizeof (tn))) != 0)
204                 return (ret);
205
206         if (argc != 1)
207                 return (DCMD_USAGE);
208
209         if (mdb_ctf_lookup_by_name(tn, &id) != 0) {
210                 mdb_warn("failed to look up type %s", tn);
211                 return (DCMD_ERR);
212         }
213
214         if (flags & DCMD_PIPE_OUT)
215                 mdb_printf("%#lr\n", mdb_ctf_type_size(id));
216         else
217                 mdb_printf("sizeof (%s) = %#lr\n", tn, mdb_ctf_type_size(id));
218
219         return (DCMD_OK);
220 }
221
222 int
223 cmd_sizeof_tab(mdb_tab_cookie_t *mcp, uint_t flags, int argc,
224     const mdb_arg_t *argv)
225 {
226         char tn[MDB_SYM_NAMLEN];
227         int ret;
228
229         if (argc == 0 && !(flags & DCMD_TAB_SPACE))
230                 return (0);
231
232         if (argc == 0 && (flags & DCMD_TAB_SPACE))
233                 return (mdb_tab_complete_type(mcp, NULL, MDB_TABC_NOPOINT));
234
235         if ((ret = mdb_tab_typename(&argc, &argv, tn, sizeof (tn))) < 0)
236                 return (ret);
237
238         if (argc == 1)
239                 return (mdb_tab_complete_type(mcp, tn, MDB_TABC_NOPOINT));
240
241         return (0);
242 }
243
244 /*ARGSUSED*/
245 int
246 cmd_offsetof(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
247 {
248         const char *member;
249         mdb_ctf_id_t id;
250         ulong_t off;
251         char tn[MDB_SYM_NAMLEN];
252         ssize_t sz;
253         int ret;
254
255         if (flags & DCMD_ADDRSPEC)
256                 return (DCMD_USAGE);
```

```
258         if ((ret = args_to_typename(&argc, &argv, tn, sizeof (tn))) != 0)
259                 return (ret);
260
261         if (argc != 2 || argv[1].a_type != MDB_TYPE_STRING)
262                 return (DCMD_USAGE);
263
264         if (mdb_ctf_lookup_by_name(tn, &id) != 0) {
265                 mdb_warn("failed to look up type %s", tn);
266                 return (DCMD_ERR);
267         }
268
269         member = argv[1].a_un.a_str;
270
271         if (mdb_ctf_member_info(id, member, &off, &id) != 0) {
272                 mdb_warn("failed to find member %s of type %s", member, tn);
273                 return (DCMD_ERR);
274         }
275
276         if (flags & DCMD_PIPE_OUT) {
277                 if (off % NBBY != 0) {
278                         mdb_warn("member %s of type %s is not byte-aligned\n",
279                             member, tn);
280                         return (DCMD_ERR);
281                 }
282                 mdb_printf("%#lr", off / NBBY);
283                 return (DCMD_OK);
284         }
285
286         mdb_printf("offsetof (%s, %s) = %#lr",
287             tn, member, off / NBBY);
288         if (off % NBBY != 0)
289                 mdb_printf(".%lr", off % NBBY);
290
291         if ((sz = mdb_ctf_type_size(id)) > 0)
292                 mdb_printf(", sizeof (...->%s) = %#lr", member, sz);
293
294         mdb_printf("\n");
295
296         return (DCMD_OK);
297 }
298
299 /*ARGSUSED*/
300 static int
301 enum_prefix_scan_cb(const char *name, int value, void *arg)
302 {
303         char *str = arg;
304
305         /*
306          * This function is called with every name in the enum.  We make
307          * "arg" be the common prefix, if any.
308          */
309         if (str[0] == 0) {
310                 if (strlcpy(arg, name, MDB_SYM_NAMLEN) >= MDB_SYM_NAMLEN)
311                         return (1);
312                 return (0);
313         }
314
315         while (*name == *str) {
316                 if (*str == 0) {
317                         if (str != arg) {
318                                 str--;  /* don't smother a name completely */
319                         }
320                         break;
321                 }
322                 name++;
```



```
323                 str++;
324         }
325         *str = 0;
326
327         return (str == arg);    /* only continue if prefix is non-empty */
328 }
329
330 struct enum_p2_info {
331         intmax_t e_value;       /* value we're processing */
332         char    *e_buf;         /* buffer for holding names */
333         size_t  e_size;         /* size of buffer */
334         size_t  e_prefix;       /* length of initial prefix */
335         uint_t  e_allprefix;    /* apply prefix to first guy, too */
336         uint_t  e_bits;         /* bits seen */
337         uint8_t e_found;        /* have we seen anything? */
338         uint8_t e_first;        /* does buf contain the first one? */
339         uint8_t e_zero;         /* have we seen a zero value? */
340 };
341
342 static int
343 enum_p2_cb(const char *name, int bit_arg, void *arg)
344 {
345         struct enum_p2_info *eiip = arg;
346         uintmax_t bit = bit_arg;
347
348         if (bit != 0 && !ISP2(bit))
349                 return (1);     /* non-power-of-2; abort processing */
350
351         if ((bit == 0 && eiip->e_zero) ||
352             (bit != 0 && (eiip->e_bits & bit) != 0)) {
353                 return (0);     /* already seen this value */
354         }
355
356         if (bit == 0)
357                 eiip->e_zero = 1;
358         else
359                 eiip->e_bits |= bit;
360
361         if (eiip->e_buf != NULL && (eiip->e_value & bit) != 0) {
362                 char *buf = eiip->e_buf;
363                 size_t prefix = eiip->e_prefix;
364
365                 if (eiip->e_found) {
366                         (void) strlcat(buf, "|", eiip->e_size);
367
368                         if (eiip->e_first && !eiip->e_allprefix && prefix > 0) {
369                                 char c1 = buf[prefix];
370                                 char c2 = buf[prefix + 1];
371                                 buf[prefix] = '{';
372                                 buf[prefix + 1] = 0;
373                                 mdb_printf("%s", buf);
374                                 buf[prefix] = c1;
375                                 buf[prefix + 1] = c2;
376                                 mdb_printf("%s", buf + prefix);
377                         } else {
378                                 mdb_printf("%s", buf);
379                         }
380
381                 }
382                 /* skip the common prefix as necessary */
383                 if ((eiip->e_found || eiip->e_allprefix) &&
384                     strlen(name) > prefix)
385                         name += prefix;
386
387                 (void) strlcpy(eiip->e_buf, name, eiip->e_size);
388                 eiip->e_first = !eiip->e_found;
```

```
389                 eiip->e_found = 1;
390         }
391         return (0);
392 }

394 static int
395 enum_is_p2(mdb_ctf_id_t id)
396 {
397         struct enum_p2_info eii;
398         bzero(&eii, sizeof (eii));

400         return (mdb_ctf_type_kind(id) == CTF_K_ENUM &&
401             mdb_ctf_enum_iter(id, enum_p2_cb, &eii) == 0 &&
402             eii.e_bits != 0);
403 }

405 static int
406 enum_value_print_p2(mdb_ctf_id_t id, intmax_t value, uint_t allprefix)
407 {
408         struct enum_p2_info eii;
409         char prefix[MDB_SYM_NAMLEN + 2];
410         intmax_t missed;

412         bzero(&eii, sizeof (eii));

414         eii.e_value = value;
415         eii.e_buf = prefix;
416         eii.e_size = sizeof (prefix);
417         eii.e_allprefix = allprefix;

419         prefix[0] = 0;
420         if (mdb_ctf_enum_iter(id, enum_prefix_scan_cb, prefix) == 0)
421                 eii.e_prefix = strlen(prefix);

423         if (mdb_ctf_enum_iter(id, enum_p2_cb, &eii) != 0 || eii.e_bits == 0)
424                 return (-1);

426         missed = (value & ~(intmax_t)eii.e_bits);

428         if (eii.e_found) {
429                 /* push out any final value, with a | if we missed anything */
430                 if (!eii.e_first)
431                         (void) strlcat(prefix, "}", sizeof (prefix));
432                 if (missed != 0)
433                         (void) strlcat(prefix, "|", sizeof (prefix));

435                 mdb_printf("%s", prefix);
436         }

438         if (!eii.e_found || missed) {
439                 mdb_printf("%#llx", missed);
440         }

442         return (0);
443 }

445 struct enum_cbinfo {
446         uint_t          e_flags;
447         const char      *e_string;      /* NULL for value searches */
448         size_t          e_prefix;
449         intmax_t        e_value;
450         uint_t          e_found;
451         mdb_ctf_id_t    e_id;
452 };
453 #define E_PRETTY                0x01
454 #define E_HEX                   0x02
```



```
455 #define E_SEARCH_STRING         0x04
456 #define E_SEARCH_VALUE          0x08
457 #define E_ELIDE_PREFIX          0x10

459 static void
460 enum_print(struct enum_cbinfo *info, const char *name, int value)
461 {
462         uint_t flags = info->e_flags;
463         uint_t elide_prefix = (info->e_flags & E_ELIDE_PREFIX);

465         if (name != NULL && info->e_prefix && strlen(name) > info->e_prefix)
466                 name += info->e_prefix;

468         if (flags & E_PRETTY) {
469                 uint_t indent = 5 + ((flags & E_HEX) ? 8 : 11);

471                 mdb_printf((flags & E_HEX)? "%8x " : "%11d ", value);
472                 (void) mdb_inc_indent(indent);
473                 if (name != NULL) {
474                         mdb_iob_puts(mdb.m_out, name);
475                 } else {
476                         (void) enum_value_print_p2(info->e_id, value,
477                             elide_prefix);
478                 }
479                 (void) mdb_dec_indent(indent);
480                 mdb_printf("\n");
481         } else {
482                 mdb_printf("%#r\n", value);
483         }
484 }

486 static int
487 enum_cb(const char *name, int value, void *arg)
488 {
489         struct enum_cbinfo *info = arg;
490         uint_t flags = info->e_flags;

492         if (flags & E_SEARCH_STRING) {
493                 if (strcmp(name, info->e_string) != 0)
494                         return (0);

496         } else if (flags & E_SEARCH_VALUE) {
497                 if (value != info->e_value)
498                         return (0);
499         }

501         enum_print(info, name, value);

503         info->e_found = 1;
504         return (0);
505 }

507 void
508 enum_help(void)
509 {
510         mdb_printf("%s",
511 "Without an address and name, print all values for the enumeration \"enum\".\n"
512 "With an address, look up a particular value in \"enum\".  With a name, look\n"
513 "up a particular name in \"enum\".\n");

515         (void) mdb_dec_indent(2);
516         mdb_printf("\n%<b>OPTIONS%</b>\n");
517         (void) mdb_inc_indent(2);

519         mdb_printf("%s",
520 "   -e    remove common prefixes from enum names\n"
```

```
"   -x    report enum values in hexadecimal\n");
}

/*ARGSUSED*/
int
cmd_enum(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
{
        struct enum_cbinfo info;

        char type[MDB_SYM_NAMLEN + sizeof ("enum ")];
        char tn2[MDB_SYM_NAMLEN + sizeof ("enum ")];
        char prefix[MDB_SYM_NAMLEN];
        mdb_ctf_id_t id;
        mdb_ctf_id_t idr;

        int i;
        intmax_t search;
        uint_t isp2;

        info.e_flags = (flags & DCMD_PIPE_OUT)? 0 : E_PRETTY;
        info.e_string = NULL;
        info.e_value = 0;
        info.e_found = 0;

        i = mdb_getopts(argc, argv,
            'e', MDB_OPT_SETBITS, E_ELIDE_PREFIX, &info.e_flags,
            'x', MDB_OPT_SETBITS, E_HEX, &info.e_flags,
            NULL);

        argc -= i;
        argv += i;

        if ((i = args_to_typename(&argc, &argv, type, MDB_SYM_NAMLEN)) != 0)
                return (i);

        if (strchr(type, ' ') == NULL) {
                /*
                 * Check as an enumeration tag first, and fall back
                 * to checking for a typedef.  Yes, this means that
                 * anonymous enumerations whose typedefs conflict with
                 * an enum tag can't be accessed.  Don't do that.
                 */
                (void) mdb_snprintf(tn2, sizeof (tn2), "enum %s", type);

                if (mdb_ctf_lookup_by_name(tn2, &id) == 0) {
                        (void) strcpy(type, tn2);
                } else if (mdb_ctf_lookup_by_name(type, &id) != 0) {
                        mdb_warn("types '%s', '%s'", tn2, type);
                        return (DCMD_ERR);
                }
        } else {
                if (mdb_ctf_lookup_by_name(type, &id) != 0) {
                        mdb_warn("'%s'", type);
                        return (DCMD_ERR);
                }
        }

        /* resolve it, and make sure we're looking at an enumeration */
        if (mdb_ctf_type_resolve(id, &idr) == -1) {
                mdb_warn("unable to resolve '%s'", type);
                return (DCMD_ERR);
        }
        if (mdb_ctf_type_kind(idr) != CTF_K_ENUM) {
                mdb_warn("'%s': not an enumeration\n", type);
                return (DCMD_ERR);
        }
```



```
        info.e_id = idr;

        if (argc > 2)
                return (DCMD_USAGE);

        if (argc == 2) {
                if (flags & DCMD_ADDRSPEC) {
                        mdb_warn("may only specify one of: name, address\n");
                        return (DCMD_USAGE);
                }

                if (argv[1].a_type == MDB_TYPE_STRING) {
                        info.e_flags |= E_SEARCH_STRING;
                        info.e_string = argv[1].a_un.a_str;
                } else if (argv[1].a_type == MDB_TYPE_IMMEDIATE) {
                        info.e_flags |= E_SEARCH_VALUE;
                        search = argv[1].a_un.a_val;
                } else {
                        return (DCMD_USAGE);
                }
        }

        if (flags & DCMD_ADDRSPEC) {
                info.e_flags |= E_SEARCH_VALUE;
                search = mdb_get_dot();
        }

        if (info.e_flags & E_SEARCH_VALUE) {
                if ((int)search != search) {
                        mdb_warn("value '%lld' out of enumeration range\n",
                            search);
                }
                info.e_value = search;
        }

        isp2 = enum_is_p2(idr);
        if (isp2)
                info.e_flags |= E_HEX;

        if (DCMD_HDRSPEC(flags) && (info.e_flags & E_PRETTY)) {
                if (info.e_flags & E_HEX)
                        mdb_printf("%<u>%8s %-64s%</u>\n", "VALUE", "NAME");
                else
                        mdb_printf("%<u>%11s %-64s%</u>\n", "VALUE", "NAME");
        }

        /* if the enum is a power-of-two one, process it that way */
        if ((info.e_flags & E_SEARCH_VALUE) && isp2) {
                enum_print(&info, NULL, info.e_value);
                return (DCMD_OK);
        }

        prefix[0] = 0;
        if ((info.e_flags & E_ELIDE_PREFIX) &&
            mdb_ctf_enum_iter(id, enum_prefix_scan_cb, prefix) == 0)
                info.e_prefix = strlen(prefix);

        if (mdb_ctf_enum_iter(idr, enum_cb, &info) == -1) {
                mdb_warn("cannot walk '%s' as enum", type);
                return (DCMD_ERR);
        }

        if (info.e_found == 0 &&
            (info.e_flags & (E_SEARCH_STRING | E_SEARCH_VALUE)) != 0) {
                if (info.e_flags & E_SEARCH_STRING)
```

```
653                         mdb_warn("name \"%s\" not in '%s'\n", info.e_string,
654                             type);
655                 else
656                         mdb_warn("value %#lld not in '%s'\n", info.e_value,
657                             type);
658
659                 return (DCMD_ERR);
660         }
661
662         return (DCMD_OK);
663 }
664
665 static int
666 setup_vcb(const char *name, uintptr_t addr)
667 {
668         const char *p;
669         mdb_var_t *v;
670
671         if ((v = mdb_nv_lookup(&mdb.m_nv, name)) == NULL) {
672                 if ((p = strbadid(name)) != NULL) {
673                         mdb_warn("'%c' may not be used in a variable "
674                             "name\n", *p);
675                         return (DCMD_ABORT);
676                 }
677
678                 if ((v = mdb_nv_insert(&mdb.m_nv, name, NULL, addr, 0)) == NULL)
679                         return (DCMD_ERR);
680         } else {
681                 if (v->v_flags & MDB_NV_RDONLY) {
682                         mdb_warn("variable %s is read-only\n", name);
683                         return (DCMD_ABORT);
684                 }
685         }
686
687         /*
688          * If there already exists a vcb for this variable, we may be
689          * calling the dcmd in a loop.  We only create a vcb for this
690          * variable on the first invocation.
691          */
692         if (mdb_vcb_find(v, mdb.m_frame) == NULL)
693                 mdb_vcb_insert(mdb_vcb_create(v), mdb.m_frame);
694
695         return (0);
696 }
697
698 /*ARGSUSED*/
699 int
700 cmd_list(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
701 {
702         int offset;
703         uintptr_t a, tmp;
704         int ret;
705
706         if (!(flags & DCMD_ADDRSPEC) || argc == 0)
707                 return (DCMD_USAGE);
708
709         if (argv->a_type != MDB_TYPE_STRING) {
710                 /*
711                  * We are being given a raw offset in lieu of a type and
712                  * member; confirm the number of arguments and argument
713                  * type.
714                  */
715                 if (argc != 1 || argv->a_type != MDB_TYPE_IMMEDIATE)
716                         return (DCMD_USAGE);
717
718                 offset = argv->a_un.a_val;
```



```
719
720                 argv++;
721                 argc--;
722
723                 if (offset % sizeof (uintptr_t)) {
724                         mdb_warn("offset must fall on a word boundary\n");
725                         return (DCMD_ABORT);
726                 }
727         } else {
728                 const char *member;
729                 char buf[MDB_SYM_NAMLEN];
730                 int ret;
731
732                 /*
733                  * Check that we were provided 2 arguments: a type name
734                  * and a member of that type.
735                  */
736                 if (argc != 2)
737                         return (DCMD_USAGE);
738
739                 ret = args_to_typename(&argc, &argv, buf, sizeof (buf));
740                 if (ret != 0)
741                         return (ret);
742
743                 argv++;
744                 argc--;
745
746                 member = argv->a_un.a_str;
747                 offset = mdb_ctf_offsetof_by_name(buf, member);
748                 if (offset == -1)
749                         return (DCMD_ABORT);
750
751                 argv++;
752                 argc--;
753
754                 if (offset % (sizeof (uintptr_t)) != 0) {
755                         mdb_warn("%s is not a word-aligned member\n", member);
756                         return (DCMD_ABORT);
757                 }
758         }
759
760         /*
761          * If we have any unchewed arguments, a variable name must be present.
762          */
763         if (argc == 1) {
764                 if (argv->a_type != MDB_TYPE_STRING)
765                         return (DCMD_USAGE);
766
767                 if ((ret = setup_vcb(argv->a_un.a_str, addr)) != 0)
768                         return (ret);
769
770         } else if (argc != 0) {
771                 return (DCMD_USAGE);
772         }
773
774         a = addr;
775
776         do {
777                 mdb_printf("%lr\n", a);
778
779                 if (mdb_vread(&tmp, sizeof (tmp), a + offset) == -1) {
780                         mdb_warn("failed to read next pointer from object %p",
781                             a);
782                         return (DCMD_ERR);
783                 }
```

```
785                 a = tmp;
786         } while (a != addr && a != NULL);
787
788         return (DCMD_OK);
789 }
790
791 int
792 cmd_array(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
793 {
794         mdb_ctf_id_t id;
795         ssize_t elemsize = 0;
796         char tn[MDB_SYM_NAMLEN];
797         int ret, nelem = -1;
798
799         mdb_tgt_t *t = mdb.m_target;
800         GElf_Sym sym;
801         mdb_ctf_arinfo_t ar;
802         mdb_syminfo_t s_info;
803
804         if (!(flags & DCMD_ADDRSPEC))
805                 return (DCMD_USAGE);
806
807         if (argc >= 2) {
808                 ret = args_to_typename(&argc, &argv, tn, sizeof (tn));
809                 if (ret != 0)
810                         return (ret);
811
812                 if (argc == 1)  /* unquoted compound type without count */
813                         return (DCMD_USAGE);
814
815                 if (mdb_ctf_lookup_by_name(tn, &id) != 0) {
816                         mdb_warn("failed to look up type %s", tn);
817                         return (DCMD_ABORT);
818                 }
819
820                 if (argv[1].a_type == MDB_TYPE_IMMEDIATE)
821                         nelem = argv[1].a_un.a_val;
822                 else
823                         nelem = mdb_strtoull(argv[1].a_un.a_str);
824
825                 elemsize = mdb_ctf_type_size(id);
826         } else if (addr_to_sym(t, addr, tn, sizeof (tn), &sym, &s_info)
827             != NULL && mdb_ctf_lookup_by_symbol(&sym, &s_info, &id)
828             == 0 && mdb_ctf_type_kind(id) == CTF_K_ARRAY &&
829             mdb_ctf_array_info(id, &ar) != -1) {
830                 elemsize = mdb_ctf_type_size(id) / ar.mta_nelems;
831                 nelem = ar.mta_nelems;
832         } else {
833                 mdb_warn("no symbol information for %a", addr);
834                 return (DCMD_ERR);
835         }
836
837         if (argc == 3 || argc == 1) {
838                 if (argv[argc - 1].a_type != MDB_TYPE_STRING)
839                         return (DCMD_USAGE);
840
841                 if ((ret = setup_vcb(argv[argc - 1].a_un.a_str, addr)) != 0)
842                         return (ret);
843
844         } else if (argc > 3) {
845                 return (DCMD_USAGE);
846         }
847
848         for (; nelem > 0; nelem--) {
849                 mdb_printf("%lr\n", addr);
850                 addr = addr + elemsize;
```



```
851         }
852
853         return (DCMD_OK);
854 }
855
856 /*
857  * Print an integer bitfield in hexadecimal by reading the enclosing byte(s)
858  * and then shifting and masking the data in the lower bits of a uint64_t.
859  */
860 static int
861 print_bitfield(ulong_t off, printarg_t *pap, ctf_encoding_t *ep)
862 {
863         mdb_tgt_addr_t addr = pap->pa_addr + off / NBBY;
864         size_t size = (ep->cte_bits + (NBBY - 1)) / NBBY;
865         uint64_t mask = (1ULL << ep->cte_bits) - 1;
866         uint64_t value = 0;
867         uint8_t *buf = (uint8_t *)&value;
868         uint8_t shift;
869
870         const char *format;
871
872         if (!(pap->pa_flags & PA_SHOWVAL))
873                 return (0);
874
875         if (ep->cte_bits > sizeof (value) * NBBY - 1) {
876                 mdb_printf("??? (invalid bitfield size %u)", ep->cte_bits);
877                 return (0);
878         }
879
880         /*
881          * On big-endian machines, we need to adjust the buf pointer to refer
882          * to the lowest 'size' bytes in 'value', and we need shift based on
883          * the offset from the end of the data, not the offset of the start.
884          */
885 #ifdef _BIG_ENDIAN
886         buf += sizeof (value) - size;
887         off += ep->cte_bits;
888 #endif
889         if (mdb_tgt_aread(pap->pa_tgt, pap->pa_as, buf, size, addr) != size) {
890                 mdb_warn("failed to read %lu bytes at %llx",
891                     (ulong_t)size, addr);
892                 return (1);
893         }
894
895         shift = off % NBBY;
896
897         /*
898          * Offsets are counted from opposite ends on little- and
899          * big-endian machines.
900          */
901 #ifdef _BIG_ENDIAN
902         shift = NBBY - shift;
903 #endif
904
905         /*
906          * If the bits we want do not begin on a byte boundary, shift the data
907          * right so that the value is in the lowest 'cte_bits' of 'value'.
908          */
909         if (off % NBBY != 0)
910                 value >>= shift;
911         value &= mask;
912
913         /*
914          * We default to printing signed bitfields as decimals,
915          * and unsigned bitfields in hexadecimal.  If they specify
916          * hexadecimal, we treat the field as unsigned.
```

```
917          */
918         if ((pap->pa_flags & PA_INTHEX) ||
919             !(ep->cte_format & CTF_INT_SIGNED)) {
920                 format = (pap->pa_flags & PA_INTDEC)? "%#llu" : "%#llx";
921         } else {
922                 int sshift = sizeof (value) * NBBY - ep->cte_bits;

924                 /* sign-extend value, and print as a signed decimal */
925                 value = ((int64_t)value << sshift) >> sshift;
926                 format = "%#lld";
927         }
928         mdb_printf(format, value);

930         return (0);
931 }

933 /*
934  * Print out a character or integer value.  We use some simple heuristics,
935  * described below, to determine the appropriate radix to use for output.
936  */
937 static int
938 print_int_val(const char *type, ctf_encoding_t *ep, ulong_t off,
939     printarg_t *pap)
940 {
941         static const char *const sformat[] = { "%#d", "%#d", "%#d", "%#lld" };
942         static const char *const uformat[] = { "%#u", "%#u", "%#u", "%#llu" };
943         static const char *const xformat[] = { "%#x", "%#x", "%#x", "%#llx" };

945         mdb_tgt_addr_t addr = pap->pa_addr + off / NBBY;
946         const char *const *fsp;
947         size_t size;

949         union {
950                 uint64_t i8;
951                 uint32_t i4;
952                 uint16_t i2;
953                 uint8_t i1;
954                 time_t t;
955                 ipaddr_t I;
956         } u;

958         if (!(pap->pa_flags & PA_SHOWVAL))
959                 return (0);

961         if (ep->cte_format & CTF_INT_VARARGS) {
962                 mdb_printf("...\n");
963                 return (0);
964         }

966         /*
967          * If the size is not a power-of-two number of bytes in the range 1-8
968          * then we assume it is a bitfield and print it as such.
969          */
970         size = ep->cte_bits / NBBY;
971         if (size > 8 || (ep->cte_bits % NBBY) != 0 || (size & (size - 1)) != 0)
972                 return (print_bitfield(off, pap, ep));

974         if (IS_CHAR(*ep)) {
975                 mdb_printf("'");
976                 if (mdb_fmt_print(pap->pa_tgt, pap->pa_as,
977                     addr, 1, 'C') == addr)
978                         return (1);
979                 mdb_printf("'");
980                 return (0);
981         }
```



```
983         if (mdb_tgt_aread(pap->pa_tgt, pap->pa_as, &u.i8, size, addr) != size) {
984                 mdb_warn("failed to read %lu bytes at %llx",
985                     (ulong_t)size, addr);
986                 return (1);
987         }

989         /*
990          * We pretty-print some integer based types.  time_t values are
991          * printed as a calendar date and time, and IPv4 addresses as human
992          * readable dotted quads.
993          */
994         if (!(pap->pa_flags & (PA_INTHEX | PA_INTDEC))) {
995                 if (strcmp(type, "time_t") == 0 && u.t != 0) {
996                         mdb_printf("%Y", u.t);
997                         return (0);
998                 }
999                 if (strcmp(type, "ipaddr_t") == 0 ||
1000                     strcmp(type, "in_addr_t") == 0) {
1001                         mdb_printf("%I", u.I);
1002                         return (0);
1003                 }
1004         }

1006         /*
1007          * The default format is hexadecimal.
1008          */
1009         if (!(pap->pa_flags & PA_INTDEC))
1010                 fsp = xformat;
1011         else if (ep->cte_format & CTF_INT_SIGNED)
1012                 fsp = sformat;
1013         else
1014                 fsp = uformat;

1016         switch (size) {
1017         case sizeof (uint8_t):
1018                 mdb_printf(fsp[0], u.i1);
1019                 break;
1020         case sizeof (uint16_t):
1021                 mdb_printf(fsp[1], u.i2);
1022                 break;
1023         case sizeof (uint32_t):
1024                 mdb_printf(fsp[2], u.i4);
1025                 break;
1026         case sizeof (uint64_t):
1027                 mdb_printf(fsp[3], u.i8);
1028                 break;
1029         }
1030         return (0);
1031 }

1033 /*ARGSUSED*/
1034 static int
1035 print_int(const char *type, const char *name, mdb_ctf_id_t id,
1036     mdb_ctf_id_t base, ulong_t off, printarg_t *pap)
1037 {
1038         ctf_encoding_t e;

1040         if (!(pap->pa_flags & PA_SHOWVAL))
1041                 return (0);

1043         if (mdb_ctf_type_encoding(base, &e) != 0) {
1044                 mdb_printf("??? (%s)", mdb_strerror(errno));
1045                 return (0);
1046         }

1048         return (print_int_val(type, &e, off, pap));
```

```
}

/*
 * Print out a floating point value.  We only provide support for floats in
 * the ANSI-C float, double, and long double formats.
 */
/*ARGSUSED*/
static int
print_float(const char *type, const char *name, mdb_ctf_id_t id,
    mdb_ctf_id_t base, ulong_t off, printarg_t *pap)
{
#ifndef _KMDB
        mdb_tgt_addr_t addr = pap->pa_addr + off / NBBY;
        ctf_encoding_t e;

        union {
                float f;
                double d;
                long double ld;
        } u;

        if (!(pap->pa_flags & PA_SHOWVAL))
                return (0);

        if (mdb_ctf_type_encoding(base, &e) == 0) {
                if (e.cte_format == CTF_FP_SINGLE &&
                    e.cte_bits == sizeof (float) * NBBY) {
                        if (mdb_tgt_aread(pap->pa_tgt, pap->pa_as, &u.f,
                            sizeof (u.f), addr) != sizeof (u.f)) {
                                mdb_warn("failed to read float at %llx", addr);
                                return (1);
                        }
                        mdb_printf("%s", doubletos(u.f, 7, 'e'));

                } else if (e.cte_format == CTF_FP_DOUBLE &&
                    e.cte_bits == sizeof (double) * NBBY) {
                        if (mdb_tgt_aread(pap->pa_tgt, pap->pa_as, &u.d,
                            sizeof (u.d), addr) != sizeof (u.d)) {
                                mdb_warn("failed to read float at %llx", addr);
                                return (1);
                        }
                        mdb_printf("%s", doubletos(u.d, 7, 'e'));

                } else if (e.cte_format == CTF_FP_LDOUBLE &&
                    e.cte_bits == sizeof (long double) * NBBY) {
                        if (mdb_tgt_aread(pap->pa_tgt, pap->pa_as, &u.ld,
                            sizeof (u.ld), addr) != sizeof (u.ld)) {
                                mdb_warn("failed to read float at %llx", addr);
                                return (1);
                        }
                        mdb_printf("%s", longdoubletos(&u.ld, 16, 'e'));

                } else {
                        mdb_printf("??? (unsupported FP format %u / %u bits\n",
                            e.cte_format, e.cte_bits);
                }
        } else
                mdb_printf("??? (%s)", mdb_strerror(errno));
#else
        mdb_printf("<FLOAT>");
#endif
        return (0);
}


/*
```



```
 * Print out a pointer value as a symbol name + offset or a hexadecimal value.
 * If the pointer itself is a char *, we attempt to read a bit of the data
 * referenced by the pointer and display it if it is a printable ASCII string.
 */
/*ARGSUSED*/
static int
print_ptr(const char *type, const char *name, mdb_ctf_id_t id,
    mdb_ctf_id_t base, ulong_t off, printarg_t *pap)
{
        mdb_tgt_addr_t addr = pap->pa_addr + off / NBBY;
        ctf_encoding_t e;
        uintptr_t value;
        char buf[256];
        ssize_t len;

        if (!(pap->pa_flags & PA_SHOWVAL))
                return (0);

        if (mdb_tgt_aread(pap->pa_tgt, pap->pa_as,
            &value, sizeof (value), addr) != sizeof (value)) {
                mdb_warn("failed to read %s pointer at %llx", name, addr);
                return (1);
        }

        if (pap->pa_flags & PA_NOSYMBOLIC) {
                mdb_printf("%#lx", value);
                return (0);
        }

        mdb_printf("%a", value);

        if (value == NULL || strcmp(type, "caddr_t") == 0)
                return (0);

        if (mdb_ctf_type_kind(base) == CTF_K_POINTER &&
            mdb_ctf_type_reference(base, &base) != -1 &&
            mdb_ctf_type_resolve(base, &base) != -1 &&
            mdb_ctf_type_encoding(base, &e) == 0 && IS_CHAR(e)) {
                if ((len = mdb_tgt_readstr(pap->pa_realtgt, pap->pa_as,
                    buf, sizeof (buf), value)) >= 0 && strisprint(buf)) {
                        if (len == sizeof (buf))
                                (void) strabbr(buf, sizeof (buf));
                        mdb_printf(" \"%s\"", buf);
                }
        }

        return (0);
}


/*
 * Print out a fixed-size array.  We special-case arrays of characters
 * and attempt to print them out as ASCII strings if possible.  For other
 * arrays, we iterate over a maximum of pa_armemlim members and call
 * mdb_ctf_type_visit() again on each element to print its value.
 */
/*ARGSUSED*/
static int
print_array(const char *type, const char *name, mdb_ctf_id_t id,
    mdb_ctf_id_t base, ulong_t off, printarg_t *pap)
{
        mdb_tgt_addr_t addr = pap->pa_addr + off / NBBY;
        printarg_t pa = *pap;
        ssize_t eltsize;
        mdb_ctf_arinfo_t r;
        ctf_encoding_t e;
```

```
1181         uint_t i, kind, limit;
1182         int d, sou;
1183         char buf[8];
1184         char *str;

1186         if (!(pap->pa_flags & PA_SHOWVAL))
1187                 return (0);

1189         if (pap->pa_depth == pap->pa_maxdepth) {
1190                 mdb_printf("[ ... ]");
1191                 return (0);
1192         }

1194         /*
1195          * Determine the base type and size of the array's content.  If this
1196          * fails, we cannot print anything and just give up.
1197          */
1198         if (mdb_ctf_array_info(base, &r) == -1 ||
1199             mdb_ctf_type_resolve(r.mta_contents, &base) == -1 ||
1200             (eltsize = mdb_ctf_type_size(base)) == -1) {
1201                 mdb_printf("[ ??? ] (%s)", mdb_strerror(errno));
1202                 return (0);
1203         }

1205         /*
1206          * Read a few bytes and determine if the content appears to be
1207          * printable ASCII characters.  If so, read the entire array and
1208          * attempt to display it as a string if it is printable.
1209          */
1210         if ((pap->pa_arstrlim == MDB_ARR_NOLIMIT ||
1211             r.mta_nelems <= pap->pa_arstrlim) &&
1212             mdb_ctf_type_encoding(base, &e) == 0 && IS_CHAR(e) &&
1213             mdb_tgt_readstr(pap->pa_tgt, pap->pa_as, buf,
1214             MIN(sizeof (buf), r.mta_nelems), addr) > 0 && strisprint(buf)) {

1216                 str = mdb_alloc(r.mta_nelems + 1, UM_SLEEP | UM_GC);
1217                 str[r.mta_nelems] = '\0';

1219                 if (mdb_tgt_aread(pap->pa_tgt, pap->pa_as, str,
1220                     r.mta_nelems, addr) != r.mta_nelems) {
1221                         mdb_warn("failed to read char array at %llx", addr);
1222                         return (1);
1223                 }

1225                 if (strisprint(str)) {
1226                         mdb_printf("[ \"%s\" ]", str);
1227                         return (0);
1228                 }
1229         }

1231         if (pap->pa_armemlim != MDB_ARR_NOLIMIT)
1232                 limit = MIN(r.mta_nelems, pap->pa_armemlim);
1233         else
1234                 limit = r.mta_nelems;

1236         if (limit == 0) {
1237                 mdb_printf("[ ... ]");
1238                 return (0);
1239         }

1241         kind = mdb_ctf_type_kind(base);
1242         sou = IS_COMPOSITE(kind);

1244         pa.pa_addr = addr;              /* set base address to start of array */
1245         pa.pa_maxdepth = pa.pa_maxdepth - pa.pa_depth - 1;
1246         pa.pa_nest += pa.pa_depth + 1;  /* nesting level is current depth + 1 */
```



```
1247         pa.pa_depth = 0;                /* reset depth to 0 for new scope */
1248         pa.pa_prefix = NULL;

1250         if (sou) {
1251                 pa.pa_delim = "\n";
1252                 mdb_printf("[\n");
1253         } else {
1254                 pa.pa_flags &= ~(PA_SHOWTYPE | PA_SHOWNAME | PA_SHOWADDR);
1255                 pa.pa_delim = ", ";
1256                 mdb_printf("[ ");
1257         }

1259         for (i = 0; i < limit; i++, pa.pa_addr += eltsize) {
1260                 if (i == limit - 1 && !sou) {
1261                         if (limit < r.mta_nelems)
1262                                 pa.pa_delim = ", ... ]";
1263                         else
1264                                 pa.pa_delim = " ]";
1265                 }

1267                 if (mdb_ctf_type_visit(r.mta_contents, elt_print, &pa) == -1) {
1268                         mdb_warn("failed to print array data");
1269                         return (1);
1270                 }
1271         }

1273         if (sou) {
1274                 for (d = pa.pa_depth - 1; d >= 0; d--)
1275                         print_close_sou(&pa, d);

1277                 if (limit < r.mta_nelems) {
1278                         mdb_printf("%*s... ]",
1279                             (pap->pa_depth + pap->pa_nest) * pap->pa_tab, "");
1280                 } else {
1281                         mdb_printf("%*s]",
1282                             (pap->pa_depth + pap->pa_nest) * pap->pa_tab, "");
1283                 }
1284         }

1286         /* copy the hole array info, since it may have been grown */
1287         pap->pa_holes = pa.pa_holes;
1288         pap->pa_nholes = pa.pa_nholes;

1290         return (0);
1291 }

1293 /*
1294  * Print out a struct or union header.  We need only print the open brace
1295  * because mdb_ctf_type_visit() itself will automatically recurse through
1296  * all members of the given struct or union.
1297  */
1298 /*ARGSUSED*/
1299 static int
1300 print_sou(const char *type, const char *name, mdb_ctf_id_t id,
1301     mdb_ctf_id_t base, ulong_t off, printarg_t *pap)
1302 {
1303         mdb_tgt_addr_t addr = pap->pa_addr + off / NBBY;

1305         /*
1306          * We have pretty-printing for some structures where displaying
1307          * structure contents has no value.
1308          */
1309         if (pap->pa_flags & PA_SHOWVAL) {
1310                 if (strcmp(type, "in6_addr_t") == 0 ||
1311                     strcmp(type, "struct in6_addr") == 0) {
1312                         in6_addr_t in6addr;
```

```
                        if (mdb_tgt_aread(pap->pa_tgt, pap->pa_as, &in6addr,
                            sizeof (in6addr), addr) != sizeof (in6addr)) {
                                mdb_warn("failed to read %s pointer at %llx",
                                    name, addr);
                                return (1);
                        }
                        mdb_printf("%N", &in6addr);
                        /*
                         * Don't print anything further down in the
                         * structure.
                         */
                        pap->pa_nooutdepth = pap->pa_depth;
                        return (0);
                }
                if (strcmp(type, "struct in_addr") == 0) {
                        in_addr_t inaddr;

                        if (mdb_tgt_aread(pap->pa_tgt, pap->pa_as, &inaddr,
                            sizeof (inaddr), addr) != sizeof (inaddr)) {
                                mdb_warn("failed to read %s pointer at %llx",
                                    name, addr);
                                return (1);
                        }
                        mdb_printf("%I", inaddr);
                        pap->pa_nooutdepth = pap->pa_depth;
                        return (0);
                }
        }

        if (pap->pa_depth == pap->pa_maxdepth)
                mdb_printf("{ ... }");
        else
                mdb_printf("{");
        pap->pa_delim = "\n";
        return (0);
}

/*
 * Print an enum value.  We attempt to convert the value to the corresponding
 * enum name and print that if possible.
 */
/*ARGSUSED*/
static int
print_enum(const char *type, const char *name, mdb_ctf_id_t id,
    mdb_ctf_id_t base, ulong_t off, printarg_t *pap)
{
        mdb_tgt_addr_t addr = pap->pa_addr + off / NBBY;
        const char *ename;
        int value;
        int isp2 = enum_is_p2(base);
        int flags = pap->pa_flags | (isp2 ? PA_INTHEX : 0);

        if (!(flags & PA_SHOWVAL))
                return (0);

        if (mdb_tgt_aread(pap->pa_tgt, pap->pa_as,
            &value, sizeof (value), addr) != sizeof (value)) {
                mdb_warn("failed to read %s integer at %llx", name, addr);
                return (1);
        }

        if (flags & PA_INTHEX)
                mdb_printf("%#x", value);
        else
                mdb_printf("%#d", value);
```



```
        (void) mdb_inc_indent(8);
        mdb_printf(" (");

        if (!isp2 || enum_value_print_p2(base, value, 0) != 0) {
                ename = mdb_ctf_enum_name(base, value);
                if (ename == NULL) {
                        ename = "???";
                }
                mdb_printf("%s", ename);
        }
        mdb_printf(")");
        (void) mdb_dec_indent(8);

        return (0);
}

/*
 * This will only get called if the structure isn't found in any available CTF
 * data.
 */
/*ARGSUSED*/
static int
print_tag(const char *type, const char *name, mdb_ctf_id_t id,
    mdb_ctf_id_t base, ulong_t off, printarg_t *pap)
{
        char basename[MDB_SYM_NAMLEN];

        if (pap->pa_flags & PA_SHOWVAL)
                mdb_printf("; ");

        if (mdb_ctf_type_name(base, basename, sizeof (basename)) != NULL)
                mdb_printf("<forward declaration of %s>", basename);
        else
                mdb_printf("<forward declaration of unknown type>");

        return (0);
}

static void
print_hole(printarg_t *pap, int depth, ulong_t off, ulong_t endoff)
{
        ulong_t bits = endoff - off;
        ulong_t size = bits / NBBY;
        ctf_encoding_t e;

        static const char *const name = "<<HOLE>>";
        char type[MDB_SYM_NAMLEN];

        int bitfield =
            (off % NBBY != 0 ||
            bits % NBBY != 0 ||
            size > 8 ||
            (size & (size - 1)) != 0);

        ASSERT(off < endoff);

        if (bits > NBBY * sizeof (uint64_t)) {
                ulong_t end;

                /*
                 * The hole is larger than the largest integer type.  To
                 * handle this, we split up the hole at 8-byte-aligned
                 * boundaries, recursing to print each subsection.  For
                 * normal C structures, we'll loop at most twice.
                 */
```

```
1445                 for (; off < endoff; off = end) {
1446                         end = P2END(off, NBBY * sizeof (uint64_t));
1447                         if (end > endoff)
1448                                 end = endoff;
1449 
1450                         ASSERT((end - off) <= NBBY * sizeof (uint64_t));
1451                         print_hole(pap, depth, off, end);
1452                 }
1453                 ASSERT(end == endoff);
1454 
1455                 return;
1456         }
1457 
1458         if (bitfield)
1459                 (void) mdb_snprintf(type, sizeof (type), "unsigned");
1460         else
1461                 (void) mdb_snprintf(type, sizeof (type), "uint%d_t", bits);
1462 
1463         if (pap->pa_flags & (PA_SHOWTYPE | PA_SHOWNAME | PA_SHOWADDR))
1464                 mdb_printf("%*s", (depth + pap->pa_nest) * pap->pa_tab, "");
1465 
1466         if (pap->pa_flags & PA_SHOWADDR) {
1467                 if (off % NBBY == 0)
1468                         mdb_printf("%llx ", pap->pa_addr + off / NBBY);
1469                 else
1470                         mdb_printf("%llx.%lx ",
1471                             pap->pa_addr + off / NBBY, off % NBBY);
1472         }
1473 
1474         if (pap->pa_flags & PA_SHOWTYPE)
1475                 mdb_printf("%s ", type);
1476 
1477         if (pap->pa_flags & PA_SHOWNAME)
1478                 mdb_printf("%s", name);
1479 
1480         if (bitfield && (pap->pa_flags & PA_SHOWTYPE))
1481                 mdb_printf(" :%d", bits);
1482 
1483         mdb_printf("%s ", (pap->pa_flags & PA_SHOWVAL)? " =" : "");
1484 
1485         /*
1486          * We fake up a ctf_encoding_t, and use print_int_val() to print
1487          * the value.  Holes are always processed as unsigned integers.
1488          */
1489         bzero(&e, sizeof (e));
1490         e.cte_format = 0;
1491         e.cte_offset = 0;
1492         e.cte_bits = bits;
1493 
1494         if (print_int_val(type, &e, off, pap) != 0)
1495                 mdb_iob_discard(mdb.m_out);
1496         else
1497                 mdb_iob_puts(mdb.m_out, pap->pa_delim);
1498 }
1499 
1500 /*
1501  * The print_close_sou() function is called for each structure or union
1502  * which has been completed.  For structures, we detect and print any holes
1503  * before printing the closing brace.
1504  */
1505 static void
1506 print_close_sou(printarg_t *pap, int newdepth)
1507 {
1508         int d = newdepth + pap->pa_nest;
1509 
1510         if ((pap->pa_flags & PA_SHOWHOLES) && !pap->pa_holes[d].hi_isunion) {
```



```
1511                 ulong_t end = pap->pa_holes[d + 1].hi_offset;
1512                 ulong_t expected = pap->pa_holes[d].hi_offset;
1513 
1514                 if (end < expected)
1515                         print_hole(pap, newdepth + 1, end, expected);
1516         }
1517         /* if the struct is an array element, print a comma after the } */
1518         mdb_printf("%*s}%s\n", d * pap->pa_tab, "",
1519             (newdepth == 0 && pap->pa_nest > 0)? "," : "");
1520 }
1521 
1522 static printarg_f *const printfuncs[] = {
1523         print_int,      /* CTF_K_INTEGER */
1524         print_float,    /* CTF_K_FLOAT */
1525         print_ptr,      /* CTF_K_POINTER */
1526         print_array,    /* CTF_K_ARRAY */
1527         print_ptr,      /* CTF_K_FUNCTION */
1528         print_sou,      /* CTF_K_STRUCT */
1529         print_sou,      /* CTF_K_UNION */
1530         print_enum,     /* CTF_K_ENUM */
1531         print_tag       /* CTF_K_FORWARD */
1532 };
1533 
1534 /*
1535  * The elt_print function is used as the mdb_ctf_type_visit callback.  For
1536  * each element, we print an appropriate name prefix and then call the
1537  * print subroutine for this type class in the array above.
1538  */
1539 static int
1540 elt_print(const char *name, mdb_ctf_id_t id, mdb_ctf_id_t base,
1541     ulong_t off, int depth, void *data)
1542 {
1543         char type[MDB_SYM_NAMLEN + sizeof (" <<12345678...>>")];
1544         int kind, rc, d;
1545         printarg_t *pap = data;
1546 
1547         for (d = pap->pa_depth - 1; d >= depth; d--) {
1548                 if (d < pap->pa_nooutdepth)
1549                         print_close_sou(pap, d);
1550         }
1551 
1552         /*
1553          * Reset pa_nooutdepth if we've come back out of the structure we
1554          * didn't want to print.
1555          */
1556         if (depth <= pap->pa_nooutdepth)
1557                 pap->pa_nooutdepth = (uint_t)-1;
1558 
1559         if (depth > pap->pa_maxdepth || depth > pap->pa_nooutdepth)
1560                 return (0);
1561 
1562         if (!mdb_ctf_type_valid(base) ||
1563             (kind = mdb_ctf_type_kind(base)) == -1)
1564                 return (-1); /* errno is set for us */
1565 
1566         if (mdb_ctf_type_name(id, type, MDB_SYM_NAMLEN) == NULL)
1567                 (void) strcpy(type, "(?)");
1568 
1569         if (pap->pa_flags & PA_SHOWBASETYPE) {
1570                 /*
1571                  * If basetype is different and informative, concatenate
1572                  * <<basetype>> (or <<baset...>> if it doesn't fit)
1573                  *
1574                  * We just use the end of the buffer to store the type name, and
1575                  * only connect it up if that's necessary.
1576                  */
```

```
1578                char *type_end = type + strlen(type);
1579                char *basetype;
1580                size_t sz;

1582                (void) strlcat(type, " <<", sizeof (type));

1584                basetype = type + strlen(type);
1585                sz = sizeof (type) - (basetype - type);

1587                *type_end = '\0'; /* restore the end of type for strcmp() */

1589                if (mdb_ctf_type_name(base, basetype, sz) != NULL &&
1590                    strcmp(basetype, type) != 0 &&
1591                    strcmp(basetype, "struct ") != 0 &&
1592                    strcmp(basetype, "enum ") != 0 &&
1593                    strcmp(basetype, "union ") != 0) {
1594                        type_end[0] = ' ';      /* reconnect */
1595                        if (strlcat(type, ">>", sizeof (type)) >= sizeof (type))
1596                                (void) strlcpy(
1597                                    type + sizeof (type) - 6, "...>>", 6);
1598                }
1599        }

1601        if (pap->pa_flags & PA_SHOWHOLES) {
1602                ctf_encoding_t e;
1603                ssize_t nsize;
1604                ulong_t newoff;
1605                holeinfo_t *hole;
1606                int extra = IS_COMPOSITE(kind)? 1 : 0;

1608                /*
1609                 * grow the hole array, if necessary
1610                 */
1611                if (pap->pa_nest + depth + extra >= pap->pa_nholes) {
1612                        int new = MAX(MAX(8, pap->pa_nholes * 2),
1613                            pap->pa_nest + depth + extra + 1);

1615                        holeinfo_t *nhi = mdb_zalloc(
1616                            sizeof (*nhi) * new, UM_NOSLEEP | UM_GC);

1618                        bcopy(pap->pa_holes, nhi,
1619                            pap->pa_nholes * sizeof (*nhi));

1621                        pap->pa_holes = nhi;
1622                        pap->pa_nholes = new;
1623                }

1625                hole = &pap->pa_holes[depth + pap->pa_nest];

1627                if (depth != 0 && off > hole->hi_offset)
1628                        print_hole(pap, depth, hole->hi_offset, off);

1630                /* compute the next expected offset */
1631                if (kind == CTF_K_INTEGER &&
1632                    mdb_ctf_type_encoding(base, &e) == 0)
1633                        newoff = off + e.cte_bits;
1634                else if ((nsize = mdb_ctf_type_size(base)) >= 0)
1635                        newoff = off + nsize * NBBY;
1636                else {
1637                        /* something bad happened, disable hole checking */
1638                        newoff = -1UL;          /* ULONG_MAX */
1639                }

1641                hole->hi_offset = newoff;
```



```
1643                if (IS_COMPOSITE(kind)) {
1644                        hole->hi_isunion = (kind == CTF_K_UNION);
1645                        hole++;
1646                        hole->hi_offset = off;
1647                }
1648        }

1650        if (pap->pa_flags & (PA_SHOWTYPE | PA_SHOWNAME | PA_SHOWADDR))
1651                mdb_printf("%*s", (depth + pap->pa_nest) * pap->pa_tab, "");

1653        if (pap->pa_flags & PA_SHOWADDR) {
1654                if (off % NBBY == 0)
1655                        mdb_printf("%llx ", pap->pa_addr + off / NBBY);
1656                else
1657                        mdb_printf("%llx.%lx ",
1658                            pap->pa_addr + off / NBBY, off % NBBY);
1659        }

1661        if ((pap->pa_flags & PA_SHOWTYPE)) {
1662                mdb_printf("%s", type);
1663                /*
1664                 * We want to avoid printing a trailing space when
1665                 * dealing with pointers in a structure, so we end
1666                 * up with:
1667                 *
1668                 *      label_t *t_onfault = 0
1669                 *
1670                 * If depth is zero, always print the trailing space unless
1671                 * we also have a prefix.
1672                 */
1673                if (type[strlen(type) - 1] != '*' ||
1674                    (depth == 0 && (!(pap->pa_flags & PA_SHOWNAME) ||
1675                    pap->pa_prefix == NULL)))
1676                        mdb_printf(" ");
1677        }

1679        if (pap->pa_flags & PA_SHOWNAME) {
1680                if (pap->pa_prefix != NULL && depth <= 1)
1681                        mdb_printf("%s%s", pap->pa_prefix,
1682                            (depth == 0) ? "" : pap->pa_suffix);
1683                mdb_printf("%s", name);
1684        }

1686        if ((pap->pa_flags & PA_SHOWTYPE) && kind == CTF_K_INTEGER) {
1687                ctf_encoding_t e;

1689                if (mdb_ctf_type_encoding(base, &e) == 0) {
1690                        ulong_t bits = e.cte_bits;
1691                        ulong_t size = bits / NBBY;

1693                        if (bits % NBBY != 0 ||
1694                            off % NBBY != 0 ||
1695                            size > 8 ||
1696                            size != mdb_ctf_type_size(base))
1697                                mdb_printf(" :%d", bits);
1698                }
1699        }

1701        if (depth != 0 ||
1702            ((pap->pa_flags & PA_SHOWNAME) && pap->pa_prefix != NULL))
1703                mdb_printf("%s ", pap->pa_flags & PA_SHOWVAL ? " =" : "");

1705        if (depth == 0 && pap->pa_prefix != NULL)
1706                name = pap->pa_prefix;

1708        pap->pa_depth = depth;
```

```
1709         if (kind <= CTF_K_UNKNOWN || kind >= CTF_K_TYPEDEF) {
1710                 mdb_warn("unknown ctf for %s type %s kind %d\n",
1711                     name, type, kind);
1712                 return (-1);
1713         }
1714         rc = printfuncs[kind - 1](type, name, id, base, off, pap);

1716         if (rc != 0)
1717                 mdb_iob_discard(mdb.m_out);
1718         else
1719                 mdb_iob_puts(mdb.m_out, pap->pa_delim);

1721         return (rc);
1722 }

1724 /*
1725  * Special semantics for pipelines.
1726  */
1727 static int
1728 pipe_print(mdb_ctf_id_t id, ulong_t off, void *data)
1729 {
1730         printarg_t *pap = data;
1731         ssize_t size;
1732         static const char *const fsp[] = { "%#r", "%#r", "%#r", "%#llr" };
1733         uintptr_t value;
1734         uintptr_t addr = pap->pa_addr + off / NBBY;
1735         mdb_ctf_id_t base;
1736         int enum_value;
1737         ctf_encoding_t e;

1739         union {
1740                 uint64_t i8;
1741                 uint32_t i4;
1742                 uint16_t i2;
1743                 uint8_t i1;
1744         } u;

1746         if (mdb_ctf_type_resolve(id, &base) == -1) {
1747                 mdb_warn("could not resolve type");
1748                 return (-1);
1749         }

1751         /*
1752          * If the user gives -a, then always print out the address of the
1753          * member.
1754          */
1755         if ((pap->pa_flags & PA_SHOWADDR)) {
1756                 mdb_printf("%#lr\n", addr);
1757                 return (0);
1758         }

1760 again:
1761         switch (mdb_ctf_type_kind(base)) {
1762         case CTF_K_POINTER:
1763                 if (mdb_tgt_aread(pap->pa_tgt, pap->pa_as,
1764                     &value, sizeof (value), addr) != sizeof (value)) {
1765                         mdb_warn("failed to read pointer at %p", addr);
1766                         return (-1);
1767                 }
1768                 mdb_printf("%#lr\n", value);
1769                 break;

1771         case CTF_K_ENUM:
1772                 if (mdb_tgt_aread(pap->pa_tgt, pap->pa_as, &enum_value,
1773                     sizeof (enum_value), addr) != sizeof (enum_value)) {
1774                         mdb_warn("failed to read enum at %llx", addr);
```



```
1775                         return (-1);
1776                 }
1777                 mdb_printf("%#r\n", enum_value);
1778                 break;

1780         case CTF_K_INTEGER:
1781                 if (mdb_ctf_type_encoding(base, &e) != 0) {
1782                         mdb_warn("could not get type encoding\n");
1783                         return (-1);
1784                 }

1786                 /*
1787                  * For immediate values, we just print out the value.
1788                  */
1789                 size = e.cte_bits / NBBY;
1790                 if (size > 8 || (e.cte_bits % NBBY) != 0 ||
1791                     (size & (size - 1)) != 0) {
1792                         return (print_bitfield(off, pap, &e));
1793                 }

1795                 if (mdb_tgt_aread(pap->pa_tgt, pap->pa_as, &u.i8, size,
1796                     addr) != size) {
1797                         mdb_warn("failed to read %lu bytes at %p",
1798                             (ulong_t)size, pap->pa_addr);
1799                         return (-1);
1800                 }

1802                 switch (size) {
1803                 case sizeof (uint8_t):
1804                         mdb_printf(fsp[0], u.i1);
1805                         break;
1806                 case sizeof (uint16_t):
1807                         mdb_printf(fsp[1], u.i2);
1808                         break;
1809                 case sizeof (uint32_t):
1810                         mdb_printf(fsp[2], u.i4);
1811                         break;
1812                 case sizeof (uint64_t):
1813                         mdb_printf(fsp[3], u.i8);
1814                         break;
1815                 }
1816                 mdb_printf("\n");
1817                 break;

1819         case CTF_K_FUNCTION:
1820         case CTF_K_FLOAT:
1821         case CTF_K_ARRAY:
1822         case CTF_K_UNKNOWN:
1823         case CTF_K_STRUCT:
1824         case CTF_K_UNION:
1825         case CTF_K_FORWARD:
1826                 /*
1827                  * For these types, always print the address of the member
1828                  */
1829                 mdb_printf("%#lr\n", addr);
1830                 break;

1832         default:
1833                 mdb_warn("unknown type %d", mdb_ctf_type_kind(base));
1834                 break;
1835         }

1837         return (0);
1838 }

1840 static int
```

```
1841 parse_delimiter(char **strp)
1842 {
1843         switch (**strp) {
1844         case '\0':
1845                 return (MEMBER_DELIM_DONE);

1847         case '.':
1848                 *strp = *strp + 1;
1849                 return (MEMBER_DELIM_DOT);

1851         case '[':
1852                 *strp = *strp + 1;
1853                 return (MEMBER_DELIM_LBR);

1855         case '-':
1856                 *strp = *strp + 1;
1857                 if (**strp == '>') {
1858                         *strp = *strp + 1;
1859                         return (MEMBER_DELIM_PTR);
1860                 }
1861                 *strp = *strp - 1;
1862                 /*FALLTHROUGH*/
1863         default:
1864                 return (MEMBER_DELIM_ERR);
1865         }
1866 }

1868 static int
1869 deref(printarg_t *pap, size_t size)
1870 {
1871         uint32_t a32;
1872         mdb_tgt_as_t as = pap->pa_as;
1873         mdb_tgt_addr_t *ap = &pap->pa_addr;

1875         if (size == sizeof (mdb_tgt_addr_t)) {
1876                 if (mdb_tgt_aread(mdb.m_target, as, ap, size, *ap) == -1) {
1877                         mdb_warn("could not dereference pointer %llx\n", *ap);
1878                         return (-1);
1879                 }
1880         } else {
1881                 if (mdb_tgt_aread(mdb.m_target, as, &a32, size, *ap) == -1) {
1882                         mdb_warn("could not dereference pointer %x\n", *ap);
1883                         return (-1);
1884                 }

1886                 *ap = (mdb_tgt_addr_t)a32;
1887         }

1889         /*
1890          * We've dereferenced at least once, we must be on the real
1891          * target. If we were in the immediate target, reset to the real
1892          * target; it's reset as needed when we return to the print
1893          * routines.
1894          */
1895         if (pap->pa_tgt == pap->pa_immtgt)
1896                 pap->pa_tgt = pap->pa_realtgt;

1898         return (0);
1899 }

1901 static int
1902 parse_member(printarg_t *pap, const char *str, mdb_ctf_id_t id,
1903     mdb_ctf_id_t *idp, ulong_t *offp, int *last_deref)
1904 {
1905         int delim;
1906         char member[64];
```



```
1907         char buf[128];
1908         uint_t index;
1909         char *start = (char *)str;
1910         char *end;
1911         ulong_t off = 0;
1912         mdb_ctf_arinfo_t ar;
1913         mdb_ctf_id_t rid;
1914         int kind;
1915         ssize_t size;
1916         int non_array = FALSE;

1918         /*
1919          * id always has the unresolved type for printing error messages
1920          * that include the type; rid always has the resolved type for
1921          * use in mdb_ctf_* calls.  It is possible for this command to fail,
1922          * however, if the resolved type is in the parent and it is currently
1923          * unavailable.  Note that we also can't print out the name of the
1924          * type, since that would also rely on looking up the resolved name.
1925          */
1926         if (mdb_ctf_type_resolve(id, &rid) != 0) {
1927                 mdb_warn("failed to resolve type");
1928                 return (-1);
1929         }

1931         delim = parse_delimiter(&start);
1932         /*
1933          * If the user fails to specify an initial delimiter, guess -> for
1934          * pointer types and . for non-pointer types.
1935          */
1936         if (delim == MEMBER_DELIM_ERR)
1937                 delim = (mdb_ctf_type_kind(rid) == CTF_K_POINTER) ?
1938                     MEMBER_DELIM_PTR : MEMBER_DELIM_DOT;

1940         *last_deref = FALSE;

1942         while (delim != MEMBER_DELIM_DONE) {
1943                 switch (delim) {
1944                 case MEMBER_DELIM_PTR:
1945                         kind = mdb_ctf_type_kind(rid);
1946                         if (kind != CTF_K_POINTER) {
1947                                 mdb_warn("%s is not a pointer type\n",
1948                                     mdb_ctf_type_name(id, buf, sizeof (buf)));
1949                                 return (-1);
1950                         }

1952                         size = mdb_ctf_type_size(id);
1953                         if (deref(pap, size) != 0)
1954                                 return (-1);

1956                         (void) mdb_ctf_type_reference(rid, &id);
1957                         (void) mdb_ctf_type_resolve(id, &rid);

1959                         off = 0;
1960                         break;

1962                 case MEMBER_DELIM_DOT:
1963                         kind = mdb_ctf_type_kind(rid);
1964                         if (kind != CTF_K_STRUCT && kind != CTF_K_UNION) {
1965                                 mdb_warn("%s is not a struct or union type\n",
1966                                     mdb_ctf_type_name(id, buf, sizeof (buf)));
1967                                 return (-1);
1968                         }
1969                         break;

1971                 case MEMBER_DELIM_LBR:
1972                         end = strchr(start, ']');
```

```
1973                        if (end == NULL) {
1974                                mdb_warn("no trailing ']'\n");
1975                                return (-1);
1976                        }
1977
1978                        (void) mdb_snprintf(member, end - start + 1, "%s",
1979                            start);
1980
1981                        index = mdb_strtoull(member);
1982
1983                        switch (mdb_ctf_type_kind(rid)) {
1984                        case CTF_K_POINTER:
1985                                size = mdb_ctf_type_size(rid);
1986
1987                                if (deref(pap, size) != 0)
1988                                        return (-1);
1989
1990                                (void) mdb_ctf_type_reference(rid, &id);
1991                                (void) mdb_ctf_type_resolve(id, &rid);
1992
1993                                size = mdb_ctf_type_size(id);
1994                                if (size <= 0) {
1995                                        mdb_warn("cannot dereference void "
1996                                            "type\n");
1997                                        return (-1);
1998                                }
1999
2000                                pap->pa_addr += index * size;
2001                                off = 0;
2002
2003                                if (index == 0 && non_array)
2004                                        *last_deref = TRUE;
2005                                break;
2006
2007                        case CTF_K_ARRAY:
2008                                (void) mdb_ctf_array_info(rid, &ar);
2009
2010                                if (index >= ar.mta_nelems) {
2011                                        mdb_warn("index %r is outside of "
2012                                            "array bounds [0 .. %r]\n",
2013                                            index, ar.mta_nelems - 1);
2014                                }
2015
2016                                id = ar.mta_contents;
2017                                (void) mdb_ctf_type_resolve(id, &rid);
2018
2019                                size = mdb_ctf_type_size(id);
2020                                if (size <= 0) {
2021                                        mdb_warn("cannot dereference void "
2022                                            "type\n");
2023                                        return (-1);
2024                                }
2025
2026                                pap->pa_addr += index * size;
2027                                off = 0;
2028                                break;
2029
2030                        default:
2031                                mdb_warn("cannot index into non-array, "
2032                                    "non-pointer type\n");
2033                                return (-1);
2034                        }
2035
2036                        start = end + 1;
2037                        delim = parse_delimiter(&start);
2038                        continue;
```



```
2040                case MEMBER_DELIM_ERR:
2041                default:
2042                        mdb_warn("'%c' is not a valid delimiter\n", *start);
2043                        return (-1);
2044                }
2045
2046                *last_deref = FALSE;
2047                non_array = TRUE;
2048
2049                /*
2050                 * Find the end of the member name; assume that a member
2051                 * name is at least one character long.
2052                 */
2053                for (end = start + 1; isalnum(*end) || *end == '_'; end++)
2054                        continue;
2055
2056                (void) mdb_snprintf(member, end - start + 1, "%s", start);
2057
2058                if (mdb_ctf_member_info(rid, member, &off, &id) != 0) {
2059                        mdb_warn("failed to find member %s of %s", member,
2060                            mdb_ctf_type_name(id, buf, sizeof (buf)));
2061                        return (-1);
2062                }
2063                (void) mdb_ctf_type_resolve(id, &rid);
2064
2065                pap->pa_addr += off / NBBY;
2066
2067                start = end;
2068                delim = parse_delimiter(&start);
2069        }
2070
2071        *idp = id;
2072        *offp = off;
2073
2074        return (0);
2075 }
2076
2077 static int
2078 cmd_print_tab_common(mdb_tab_cookie_t *mcp, uint_t flags, int argc,
2079     const mdb_arg_t *argv)
2080 {
2081        char tn[MDB_SYM_NAMLEN];
2082        char member[64];
2083        int delim, kind;
2084        int ret = 0;
2085        mdb_ctf_id_t id, rid;
2086        mdb_ctf_arinfo_t ar;
2087        char *start, *end;
2088        ulong_t dul;
2089
2090        if (argc == 0 && !(flags & DCMD_TAB_SPACE))
2091                return (0);
2092
2093        if (argc == 0 && (flags & DCMD_TAB_SPACE))
2094                return (mdb_tab_complete_type(mcp, NULL, MDB_TABC_NOPOINT |
2095                    MDB_TABC_NOARRAY));
2096
2097        if ((ret = mdb_tab_typename(&argc, &argv, tn, sizeof (tn))) < 0)
2098                return (ret);
2099
2100        if (argc == 1 && (!(flags & DCMD_TAB_SPACE) || ret == 1))
2101                return (mdb_tab_complete_type(mcp, tn, MDB_TABC_NOPOINT |
2102                    MDB_TABC_NOARRAY));
2103
2104        if (argc == 1 && (flags & DCMD_TAB_SPACE))
```

```
2105                    return (mdb_tab_complete_member(mcp, tn, NULL));
2106
2107            /*
2108             * This is the reason that tab completion was created. We're going to go
2109             * along and walk the delimiters until we find something a member that
2110             * we don't recognize, at which point we'll try and tab complete it.
2111             * Note that ::print takes multiple args, so this is going to operate on
2112             * whatever the last arg that we have is.
2113             */
2114            if (mdb_ctf_lookup_by_name(tn, &id) != 0)
2115                    return (1);
2116
2117            (void) mdb_ctf_type_resolve(id, &rid);
2118            start = (char *)argv[argc-1].a_un.a_str;
2119            delim = parse_delimiter(&start);
2120
2121            /*
2122             * If we hit the case where we actually have no delimiters, than we need
2123             * to make sure that we properly set up the fields the loops would.
2124             */
2125            if (delim == MEMBER_DELIM_DONE)
2126                    (void) mdb_snprintf(member, sizeof (member), "%s", start);
2127
2128            while (delim != MEMBER_DELIM_DONE) {
2129                    switch (delim) {
2130                    case MEMBER_DELIM_PTR:
2131                            kind = mdb_ctf_type_kind(rid);
2132                            if (kind != CTF_K_POINTER)
2133                                    return (1);
2134
2135                            (void) mdb_ctf_type_reference(rid, &id);
2136                            (void) mdb_ctf_type_resolve(id, &rid);
2137                            break;
2138                    case MEMBER_DELIM_DOT:
2139                            kind = mdb_ctf_type_kind(rid);
2140                            if (kind != CTF_K_STRUCT && kind != CTF_K_UNION)
2141                                    return (1);
2142                            break;
2143                    case MEMBER_DELIM_LBR:
2144                            end = strchr(start, ']');
2145                            /*
2146                             * We're not going to try and tab complete the indexes
2147                             * here. So for now, punt on it. Also, we're not going
2148                             * to try and validate you're within the bounds, just
2149                             * that you get the type you asked for.
2150                             */
2151                            if (end == NULL)
2152                                    return (1);
2153
2154                            switch (mdb_ctf_type_kind(rid)) {
2155                            case CTF_K_POINTER:
2156                                    (void) mdb_ctf_type_reference(rid, &id);
2157                                    (void) mdb_ctf_type_resolve(id, &rid);
2158                                    break;
2159                            case CTF_K_ARRAY:
2160                                    (void) mdb_ctf_array_info(rid, &ar);
2161                                    id = ar.mta_contents;
2162                                    (void) mdb_ctf_type_resolve(id, &rid);
2163                                    break;
2164                            default:
2165                                    return (1);
2166                            }
2167
2168                            start = end + 1;
2169                            delim = parse_delimiter(&start);
2170                            break;
```



```
2171                    case MEMBER_DELIM_ERR:
2172                    default:
2173                            break;
2174                    }
2175
2176                    for (end = start + 1; isalnum(*end) || *end == '_'; end++)
2177                            continue;
2178
2179                    (void) mdb_snprintf(member, end - start + 1, start);
2180
2181                    /*
2182                     * We are going to try to resolve this name as a member. There
2183                     * are a few two different questions that we need to answer. The
2184                     * first is do we recognize this member. The second is are we at
2185                     * the end of the string. If we encounter a member that we don't
2186                     * recognize before the end, then we have to error out and can't
2187                     * complete it. But if there are no more delimiters then we can
2188                     * try and complete it.
2189                     */
2190                    ret = mdb_ctf_member_info(rid, member, &dul, &id);
2191                    start = end;
2192                    delim = parse_delimiter(&start);
2193                    if (ret != 0 && errno == EMDB_CTFNOMEMB) {
2194                            if (delim != MEMBER_DELIM_DONE)
2195                                    return (1);
2196                            continue;
2197                    } else if (ret != 0)
2198                            return (1);
2199
2200                    if (delim == MEMBER_DELIM_DONE)
2201                            return (mdb_tab_complete_member_by_id(mcp, rid,
2202                                member));
2203
2204                    (void) mdb_ctf_type_resolve(id, &rid);
2205            }
2206
2207            /*
2208             * If we've reached here, then we need to try and tab complete the last
2209             * field, which is currently member, based on the ctf type id that we
2210             * already have in rid.
2211             */
2212            return (mdb_tab_complete_member_by_id(mcp, rid, member));
2213    }
2214
2215    int
2216    cmd_print_tab(mdb_tab_cookie_t *mcp, uint_t flags, int argc,
2217        const mdb_arg_t *argv)
2218    {
2219            int i, dummy;
2220
2221            /*
2222             * This getopts is only here to make the tab completion work better when
2223             * including options in the ::print arguments. None of the values should
2224             * be used. This should only be updated with additional arguments, if
2225             * they are added to cmd_print.
2226             */
2227            i = mdb_getopts(argc, argv,
2228                'a', MDB_OPT_SETBITS, PA_SHOWADDR, &dummy,
2229                'C', MDB_OPT_SETBITS, TRUE, &dummy,
2230                'c', MDB_OPT_UINTPTR, &dummy,
2231                'd', MDB_OPT_SETBITS, PA_INTDEC, &dummy,
2232                'h', MDB_OPT_SETBITS, PA_SHOWHOLES, &dummy,
2233                'i', MDB_OPT_SETBITS, TRUE, &dummy,
2234                'L', MDB_OPT_SETBITS, TRUE, &dummy,
2235                'l', MDB_OPT_UINTPTR, &dummy,
2236                'n', MDB_OPT_SETBITS, PA_NOSYMBOLIC, &dummy,
```

```
2237             'p', MDB_OPT_SETBITS, TRUE, &dummy,
2238             's', MDB_OPT_UINTPTR, &dummy,
2239             'T', MDB_OPT_SETBITS, PA_SHOWTYPE | PA_SHOWBASETYPE, &dummy,
2240             't', MDB_OPT_SETBITS, PA_SHOWTYPE, &dummy,
2241             'x', MDB_OPT_SETBITS, PA_INTHEX, &dummy,
2242             NULL);

2244         argc -= i;
2245         argv += i;

2247         return (cmd_print_tab_common(mcp, flags, argc, argv));
2248 }

2250 /*
2251  * Recursively descend a print a given data structure.  We create a struct of
2252  * the relevant print arguments and then call mdb_ctf_type_visit() to do the
2253  * traversal, using elt_print() as the callback for each element.
2254  */
2255 /*ARGSUSED*/
2256 int
2257 cmd_print(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
2258 {
2259         uintptr_t opt_c = MDB_ARR_NOLIMIT, opt_l = MDB_ARR_NOLIMIT;
2260         uint_t opt_C = FALSE, opt_L = FALSE, opt_p = FALSE, opt_i = FALSE;
2261         uintptr_t opt_s = (uintptr_t)-1ul;
2262         int uflags = (flags & DCMD_ADDRSPEC) ? PA_SHOWVAL : 0;
2263         mdb_ctf_id_t id;
2264         int err = DCMD_OK;

2266         mdb_tgt_t *t = mdb.m_target;
2267         printarg_t pa;
2268         int d, i;

2270         char s_name[MDB_SYM_NAMLEN];
2271         mdb_syminfo_t s_info;
2272         GElf_Sym sym;

2274         /*
2275          * If a new option is added, make sure the getopts above in
2276          * cmd_print_tab is also updated.
2277          */
2278         i = mdb_getopts(argc, argv,
2279             'a', MDB_OPT_SETBITS, PA_SHOWADDR, &uflags,
2280             'C', MDB_OPT_SETBITS, TRUE, &opt_C,
2281             'c', MDB_OPT_UINTPTR, &opt_c,
2282             'd', MDB_OPT_SETBITS, PA_INTDEC, &uflags,
2283             'h', MDB_OPT_SETBITS, PA_SHOWHOLES, &uflags,
2284             'i', MDB_OPT_SETBITS, TRUE, &opt_i,
2285             'L', MDB_OPT_SETBITS, TRUE, &opt_L,
2286             'l', MDB_OPT_UINTPTR, &opt_l,
2287             'n', MDB_OPT_SETBITS, PA_NOSYMBOLIC, &uflags,
2288             'p', MDB_OPT_SETBITS, TRUE, &opt_p,
2289             's', MDB_OPT_UINTPTR, &opt_s,
2290             'T', MDB_OPT_SETBITS, PA_SHOWTYPE | PA_SHOWBASETYPE, &uflags,
2291             't', MDB_OPT_SETBITS, PA_SHOWTYPE, &uflags,
2292             'x', MDB_OPT_SETBITS, PA_INTHEX, &uflags,
2293             NULL);

2295         if (uflags & PA_INTHEX)
2296                 uflags &= ~PA_INTDEC;   /* -x and -d are mutually exclusive */

2298         uflags |= PA_SHOWNAME;

2300         if (opt_p && opt_i) {
2301                 mdb_warn("-p and -i options are incompatible\n");
2302                 return (DCMD_ERR);
```



```
2303         }

2305         argc -= i;
2306         argv += i;

2308         if (argc != 0 && argv->a_type == MDB_TYPE_STRING) {
2309                 const char *t_name = s_name;
2310                 int ret;

2312                 if (strchr("+-", argv->a_un.a_str[0]) != NULL)
2313                         return (DCMD_USAGE);

2315                 if ((ret = args_to_typename(&argc, &argv, s_name,
2316                     sizeof (s_name))) != 0)
2317                         return (ret);

2319                 if (mdb_ctf_lookup_by_name(t_name, &id) != 0) {
2320                         if (!(flags & DCMD_ADDRSPEC) || opt_i ||
2321                             addr_to_sym(t, addr, s_name, sizeof (s_name),
2322                             &sym, &s_info) == NULL ||
2323                             mdb_ctf_lookup_by_symbol(&sym, &s_info, &id) != 0) {

2325                                 mdb_warn("failed to look up type %s", t_name);
2326                                 return (DCMD_ABORT);
2327                         }
2328                 } else {
2329                         argc--;
2330                         argv++;
2331                 }

2333         } else if (!(flags & DCMD_ADDRSPEC) || opt_i) {
2334                 return (DCMD_USAGE);

2336         } else if (addr_to_sym(t, addr, s_name, sizeof (s_name),
2337             &sym, &s_info) == NULL) {
2338                 mdb_warn("no symbol information for %a", addr);
2339                 return (DCMD_ERR);

2341         } else if (mdb_ctf_lookup_by_symbol(&sym, &s_info, &id) != 0) {
2342                 mdb_warn("no type data available for %a [%u]", addr,
2343                     s_info.sym_id);
2344                 return (DCMD_ERR);
2345         }

2347         pa.pa_tgt = mdb.m_target;
2348         pa.pa_realtgt = pa.pa_tgt;
2349         pa.pa_immtgt = NULL;
2350         pa.pa_as = opt_p ? MDB_TGT_AS_PHYS : MDB_TGT_AS_VIRT;
2351         pa.pa_armemlim = mdb.m_armemlim;
2352         pa.pa_arstrlim = mdb.m_arstrlim;
2353         pa.pa_delim = "\n";
2354         pa.pa_flags = uflags;
2355         pa.pa_nest = 0;
2356         pa.pa_tab = 4;
2357         pa.pa_prefix = NULL;
2358         pa.pa_suffix = NULL;
2359         pa.pa_holes = NULL;
2360         pa.pa_nholes = 0;
2361         pa.pa_depth = 0;
2362         pa.pa_maxdepth = opt_s;
2363         pa.pa_nooutdepth = (uint_t)-1;

2365         if ((flags & DCMD_ADDRSPEC) && !opt_i)
2366                 pa.pa_addr = opt_p ? mdb_get_dot() : addr;
2367         else
2368                 pa.pa_addr = NULL;
```

```
2370         if (opt_i) {
2371                 const char *vargv[2];
2372                 uintmax_t dot = mdb_get_dot();
2373                 size_t outsize = mdb_ctf_type_size(id);
2374                 vargv[0] = (const char *)&dot;
2375                 vargv[1] = (const char *)&outsize;
2376                 pa.pa_immtgt = mdb_tgt_create(mdb_value_tgt_create,
2377                     0, 2, vargv);
2378                 pa.pa_tgt = pa.pa_immtgt;
2379         }

2381         if (opt_c != MDB_ARR_NOLIMIT)
2382                 pa.pa_arstrlim = opt_c;
2383         if (opt_C)
2384                 pa.pa_arstrlim = MDB_ARR_NOLIMIT;
2385         if (opt_l != MDB_ARR_NOLIMIT)
2386                 pa.pa_armemlim = opt_l;
2387         if (opt_L)
2388                 pa.pa_armemlim = MDB_ARR_NOLIMIT;

2390         if (argc > 0) {
2391                 for (i = 0; i < argc; i++) {
2392                         mdb_ctf_id_t mid;
2393                         int last_deref;
2394                         ulong_t off;
2395                         int kind;
2396                         char buf[MDB_SYM_NAMLEN];

2398                         mdb_tgt_t *oldtgt = pa.pa_tgt;
2399                         mdb_tgt_as_t oldas = pa.pa_as;
2400                         mdb_tgt_addr_t oldaddr = pa.pa_addr;

2402                         if (argv->a_type == MDB_TYPE_STRING) {
2403                                 const char *member = argv[i].a_un.a_str;
2404                                 mdb_ctf_id_t rid;

2406                                 if (parse_member(&pa, member, id, &mid,
2407                                     &off, &last_deref) != 0) {
2408                                         err = DCMD_ABORT;
2409                                         goto out;
2410                                 }

2412                                 /*
2413                                  * If the member string ends with a "[0]"
2414                                  * (last_deref * is true) and the type is a
2415                                  * structure or union, * print "->" rather
2416                                  * than "[0]." in elt_print.
2417                                  */
2418                                 (void) mdb_ctf_type_resolve(mid, &rid);
2419                                 kind = mdb_ctf_type_kind(rid);
2420                                 if (last_deref && IS_SOU(kind)) {
2421                                         char *end;
2422                                         (void) mdb_snprintf(buf, sizeof (buf),
2423                                             "%s", member);
2424                                         end = strrchr(buf, '[');
2425                                         *end = '\0';
2426                                         pa.pa_suffix = "->";
2427                                         member = &buf[0];
2428                                 } else if (IS_SOU(kind)) {
2429                                         pa.pa_suffix = ".";
2430                                 } else {
2431                                         pa.pa_suffix = "";
2432                                 }

2434                                 pa.pa_prefix = member;
```



```
2435                         } else {
2436                                 ulong_t moff;

2438                                 moff = (ulong_t)argv[i].a_un.a_val;

2440                                 if (mdb_ctf_offset_to_name(id, moff * NBBY,
2441                                     buf, sizeof (buf), 0, &mid, &off) == -1) {
2442                                         mdb_warn("invalid offset %lx\n", moff);
2443                                         err = DCMD_ABORT;
2444                                         goto out;
2445                                 }

2447                                 pa.pa_prefix = buf;
2448                                 pa.pa_addr += moff - off / NBBY;
2449                                 pa.pa_suffix = strlen(buf) == 0 ? "" : ".";
2450                         }

2452                         off %= NBBY;
2453                         if (flags & DCMD_PIPE_OUT) {
2454                                 if (pipe_print(mid, off, &pa) != 0) {
2455                                         mdb_warn("failed to print type");
2456                                         err = DCMD_ERR;
2457                                         goto out;
2458                                 }
2459                         } else if (off != 0) {
2460                                 mdb_ctf_id_t base;
2461                                 (void) mdb_ctf_type_resolve(mid, &base);

2463                                 if (elt_print("", mid, base, off, 0,
2464                                     &pa) != 0) {
2465                                         mdb_warn("failed to print type");
2466                                         err = DCMD_ERR;
2467                                         goto out;
2468                                 }
2469                         } else {
2470                                 if (mdb_ctf_type_visit(mid, elt_print,
2471                                     &pa) == -1) {
2472                                         mdb_warn("failed to print type");
2473                                         err = DCMD_ERR;
2474                                         goto out;
2475                                 }

2477                                 for (d = pa.pa_depth - 1; d >= 0; d--)
2478                                         print_close_sou(&pa, d);
2479                         }

2481                         pa.pa_depth = 0;
2482                         pa.pa_tgt = oldtgt;
2483                         pa.pa_as = oldas;
2484                         pa.pa_addr = oldaddr;
2485                         pa.pa_delim = "\n";
2486                 }

2488         } else if (flags & DCMD_PIPE_OUT) {
2489                 if (pipe_print(id, 0, &pa) != 0) {
2490                         mdb_warn("failed to print type");
2491                         err = DCMD_ERR;
2492                         goto out;
2493                 }
2494         } else {
2495                 if (mdb_ctf_type_visit(id, elt_print, &pa) == -1) {
2496                         mdb_warn("failed to print type");
2497                         err = DCMD_ERR;
2498                         goto out;
2499                 }
```

```
2501                 for (d = pa.pa_depth - 1; d >= 0; d--)
2502                         print_close_sou(&pa, d);
2503         }

2505         mdb_set_dot(addr + mdb_ctf_type_size(id));
2506         err = DCMD_OK;
2507 out:
2508         if (pa.pa_immtgt)
2509                 mdb_tgt_destroy(pa.pa_immtgt);
2510         return (err);
2511 }

2513 void
2514 print_help(void)
2515 {
2516         mdb_printf(
2517             "-a         show address of object\n"
2518             "-C         unlimit the length of character arrays\n"
2519             "-c limit   limit the length of character arrays\n"
2520             "-d         output values in decimal\n"
2521             "-h         print holes in structures\n"
2522             "-i         interpret address as data of the given type\n"
2523             "-L         unlimit the length of standard arrays\n"
2524             "-l limit   limit the length of standard arrays\n"
2525             "-n         don't print pointers as symbol offsets\n"
2526             "-p         interpret address as a physical memory address\n"
2527             "-s depth   limit the recursion depth\n"
2528             "-T         show type and <<base type>> of object\n"
2529             "-t         show type of object\n"
2530             "-x         output values in hexadecimal\n"
2531             "\n"
2532             "type may be omitted if the C type of addr can be inferred.\n"
2533             "\n"
2534             "Members may be specified with standard C syntax using the\n"
2535             "array indexing operator \"[index]\", structure member\n"
2536             "operator \".\", or structure pointer operator \"->\".\n"
2537             "\n"
2538             "Offsets must use the $[ expression ] syntax\n");
2539 }

2541 static int
2542 printf_signed(mdb_ctf_id_t id, uintptr_t addr, ulong_t off, char *fmt,
2543     boolean_t sign)
2544 {
2545         ssize_t size;
2546         mdb_ctf_id_t base;
2547         ctf_encoding_t e;

2549         union {
2550                 uint64_t ui8;
2551                 uint32_t ui4;
2552                 uint16_t ui2;
2553                 uint8_t ui1;
2554                 int64_t i8;
2555                 int32_t i4;
2556                 int16_t i2;
2557                 int8_t i1;
2558         } u;

2560         if (mdb_ctf_type_resolve(id, &base) == -1) {
2561                 mdb_warn("could not resolve type");
2562                 return (DCMD_ABORT);
2563         }

2565         switch (mdb_ctf_type_kind(base)) {
2566                 case CTF_K_ENUM:
```



```
2567                         e.cte_format = CTF_INT_SIGNED;
2568                         e.cte_offset = 0;
2569                         e.cte_bits = mdb_ctf_type_size(id) * NBBY;
2570                         break;
2571                 case CTF_K_INTEGER:
2572                         if (mdb_ctf_type_encoding(base, &e) != 0) {
2573                                 mdb_warn("could not get type encoding");
  29         if (mdb_ctf_type_kind(base) != CTF_K_INTEGER) {
  30                 mdb_warn("expected integer type\n");
2574                                 return (DCMD_ABORT);
2575                         }
2576                         break;
2577                 default:
2578                         mdb_warn("expected integer type\n");

  34         if (mdb_ctf_type_encoding(base, &e) != 0) {
  35                 mdb_warn("could not get type encoding");
2579                         return (DCMD_ABORT);
2580         }

2582         if (sign)
2583                 sign = e.cte_format & CTF_INT_SIGNED;

2585         size = e.cte_bits / NBBY;

2587         /*
2588          * Check to see if our life has been complicated by the presence of
2589          * a bitfield.  If it has, we will print it using logic that is only
2590          * slightly different than that found in print_bitfield(), above.  (In
2591          * particular, see the comments there for an explanation of the
2592          * endianness differences in this code.)
2593          */
2594         if (size > 8 || (e.cte_bits % NBBY) != 0 ||
2595             (size & (size - 1)) != 0) {
2596                 uint64_t mask = (1ULL << e.cte_bits) - 1;
2597                 uint64_t value = 0;
2598                 uint8_t *buf = (uint8_t *)&value;
2599                 uint8_t shift;

2601                 /*
2602                  * Round our size up one byte.
2603                  */
2604                 size = (e.cte_bits + (NBBY - 1)) / NBBY;

2606                 if (e.cte_bits > sizeof (value) * NBBY - 1) {
2607                         mdb_printf("invalid bitfield size %u", e.cte_bits);
2608                         return (DCMD_ABORT);
2609                 }

2611 #ifdef _BIG_ENDIAN
2612                 buf += sizeof (value) - size;
2613                 off += e.cte_bits;
2614 #endif

2616                 if (mdb_vread(buf, size, addr) == -1) {
2617                         mdb_warn("failed to read %lu bytes at %p", size, addr);
2618                         return (DCMD_ERR);
2619                 }

2621                 shift = off % NBBY;
2622 #ifdef _BIG_ENDIAN
2623                 shift = NBBY - shift;
2624 #endif

2626                 /*
2627                  * If we have a bit offset within the byte, shift it down.
```

```
2628                  */
2629                 if (off % NBBY != 0)
2630                         value >>= shift;
2631                 value &= mask;

2633                 if (sign) {
2634                         int sshift = sizeof (value) * NBBY - e.cte_bits;
2635                         value = ((int64_t)value << sshift) >> sshift;
2636                 }

2638                 mdb_printf(fmt, value);
2639                 return (0);
2640         }

2642         if (mdb_vread(&u.i8, size, addr) == -1) {
2643                 mdb_warn("failed to read %lu bytes at %p", (ulong_t)size, addr);
2644                 return (DCMD_ERR);
2645         }

2647         switch (size) {
2648         case sizeof (uint8_t):
2649                 mdb_printf(fmt, (uint64_t)(sign ? u.i1 : u.ui1));
2650                 break;
2651         case sizeof (uint16_t):
2652                 mdb_printf(fmt, (uint64_t)(sign ? u.i2 : u.ui2));
2653                 break;
2654         case sizeof (uint32_t):
2655                 mdb_printf(fmt, (uint64_t)(sign ? u.i4 : u.ui4));
2656                 break;
2657         case sizeof (uint64_t):
2658                 mdb_printf(fmt, (uint64_t)(sign ? u.i8 : u.ui8));
2659                 break;
2660         }

2662         return (0);
2663 }
```

*unchanged_portion_omitted_*

```
2734 /*ARGSUSED*/
2735 static int
2736 printf_string(mdb_ctf_id_t id, uintptr_t addr, ulong_t off, char *fmt)
2737 {
2738         mdb_ctf_id_t base;
2739         mdb_ctf_arinfo_t r;
2740         char buf[1024];
2741         ssize_t size;

2743         if (mdb_ctf_type_resolve(id, &base) == -1) {
2744                 mdb_warn("could not resolve type");
2745                 return (DCMD_ABORT);
2746         }

2748         if (mdb_ctf_type_kind(base) == CTF_K_POINTER) {
2749                 uintptr_t value;

2751                 if (mdb_vread(&value, sizeof (value), addr) == -1) {
2752                         mdb_warn("failed to read pointer at %llx", addr);
2753                         return (DCMD_ERR);
2754                 }

2756                 if (mdb_readstr(buf, sizeof (buf) - 1, value) < 0) {
2757                         mdb_warn("failed to read string at %llx", value);
2758                         return (DCMD_ERR);
2759                 }

2761                 mdb_printf(fmt, buf);
```



```
2762                 return (0);
2763         }

2765         if (mdb_ctf_type_kind(base) == CTF_K_ENUM) {
2766                 const char *strval;
2767                 int value;

2769                 if (mdb_vread(&value, sizeof (value), addr) == -1) {
2770                         mdb_warn("failed to read pointer at %llx", addr);
2771                         return (DCMD_ERR);
2772                 }

2774                 if ((strval = mdb_ctf_enum_name(id, value))) {
2775                         mdb_printf(fmt, strval);
2776                 } else {
2777                         (void) mdb_snprintf(buf, sizeof (buf), "<%d>", value);
2778                         mdb_printf(fmt, buf);
2779                 }

2781 #endif /* ! codereview */
2782                 return (0);
2783         }

2785         if (mdb_ctf_type_kind(base) != CTF_K_ARRAY) {
2786                 mdb_warn("exepected pointer or array type\n");
2787                 return (DCMD_ABORT);
2788         }

2790         if (mdb_ctf_array_info(base, &r) == -1 ||
2791             mdb_ctf_type_resolve(r.mta_contents, &base) == -1 ||
2792             (size = mdb_ctf_type_size(base)) == -1) {
2793                 mdb_warn("can't determine array type");
2794                 return (DCMD_ABORT);
2795         }

2797         if (size != 1) {
2798                 mdb_warn("string format specifier requires "
2799                     "an array of characters\n");
2800                 return (DCMD_ABORT);
2801         }

2803         bzero(buf, sizeof (buf));

2805         if (mdb_vread(buf, MIN(r.mta_nelems, sizeof (buf) - 1), addr) == -1) {
2806                 mdb_warn("failed to read array at %p", addr);
2807                 return (DCMD_ERR);
2808         }

2810         mdb_printf(fmt, buf);

2812         return (0);
2813 }

2815 /*ARGSUSED*/
2816 static int
2817 printf_ipv6(mdb_ctf_id_t id, uintptr_t addr, ulong_t off, char *fmt)
2818 {
2819         mdb_ctf_id_t base;
2820         mdb_ctf_id_t ipv6_type, ipv6_base;
2821         in6_addr_t ipv6;

2823         if (mdb_ctf_lookup_by_name("in6_addr_t", &ipv6_type) == -1) {
2824                 mdb_warn("could not resolve in6_addr_t type\n");
2825                 return (DCMD_ABORT);
2826         }
```

```
2828         if (mdb_ctf_type_resolve(id, &base) == -1) {
2829                 mdb_warn("could not resolve type\n");
2830                 return (DCMD_ABORT);
2831         }

2833         if (mdb_ctf_type_resolve(ipv6_type, &ipv6_base) == -1) {
2834                 mdb_warn("could not resolve in6_addr_t type\n");
2835                 return (DCMD_ABORT);
2836         }

2838         if (mdb_ctf_type_cmp(base, ipv6_base) != 0) {
2839                 mdb_warn("requires argument of type in6_addr_t\n");
2840                 return (DCMD_ABORT);
2841         }

2843         if (mdb_vread(&ipv6, sizeof (ipv6), addr) == -1) {
2844                 mdb_warn("couldn't read in6_addr_t at %p", addr);
2845                 return (DCMD_ERR);
2846         }

2848         mdb_printf(fmt, &ipv6);

2850         return (0);
2851 }

2853 /*
2854  * To validate the format string specified to ::printf, we run the format
2855  * string through a very simple state machine that restricts us to a subset
2856  * of mdb_printf() functionality.
2857  */
2858 enum {
2859         PRINTF_NOFMT = 1,               /* no current format specifier */
2860         PRINTF_PERC,                    /* processed '%' */
2861         PRINTF_FMT,                     /* processing format specifier */
2862         PRINTF_LEFT,                    /* processed '-', expecting width */
2863         PRINTF_WIDTH,                   /* processing width */
2864         PRINTF_QUES                     /* processed '?', expecting format */
2865 };

2867 int
2868 cmd_printf_tab(mdb_tab_cookie_t *mcp, uint_t flags, int argc,
2869     const mdb_arg_t *argv)
2870 {
2871         int ii;
2872         char *f;

2874         /*
2875          * If argc doesn't have more than what should be the format string,
2876          * ignore it.
2877          */
2878         if (argc <= 1)
2879                 return (0);

2881         /*
2882          * Because we aren't leveraging the lex and yacc engine, we have to
2883          * manually walk the arguments to find both the first and last
2884          * open/close quote of the format string.
2885          */
2886         f = strchr(argv[0].a_un.a_str, '"');
2887         if (f == NULL)
2888                 return (0);

2890         f = strchr(f + 1, '"');
2891         if (f != NULL) {
2892                 ii = 0;
2893         } else {
```



```
2894                 for (ii = 1; ii < argc; ii++) {
2895                         if (argv[ii].a_type != MDB_TYPE_STRING)
2896                                 continue;
2897                         f = strchr(argv[ii].a_un.a_str, '"');
2898                         if (f != NULL)
2899                                 break;
2900                 }
2901                 /* Never found */
2902                 if (ii == argc)
2903                         return (0);
2904         }

2906         ii++;
2907         argc -= ii;
2908         argv += ii;

2910         return (cmd_print_tab_common(mcp, flags, argc, argv));
2911 }

2913 int
2914 cmd_printf(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
2915 {
2916         char type[MDB_SYM_NAMLEN];
2917         int i, nfmts = 0, ret;
2918         mdb_ctf_id_t id;
2919         const char *fmt, *member;
2920         char **fmts, *last, *dest, f;
2921         int (**funcs)(mdb_ctf_id_t, uintptr_t, ulong_t, char *);
2922         int state = PRINTF_NOFMT;
2923         printarg_t pa;

2925         if (!(flags & DCMD_ADDRSPEC))
2926                 return (DCMD_USAGE);

2928         bzero(&pa, sizeof (pa));
2929         pa.pa_as = MDB_TGT_AS_VIRT;
2930         pa.pa_realtgt = pa.pa_tgt = mdb.m_target;

2932         if (argc == 0 || argv[0].a_type != MDB_TYPE_STRING) {
2933                 mdb_warn("expected a format string\n");
2934                 return (DCMD_USAGE);
2935         }

2937         /*
2938          * Our first argument is a format string; rip it apart and run it
2939          * through our state machine to validate that our input is within the
2940          * subset of mdb_printf() format strings that we allow.
2941          */
2942         fmt = argv[0].a_un.a_str;
2943         /*
2944          * 'dest' must be large enough to hold a copy of the format string,
2945          * plus a NUL and up to 2 additional characters for each conversion
2946          * in the format string.  This gives us a bloat factor of 5/2 ~= 3.
2947          *   e.g. "%d" (strlen of 2) --> "%lld\0" (need 5 bytes)
2948          */
2949         dest = mdb_zalloc(strlen(fmt) * 3, UM_SLEEP | UM_GC);
2950         fmts = mdb_zalloc(strlen(fmt) * sizeof (char *), UM_SLEEP | UM_GC);
2951         funcs = mdb_zalloc(strlen(fmt) * sizeof (void *), UM_SLEEP | UM_GC);
2952         last = dest;

2954         for (i = 0; fmt[i] != '\0'; i++) {
2955                 *dest++ = f = fmt[i];

2957                 switch (state) {
2958                 case PRINTF_NOFMT:
2959                         state = f == '%' ? PRINTF_PERC : PRINTF_NOFMT;
```

```
2960                    break;
2962            case PRINTF_PERC:
2963                    state = f == '-' ? PRINTF_LEFT :
2964                        f >= '0' && f <= '9' ? PRINTF_WIDTH :
2965                        f == '?' ? PRINTF_QUES :
2966                        f == '%' ? PRINTF_NOFMT : PRINTF_FMT;
2967                    break;
2969            case PRINTF_LEFT:
2970                    state = f >= '0' && f <= '9' ? PRINTF_WIDTH :
2971                        f == '?' ? PRINTF_QUES : PRINTF_FMT;
2972                    break;
2974            case PRINTF_WIDTH:
2975                    state = f >= '0' && f <= '9' ? PRINTF_WIDTH :
2976                        PRINTF_FMT;
2977                    break;
2979            case PRINTF_QUES:
2980                    state = PRINTF_FMT;
2981                    break;
2982            }
2984            if (state != PRINTF_FMT)
2985                    continue;
2987            dest--;
2989            /*
2990             * Now check that we have one of our valid format characters.
2991             */
2992            switch (f) {
2993            case 'a':
2994            case 'A':
2995            case 'p':
2996                    funcs[nfmts] = printf_ptr;
2997                    break;
2999            case 'd':
3000            case 'q':
3001            case 'R':
3002                    funcs[nfmts] = printf_int;
3003                    *dest++ = 'l';
3004                    *dest++ = 'l';
3005                    break;
3007            case 'I':
3008                    funcs[nfmts] = printf_uint32;
3009                    break;
3011            case 'N':
3012                    funcs[nfmts] = printf_ipv6;
3013                    break;
3015            case 'H':
3016            case 'o':
3017            case 'r':
3018            case 'u':
3019            case 'x':
3020            case 'X':
3021                    funcs[nfmts] = printf_uint;
3022                    *dest++ = 'l';
3023                    *dest++ = 'l';
3024                    break;
```



```
3026            case 's':
3027                    funcs[nfmts] = printf_string;
3028                    break;
3030            case 'Y':
3031                    funcs[nfmts] = sizeof (time_t) == sizeof (int) ?
3032                        printf_uint32 : printf_uint;
3033                    break;
3035            default:
3036                    mdb_warn("illegal format string at or near "
3037                        "'%c' (position %d)\n", f, i + 1);
3038                    return (DCMD_ABORT);
3039            }
3041            *dest++ = f;
3042            *dest++ = '\0';
3043            fmts[nfmts++] = last;
3044            last = dest;
3045            state = PRINTF_NOFMT;
3046    }
3048    argc--;
3049    argv++;
3051    /*
3052     * Now we expect a type name.
3053     */
3054    if ((ret = args_to_typename(&argc, &argv, type, sizeof (type))) != 0)
3055            return (ret);
3057    argv++;
3058    argc--;
3060    if (mdb_ctf_lookup_by_name(type, &id) != 0) {
3061            mdb_warn("failed to look up type %s", type);
3062            return (DCMD_ABORT);
3063    }
3065    if (argc == 0) {
3066            mdb_warn("at least one member must be specified\n");
3067            return (DCMD_USAGE);
3068    }
3070    if (argc != nfmts) {
3071            mdb_warn("%s format specifiers (found %d, expected %d)\n",
3072                argc > nfmts ? "missing" : "extra", nfmts, argc);
3073            return (DCMD_ABORT);
3074    }
3076    for (i = 0; i < argc; i++) {
3077            mdb_ctf_id_t mid;
3078            ulong_t off;
3079            int ignored;
3081            if (argv[i].a_type != MDB_TYPE_STRING) {
3082                    mdb_warn("expected only type member arguments\n");
3083                    return (DCMD_ABORT);
3084            }
3086            if (strcmp((member = argv[i].a_un.a_str), ".") == 0) {
3087                    /*
3088                     * We allow "." to be specified to denote the current
3089                     * value of dot.
3090                     */
3091                    if (funcs[i] != printf_ptr && funcs[i] != printf_uint &&
```

```
3092                            funcs[i] != printf_int) {
3093                                mdb_warn("expected integer or pointer format "
3094                                    "specifier for '.'\n");
3095                                return (DCMD_ABORT);
3096                            }

3098                            mdb_printf(fmts[i], mdb_get_dot());
3099                            continue;
3100                    }

3102                    pa.pa_addr = addr;

3104                    if (parse_member(&pa, member, id, &mid, &off, &ignored) != 0)
3105                            return (DCMD_ABORT);

3107                    if ((ret = funcs[i](mid, pa.pa_addr, off, fmts[i])) != 0) {
3108                            mdb_warn("failed to print member '%s'\n", member);
3109                            return (ret);
3110                    }
3111            }

3113            mdb_printf("%s", last);

3115            return (DCMD_OK);
3116    }

3118    static char _mdb_printf_help[] =
3119    "The format string argument is a printf(3C)-like format string that is a\n"
3120    "subset of the format strings supported by mdb_printf().  The type argument\n"
3121    "is the name of a type to be used to interpret the memory referenced by dot.\n"
3122    "The member should either be a field in the specified structure, or the\n"
3123    "special member '.', denoting the value of dot (and treated as a pointer).\n"
3124    "The number of members must match the number of format specifiers in the\n"
3125    "format string.\n"
3126    "\n"
3127    "The following format specifiers are recognized by ::printf:\n"
3128    "\n"
3129    "  %%    Prints the '%' symbol.\n"
3130    "  %a    Prints the member in symbolic form.\n"
3131    "  %d    Prints the member as a decimal integer.  If the member is a signed\n"
3132    "        integer type, the output will be signed.\n"
3133    "  %H    Prints the member as a human-readable size.\n"
3134    "  %I    Prints the member as an IPv4 address (must be 32-bit integer type).\n"
3135    "  %N    Prints the member as an IPv6 address (must be of type in6_addr_t).\n"
3136    "  %o    Prints the member as an unsigned octal integer.\n"
3137    "  %p    Prints the member as a pointer, in hexadecimal.\n"
3138    "  %q    Prints the member in signed octal.  Honk if you ever use this!\n"
3139    "  %r    Prints the member as an unsigned value in the current output radix.\n"
3140    "  %R    Prints the member as a signed value in the current output radix.\n"
3141    "  %s    Prints the member as a string (requires a pointer or an array of\n"
3142    "        characters).\n"
3143    "  %u    Prints the member as an unsigned decimal integer.\n"
3144    "  %x    Prints the member in hexadecimal.\n"
3145    "  %X    Prints the member in hexadecimal, using the characters A-F as the\n"
3146    "        digits for the values 10-15.\n"
3147    "  %Y    Prints the member as a time_t as the string "
3148               "'year month day HH:MM:SS'.\n"
3149    "\n"
3150    "The following field width specifiers are recognized by ::printf:\n"
3151    "\n"
3152    "  %n    Field width is set to the specified decimal value.\n"
3153    "  %?    Field width is set to the maximum width of a hexadecimal pointer\n"
3154    "        value.  This is 8 in an ILP32 environment, and 16 in an LP64\n"
3155    "        environment.\n"
3156    "\n"
3157    "The following flag specifers are recognized by ::printf:\n"
```



```
3158    "\n"
3159    "  %-    Left-justify the output within the specified field width.  If the\n"
3160    "        width of the output is less than the specified field width, the\n"
3161    "        output will be padded with blanks on the right-hand side.  Without\n"
3162    "        %-, values are right-justified by default.\n"
3163    "\n"
3164    "  %0    Zero-fill the output field if the output is right-justified and the\n"
3165    "        width of the output is less than the specified field width.  Without\n"
3166    "        %0, right-justified values are prepended with blanks in order to\n"
3167    "        fill the field.\n"
3168    "\n"
3169    "Examples: \n"
3170    "\n"
3171    "  ::walk proc | "
3172            "::printf \"%-6d %s\\n\" proc_t p_pidp->pid_id p_user.u_psargs\n"
3173    "  ::walk thread | "
3174            "::printf \"%?p %3d %a\\n\" kthread_t . t_pri t_startpc\n"
3175    "  ::walk zone | "
3176            "::printf \"%-40s %20s\\n\" zone_t zone_name zone_nodename\n"
3177    "  ::walk ire | "
3178            "::printf \"%Y %I\\n\" ire_t ire_create_time ire_u.ire4_u.ire4_addr\n"
3179    "\n";

3181    void
3182    printf_help(void)
3183    {
3184            mdb_printf("%s", _mdb_printf_help);
3185    }
```