```
**********************************************************
   75101 Mon Nov  9 12:24:03 2015
new/usr/src/common/nvpair/nvpair.c
6447 handful of nvpair cleanups
**********************************************************
______unchanged_portion_omitted_

1593 /*
1594  * Find the 'name'ed nvpair in the nvlist 'nvl'. If 'name' found, the function
1595  * returns zero and a pointer to the matching nvpair is returned in '*ret'
1596  * (given 'ret' is non-NULL). If 'sep' is specified then 'name' will penitrate
1597  * multiple levels of embedded nvlists, with 'sep' as the separator. As an
1598  * example, if sep is '.', name might look like: "a" or "a.b" or "a.c[3]" or
1599  * "a.d[3].e[1]".  This matches the C syntax for array embed (for convience,
1600  * code also supports "a.d[3]e[1]" syntax).
1601  *
1602  * If 'ip' is non-NULL and the last name component is an array, return the
1603  * value of the "...[index]" array index in *ip. For an array reference that
1604  * is not indexed, *ip will be returned as -1. If there is a syntax error in
1605  * 'name', and 'ep' is non-NULL then *ep will be set to point to the location
1606  * inside the 'name' string where the syntax error was detected.
1607  */
1608 static int
1609 nvlist_lookup_nvpair_ei_sep(nvlist_t *nvl, const char *name, const char sep,
1610     nvpair_t **ret, int *ip, char **ep)
1611 {
1612         nvpair_t        *nvp;
1613         const char      *np;
1614         char            *sepp;
1615         char            *idxp, *idxep;
1616         nvlist_t        **nva;
1617         long            idx;
1618         int             n;

1620         if (ip)
1621                 *ip = -1;                       /* not indexed */
1622         if (ep)
1623                 *ep = NULL;

1625         if ((nvl == NULL) || (name == NULL))
1626                 return (EINVAL);

1628         sepp = NULL;
1629         idx = 0;
1630 #endif /* ! codereview */
1631         /* step through components of name */
1632         for (np = name; np && *np; np = sepp) {
1633                 /* ensure unique names */
1634                 if (!(nvl->nvl_nvflag & NV_UNIQUE_NAME))
1635                         return (ENOTSUP);

1637                 /* skip white space */
1638                 skip_whitespace(np);
1639                 if (*np == 0)
1640                         break;

1642                 /* set 'sepp' to end of current component 'np' */
1643                 if (sep)
1644                         sepp = strchr(np, sep);
1645                 else
1646                         sepp = NULL;

1648                 /* find start of next "[ index ]..." */
1649                 idxp = strchr(np, '[');

1651                 /* if sepp comes first, set idxp to NULL */
```



```
1652                 if (sepp && idxp && (sepp < idxp))
1653                         idxp = NULL;

1655                 /*
1656                  * At this point 'idxp' is set if there is an index
1657                  * expected for the current component.
1658                  */
1659                 if (idxp) {
1660                         /* set 'n' to length of current 'np' name component */
1661                         n = idxp++ - np;

1663                         /* keep sepp up to date for *ep use as we advance */
1664                         skip_whitespace(idxp);
1665                         sepp = idxp;

1667                         /* determine the index value */
1668 #if defined(_KERNEL) && !defined(_BOOT)
1669                         if (ddi_strtol(idxp, &idxep, 0, &idx))
1670                                 goto fail;
1671 #else
1672                         idx = strtol(idxp, &idxep, 0);
1673 #endif
1674                         if (idxep == idxp)
1675                                 goto fail;

1677                         /* keep sepp up to date for *ep use as we advance */
1678                         sepp = idxep;

1680                         /* skip white space index value and check for ']' */
1681                         skip_whitespace(sepp);
1682                         if (*sepp++ != ']')
1683                                 goto fail;

1685                         /* for embedded arrays, support C syntax: "a[1].b" */
1686                         skip_whitespace(sepp);
1687                         if (sep && (*sepp == sep))
1688                                 sepp++;
1689                 } else if (sepp) {
1690                         n = sepp++ - np;
1691                 } else {
1692                         n = strlen(np);
1693                 }

1695                 /* trim trailing whitespace by reducing length of 'np' */
1696                 if (n == 0)
1697                         goto fail;
1698                 for (n--; (np[n] == ' ') || (np[n] == '\t'); n--)
1699                         ;
1700                 n++;

1702                 /* skip whitespace, and set sepp to NULL if complete */
1703                 if (sepp) {
1704                         skip_whitespace(sepp);
1705                         if (*sepp == 0)
1706                                 sepp = NULL;
1707                 }

1709                 /*
1710                  * At this point:
1711                  * o  'n' is the length of current 'np' component.
1712                  * o  'idxp' is set if there was an index, and value 'idx'.
1713                  * o  'sepp' is set to the beginning of the next component,
1714                  *    and set to NULL if we have no more components.
1715                  *
1716                  * Search for nvpair with matching component name.
1717                  */
```

```
1718                 for (nvp = nvlist_next_nvpair(nvl, NULL); nvp != NULL;
1719                     nvp = nvlist_next_nvpair(nvl, nvp)) {

1721                         /* continue if no match on name */
1722                         if (strncmp(np, nvpair_name(nvp), n) ||
1723                             (strlen(nvpair_name(nvp)) != n))
1724                                 continue;

1726                         /* if indexed, verify type is array oriented */
1727                         if (idxp && !nvpair_type_is_array(nvp))
1728                                 goto fail;

1730                         /*
1731                          * Full match found, return nvp and idx if this
1732                          * was the last component.
1733                          */
1734                         if (sepp == NULL) {
1735                                 if (ret)
1736                                         *ret = nvp;
1737                                 if (ip && idxp)
1738                                         *ip = (int)idx; /* return index */
1739                                 return (0);             /* found */
1740                         }

1742                         /*
1743                          * More components: current match must be
1744                          * of DATA_TYPE_NVLIST or DATA_TYPE_NVLIST_ARRAY
1745                          * to support going deeper.
1746                          */
1747                         if (nvpair_type(nvp) == DATA_TYPE_NVLIST) {
1748                                 nvl = EMBEDDED_NVL(nvp);
1749                                 break;
1750                         } else if (nvpair_type(nvp) == DATA_TYPE_NVLIST_ARRAY) {
1751                                 (void) nvpair_value_nvlist_array(nvp,
1752                                     &nva, (uint_t *)&n);
1753                                 if ((n < 0) || (idx >= n))
1754                                         goto fail;
1755                                 nvl = nva[idx];
1756                                 break;
1757                         }

1759                         /* type does not support more levels */
1760                         goto fail;
1761                 }
1762                 if (nvp == NULL)
1763                         goto fail;              /* 'name' not found */

1765                 /* search for match of next component in embedded 'nvl' list */
1766         }

1768 fail:   if (ep && sepp)
1769                 *ep = sepp;
1770         return (EINVAL);
1771 }

1773 /*
1774  * Return pointer to nvpair with specified 'name'.
1775  */
1776 int
1777 nvlist_lookup_nvpair(nvlist_t *nvl, const char *name, nvpair_t **ret)
1778 {
1779         return (nvlist_lookup_nvpair_ei_sep(nvl, name, 0, ret, NULL, NULL));
1780 }

1782 /*
1783  * Determine if named nvpair exists in nvlist (use embedded separator of '.'
```



```
1784  * and return array index).  See nvlist_lookup_nvpair_ei_sep for more detailed
1785  * description.
1786  */
1787 int nvlist_lookup_nvpair_embedded_index(nvlist_t *nvl,
1788     const char *name, nvpair_t **ret, int *ip, char **ep)
1789 {
1790         return (nvlist_lookup_nvpair_ei_sep(nvl, name, '.', ret, ip, ep));
1791 }

1793 boolean_t
1794 nvlist_exists(nvlist_t *nvl, const char *name)
1795 {
1796         nvpriv_t *priv;
1797         nvpair_t *nvp;
1798         i_nvp_t *curr;

1800         if (name == NULL || nvl == NULL ||
1801             (priv = (nvpriv_t *)(uintptr_t)nvl->nvl_priv) == NULL)
1802                 return (B_FALSE);

1804         for (curr = priv->nvp_list; curr != NULL; curr = curr->nvi_next) {
1805                 nvp = &curr->nvi_nvp;

1807                 if (strcmp(name, NVP_NAME(nvp)) == 0)
1808                         return (B_TRUE);
1809         }

1811         return (B_FALSE);
1812 }

1814 int
1815 nvpair_value_boolean_value(nvpair_t *nvp, boolean_t *val)
1816 {
1817         return (nvpair_value_common(nvp, DATA_TYPE_BOOLEAN_VALUE, NULL, val));
1818 }

1820 int
1821 nvpair_value_byte(nvpair_t *nvp, uchar_t *val)
1822 {
1823         return (nvpair_value_common(nvp, DATA_TYPE_BYTE, NULL, val));
1824 }

1826 int
1827 nvpair_value_int8(nvpair_t *nvp, int8_t *val)
1828 {
1829         return (nvpair_value_common(nvp, DATA_TYPE_INT8, NULL, val));
1830 }

1832 int
1833 nvpair_value_uint8(nvpair_t *nvp, uint8_t *val)
1834 {
1835         return (nvpair_value_common(nvp, DATA_TYPE_UINT8, NULL, val));
1836 }

1838 int
1839 nvpair_value_int16(nvpair_t *nvp, int16_t *val)
1840 {
1841         return (nvpair_value_common(nvp, DATA_TYPE_INT16, NULL, val));
1842 }

1844 int
1845 nvpair_value_uint16(nvpair_t *nvp, uint16_t *val)
1846 {
1847         return (nvpair_value_common(nvp, DATA_TYPE_UINT16, NULL, val));
1848 }
```

```
1850 int
1851 nvpair_value_int32(nvpair_t *nvp, int32_t *val)
1852 {
1853         return (nvpair_value_common(nvp, DATA_TYPE_INT32, NULL, val));
1854 }

1856 int
1857 nvpair_value_uint32(nvpair_t *nvp, uint32_t *val)
1858 {
1859         return (nvpair_value_common(nvp, DATA_TYPE_UINT32, NULL, val));
1860 }

1862 int
1863 nvpair_value_int64(nvpair_t *nvp, int64_t *val)
1864 {
1865         return (nvpair_value_common(nvp, DATA_TYPE_INT64, NULL, val));
1866 }

1868 int
1869 nvpair_value_uint64(nvpair_t *nvp, uint64_t *val)
1870 {
1871         return (nvpair_value_common(nvp, DATA_TYPE_UINT64, NULL, val));
1872 }

1874 #if !defined(_KERNEL)
1875 int
1876 nvpair_value_double(nvpair_t *nvp, double *val)
1877 {
1878         return (nvpair_value_common(nvp, DATA_TYPE_DOUBLE, NULL, val));
1879 }
1880 #endif

1882 int
1883 nvpair_value_string(nvpair_t *nvp, char **val)
1884 {
1885         return (nvpair_value_common(nvp, DATA_TYPE_STRING, NULL, val));
1886 }

1888 int
1889 nvpair_value_nvlist(nvpair_t *nvp, nvlist_t **val)
1890 {
1891         return (nvpair_value_common(nvp, DATA_TYPE_NVLIST, NULL, val));
1892 }

1894 int
1895 nvpair_value_boolean_array(nvpair_t *nvp, boolean_t **val, uint_t *nelem)
1896 {
1897         return (nvpair_value_common(nvp, DATA_TYPE_BOOLEAN_ARRAY, nelem, val));
1898 }

1900 int
1901 nvpair_value_byte_array(nvpair_t *nvp, uchar_t **val, uint_t *nelem)
1902 {
1903         return (nvpair_value_common(nvp, DATA_TYPE_BYTE_ARRAY, nelem, val));
1904 }

1906 int
1907 nvpair_value_int8_array(nvpair_t *nvp, int8_t **val, uint_t *nelem)
1908 {
1909         return (nvpair_value_common(nvp, DATA_TYPE_INT8_ARRAY, nelem, val));
1910 }

1912 int
1913 nvpair_value_uint8_array(nvpair_t *nvp, uint8_t **val, uint_t *nelem)
1914 {
1915         return (nvpair_value_common(nvp, DATA_TYPE_UINT8_ARRAY, nelem, val));
```



```
1916 }

1918 int
1919 nvpair_value_int16_array(nvpair_t *nvp, int16_t **val, uint_t *nelem)
1920 {
1921         return (nvpair_value_common(nvp, DATA_TYPE_INT16_ARRAY, nelem, val));
1922 }

1924 int
1925 nvpair_value_uint16_array(nvpair_t *nvp, uint16_t **val, uint_t *nelem)
1926 {
1927         return (nvpair_value_common(nvp, DATA_TYPE_UINT16_ARRAY, nelem, val));
1928 }

1930 int
1931 nvpair_value_int32_array(nvpair_t *nvp, int32_t **val, uint_t *nelem)
1932 {
1933         return (nvpair_value_common(nvp, DATA_TYPE_INT32_ARRAY, nelem, val));
1934 }

1936 int
1937 nvpair_value_uint32_array(nvpair_t *nvp, uint32_t **val, uint_t *nelem)
1938 {
1939         return (nvpair_value_common(nvp, DATA_TYPE_UINT32_ARRAY, nelem, val));
1940 }

1942 int
1943 nvpair_value_int64_array(nvpair_t *nvp, int64_t **val, uint_t *nelem)
1944 {
1945         return (nvpair_value_common(nvp, DATA_TYPE_INT64_ARRAY, nelem, val));
1946 }

1948 int
1949 nvpair_value_uint64_array(nvpair_t *nvp, uint64_t **val, uint_t *nelem)
1950 {
1951         return (nvpair_value_common(nvp, DATA_TYPE_UINT64_ARRAY, nelem, val));
1952 }

1954 int
1955 nvpair_value_string_array(nvpair_t *nvp, char ***val, uint_t *nelem)
1956 {
1957         return (nvpair_value_common(nvp, DATA_TYPE_STRING_ARRAY, nelem, val));
1958 }

1960 int
1961 nvpair_value_nvlist_array(nvpair_t *nvp, nvlist_t ***val, uint_t *nelem)
1962 {
1963         return (nvpair_value_common(nvp, DATA_TYPE_NVLIST_ARRAY, nelem, val));
1964 }

1966 int
1967 nvpair_value_hrtime(nvpair_t *nvp, hrtime_t *val)
1968 {
1969         return (nvpair_value_common(nvp, DATA_TYPE_HRTIME, NULL, val));
1970 }

1972 /*
1973  * Add specified pair to the list.
1974  */
1975 int
1976 nvlist_add_nvpair(nvlist_t *nvl, nvpair_t *nvp)
1977 {
1978         if (nvl == NULL || nvp == NULL)
1979                 return (EINVAL);

1981         return (nvlist_add_common(nvl, NVP_NAME(nvp), NVP_TYPE(nvp),
```

```
1982             NVP_NELEM(nvp), NVP_VALUE(nvp)));
1983 }

1985 /*
1986  * Merge the supplied nvlists and put the result in dst.
1987  * The merged list will contain all names specified in both lists,
1988  * the values are taken from nvl in the case of duplicates.
1989  * Return 0 on success.
1990  */
1991 /*ARGSUSED*/
1992 int
1993 nvlist_merge(nvlist_t *dst, nvlist_t *nvl, int flag)
1994 {
1995         if (nvl == NULL || dst == NULL)
1996                 return (EINVAL);

1998         if (dst != nvl)
1999                 return (nvlist_copy_pairs(nvl, dst));

2001         return (0);
2002 }

2004 /*
2005  * Encoding related routines
2006  */
2007 #define NVS_OP_ENCODE   0
2008 #define NVS_OP_DECODE   1
2009 #define NVS_OP_GETSIZE  2

2011 typedef struct nvs_ops nvs_ops_t;

2013 typedef struct {
2014         int             nvs_op;
2015         const nvs_ops_t *nvs_ops;
2016         void            *nvs_private;
2017         nvpriv_t        *nvs_priv;
2018 } nvstream_t;

2020 /*
2021  * nvs operations are:
2022  *   - nvs_nvlist
2023  *     encoding / decoding of a nvlist header (nvlist_t)
2024  *     calculates the size used for header and end detection
2025  *
2026  *   - nvs_nvpair
2027  *     responsible for the first part of encoding / decoding of an nvpair
2028  *     calculates the decoded size of an nvpair
2029  *
2030  *   - nvs_nvp_op
2031  *     second part of encoding / decoding of an nvpair
2032  *
2033  *   - nvs_nvp_size
2034  *     calculates the encoding size of an nvpair
2035  *
2036  *   - nvs_nvl_fini
2037  *     encodes the end detection mark (zeros).
2038  */
2039 struct nvs_ops {
2040         int (*nvs_nvlist)(nvstream_t *, nvlist_t *, size_t *);
2041         int (*nvs_nvpair)(nvstream_t *, nvpair_t *, size_t *);
2042         int (*nvs_nvp_op)(nvstream_t *, nvpair_t *);
2043         int (*nvs_nvp_size)(nvstream_t *, nvpair_t *, size_t *);
2044         int (*nvs_nvl_fini)(nvstream_t *);
2045 };

2047 typedef struct {
```



```
2048         char    nvh_encoding;   /* nvs encoding method */
2049         char    nvh_endian;     /* nvs endian */
2050         char    nvh_reserved1;  /* reserved for future use */
2051         char    nvh_reserved2;  /* reserved for future use */
2052 } nvs_header_t;

2054 static int
2055 nvs_encode_pairs(nvstream_t *nvs, nvlist_t *nvl)
2056 {
2057         nvpriv_t *priv = (nvpriv_t *)(uintptr_t)nvl->nvl_priv;
2058         i_nvp_t *curr;

2060         /*
2061          * Walk nvpair in list and encode each nvpair
2062          */
2063         for (curr = priv->nvp_list; curr != NULL; curr = curr->nvi_next)
2064                 if (nvs->nvs_ops->nvs_nvpair(nvs, &curr->nvi_nvp, NULL) != 0)
2065                         return (EFAULT);

2067         return (nvs->nvs_ops->nvs_nvl_fini(nvs));
2068 }

2070 static int
2071 nvs_decode_pairs(nvstream_t *nvs, nvlist_t *nvl)
2072 {
2073         nvpair_t *nvp;
2074         size_t nvsize;
2075         int err;

2077         /*
2078          * Get decoded size of next pair in stream, alloc
2079          * memory for nvpair_t, then decode the nvpair
2080          */
2081         while ((err = nvs->nvs_ops->nvs_nvpair(nvs, NULL, &nvsize)) == 0) {
2082                 if (nvsize == 0) /* end of list */
2083                         break;

2085                 /* make sure len makes sense */
2086                 if (nvsize < NVP_SIZE_CALC(1, 0))
2087                         return (EFAULT);

2089                 if ((nvp = nvp_buf_alloc(nvl, nvsize)) == NULL)
2090                         return (ENOMEM);

2092                 if ((err = nvs->nvs_ops->nvs_nvp_op(nvs, nvp)) != 0) {
2093                         nvp_buf_free(nvl, nvp);
2094                         return (err);
2095                 }

2097                 if (i_validate_nvpair(nvp) != 0) {
2098                         nvpair_free(nvp);
2099                         nvp_buf_free(nvl, nvp);
2100                         return (EFAULT);
2101                 }

2103                 nvp_buf_link(nvl, nvp);
2104         }
2105         return (err);
2106 }

2108 static int
2109 nvs_getsize_pairs(nvstream_t *nvs, nvlist_t *nvl, size_t *buflen)
2110 {
2111         nvpriv_t *priv = (nvpriv_t *)(uintptr_t)nvl->nvl_priv;
2112         i_nvp_t *curr;
2113         uint64_t nvsize = *buflen;
```

```
2114         size_t size;

2116         /*
2117          * Get encoded size of nvpairs in nvlist
2118          */
2119         for (curr = priv->nvp_list; curr != NULL; curr = curr->nvi_next) {
2120                 if (nvs->nvs_ops->nvs_nvp_size(nvs, &curr->nvi_nvp, &size) != 0)
2121                         return (EINVAL);

2123                 if ((nvsize += size) > INT32_MAX)
2124                         return (EINVAL);
2125         }

2127         *buflen = nvsize;
2128         return (0);
2129 }

2131 static int
2132 nvs_operation(nvstream_t *nvs, nvlist_t *nvl, size_t *buflen)
2133 {
2134         int err;

2136         if (nvl->nvl_priv == 0)
2137                 return (EFAULT);

2139         /*
2140          * Perform the operation, starting with header, then each nvpair
2141          */
2142         if ((err = nvs->nvs_ops->nvs_nvlist(nvs, nvl, buflen)) != 0)
2143                 return (err);

2145         switch (nvs->nvs_op) {
2146         case NVS_OP_ENCODE:
2147                 err = nvs_encode_pairs(nvs, nvl);
2148                 break;

2150         case NVS_OP_DECODE:
2151                 err = nvs_decode_pairs(nvs, nvl);
2152                 break;

2154         case NVS_OP_GETSIZE:
2155                 err = nvs_getsize_pairs(nvs, nvl, buflen);
2156                 break;

2158         default:
2159                 err = EINVAL;
2160         }

2162         return (err);
2163 }

2165 static int
2166 nvs_embedded(nvstream_t *nvs, nvlist_t *embedded)
2167 {
2168         switch (nvs->nvs_op) {
2169         case NVS_OP_ENCODE:
2170                 return (nvs_operation(nvs, embedded, NULL));

2172         case NVS_OP_DECODE: {
2173                 nvpriv_t *priv;
2174                 int err;

2176                 if (embedded->nvl_version != NV_VERSION)
2177                         return (ENOTSUP);

2179                 if ((priv = nv_priv_alloc_embedded(nvs->nvs_priv)) == NULL)
```



```
2180                         return (ENOMEM);

2182                 nvlist_init(embedded, embedded->nvl_nvflag, priv);

2184                 if ((err = nvs_operation(nvs, embedded, NULL)) != 0)
2185                         nvlist_free(embedded);
2186                 return (err);
2187         }
2188         default:
2189                 break;
2190         }

2192         return (EINVAL);
2193 }

2195 static int
2196 nvs_embedded_nvl_array(nvstream_t *nvs, nvpair_t *nvp, size_t *size)
2197 {
2198         size_t nelem = NVP_NELEM(nvp);
2199         nvlist_t **nvlp = EMBEDDED_NVL_ARRAY(nvp);
2200         int i;

2202         switch (nvs->nvs_op) {
2203         case NVS_OP_ENCODE:
2204                 for (i = 0; i < nelem; i++)
2205                         if (nvs_embedded(nvs, nvlp[i]) != 0)
2206                                 return (EFAULT);
2207                 break;

2209         case NVS_OP_DECODE: {
2210                 size_t len = nelem * sizeof (uint64_t);
2211                 nvlist_t *embedded = (nvlist_t *)((uintptr_t)nvlp + len);

2213                 bzero(nvlp, len);       /* don't trust packed data */
2214                 for (i = 0; i < nelem; i++) {
2215                         if (nvs_embedded(nvs, embedded) != 0) {
2216                                 nvpair_free(nvp);
2217                                 return (EFAULT);
2218                         }

2220                         nvlp[i] = embedded++;
2221                 }
2222                 break;
2223         }
2224         case NVS_OP_GETSIZE: {
2225                 uint64_t nvsize = 0;

2227                 for (i = 0; i < nelem; i++) {
2228                         size_t nvp_sz = 0;

2230                         if (nvs_operation(nvs, nvlp[i], &nvp_sz) != 0)
2231                                 return (EINVAL);

2233                         if ((nvsize += nvp_sz) > INT32_MAX)
2234                                 return (EINVAL);
2235                 }

2237                 *size = nvsize;
2238                 break;
2239         }
2240         default:
2241                 return (EINVAL);
2242         }

2244         return (0);
2245 }
```

```
2247 static int nvs_native(nvstream_t *, nvlist_t *, char *, size_t *);
2248 static int nvs_xdr(nvstream_t *, nvlist_t *, char *, size_t *);

2250 /*
2251  * Common routine for nvlist operations:
2252  * encode, decode, getsize (encoded size).
2253  */
2254 static int
2255 nvlist_common(nvlist_t *nvl, char *buf, size_t *buflen, int encoding,
2256     int nvs_op)
2257 {
2258         int err = 0;
2259         nvstream_t nvs;
2260         int nvl_endian;
2261 #ifdef  _LITTLE_ENDIAN
2262         int host_endian = 1;
2263 #else
2264         int host_endian = 0;
2265 #endif  /* _LITTLE_ENDIAN */
2266         nvs_header_t *nvh = (void *)buf;

2268         if (buflen == NULL || nvl == NULL ||
2269             (nvs.nvs_priv = (nvpriv_t *)(uintptr_t)nvl->nvl_priv) == NULL)
2270                 return (EINVAL);

2272         nvs.nvs_op = nvs_op;

2274         /*
2275          * For NVS_OP_ENCODE and NVS_OP_DECODE make sure an nvlist and
2276          * a buffer is allocated.  The first 4 bytes in the buffer are
2277          * used for encoding method and host endian.
2278          */
2279         switch (nvs_op) {
2280         case NVS_OP_ENCODE:
2281                 if (buf == NULL || *buflen < sizeof (nvs_header_t))
2282                         return (EINVAL);

2284                 nvh->nvh_encoding = encoding;
2285                 nvh->nvh_endian = nvl_endian = host_endian;
2286                 nvh->nvh_reserved1 = 0;
2287                 nvh->nvh_reserved2 = 0;
2288                 break;

2290         case NVS_OP_DECODE:
2291                 if (buf == NULL || *buflen < sizeof (nvs_header_t))
2292                         return (EINVAL);

2294                 /* get method of encoding from first byte */
2295                 encoding = nvh->nvh_encoding;
2296                 nvl_endian = nvh->nvh_endian;
2297                 break;

2299         case NVS_OP_GETSIZE:
2300                 nvl_endian = host_endian;

2302                 /*
2303                  * add the size for encoding
2304                  */
2305                 *buflen = sizeof (nvs_header_t);
2306                 break;

2308         default:
2309                 return (ENOTSUP);
2310         }
```



```
2312         /*
2313          * Create an nvstream with proper encoding method
2314          */
2315         switch (encoding) {
2316         case NV_ENCODE_NATIVE:
2317                 /*
2318                  * check endianness, in case we are unpacking
2319                  * from a file
2320                  */
2321                 if (nvl_endian != host_endian)
2322                         return (ENOTSUP);
2323                 err = nvs_native(&nvs, nvl, buf, buflen);
2324                 break;
2325         case NV_ENCODE_XDR:
2326                 err = nvs_xdr(&nvs, nvl, buf, buflen);
2327                 break;
2328         default:
2329                 err = ENOTSUP;
2330                 break;
2331         }

2333         return (err);
2334 }

2336 int
2337 nvlist_size(nvlist_t *nvl, size_t *size, int encoding)
2338 {
2339         return (nvlist_common(nvl, NULL, size, encoding, NVS_OP_GETSIZE));
2340 }

2342 /*
2343  * Pack nvlist into contiguous memory
2344  */
2345 /*ARGSUSED1*/
2346 int
2347 nvlist_pack(nvlist_t *nvl, char **bufp, size_t *buflen, int encoding,
2348     int kmflag)
2349 {
2350 #if defined(_KERNEL) && !defined(_BOOT)
2351         return (nvlist_xpack(nvl, bufp, buflen, encoding,
2352             (kmflag == KM_SLEEP ? nv_alloc_sleep : nv_alloc_nosleep)));
2353 #else
2354         return (nvlist_xpack(nvl, bufp, buflen, encoding, nv_alloc_nosleep));
2355 #endif
2356 }

2358 int
2359 nvlist_xpack(nvlist_t *nvl, char **bufp, size_t *buflen, int encoding,
2360     nv_alloc_t *nva)
2361 {
2362         nvpriv_t nvpriv;
2363         size_t alloc_size;
2364         char *buf;
2365         int err;

2367         if (nva == NULL || nvl == NULL || bufp == NULL || buflen == NULL)
2368                 return (EINVAL);

2370         if (*bufp != NULL)
2371                 return (nvlist_common(nvl, *bufp, buflen, encoding,
2372                     NVS_OP_ENCODE));

2374         /*
2375          * Here is a difficult situation:
2376          * 1. The nvlist has fixed allocator properties.
2377          *    All other nvlist routines (like nvlist_add_*, ...) use
```

```
2378          *    these properties.
2379          * 2. When using nvlist_pack() the user can specify his own
2380          *    allocator properties (e.g. by using KM_NOSLEEP).
2381          *
2382          * We use the user specified properties (2). A clearer solution
2383          * will be to remove the kmflag from nvlist_pack(), but we will
2384          * not change the interface.
2385          */
2386         nv_priv_init(&nvpriv, nva, 0);

2388         if ((err = nvlist_size(nvl, &alloc_size, encoding)))
1628         if (err = nvlist_size(nvl, &alloc_size, encoding))
2389                 return (err);

2391         if ((buf = nv_mem_zalloc(&nvpriv, alloc_size)) == NULL)
2392                 return (ENOMEM);

2394         if ((err = nvlist_common(nvl, buf, &alloc_size, encoding,
2395             NVS_OP_ENCODE)) != 0) {
2396                 nv_mem_free(&nvpriv, buf, alloc_size);
2397         } else {
2398                 *buflen = alloc_size;
2399                 *bufp = buf;
2400         }

2402         return (err);
2403 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   33489 Mon Nov  9 12:24:03 2015
new/usr/src/lib/libnvpair/libnvpair.c
6447 handful of nvpair cleanups
**********************************************************
```

```
/*
 * 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 (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

#include <unistd.h>
#include <string.h>
#include <strings.h>
#include <libintl.h>
#include <sys/types.h>
#include <sys/inttypes.h>
#include <stdarg.h>
#include <note.h>
#include "libnvpair.h"

/*
 * libnvpair - A tools library for manipulating <name, value> pairs.
 *
 *      This library provides routines packing an unpacking nv pairs
 *      for transporting data across process boundaries, transporting
 *      between kernel and userland, and possibly saving onto disk files.
 */

/*
 * Print control structure.
 */

#define DEFINEOP(opname, vtype) \
        struct { \
                int (*op)(struct nvlist_prtctl *, void *, nvlist_t *, \
                    const char *, vtype); \
                void *arg; \
        } opname

#define DEFINEARROP(opname, vtype) \
        struct { \
                int (*op)(struct nvlist_prtctl *, void *, nvlist_t *, \
                    const char *, vtype, uint_t); \
                void *arg; \
        } opname
```



```
struct nvlist_printops {
        DEFINEOP(print_boolean, int);
        DEFINEOP(print_boolean_value, boolean_t);
        DEFINEOP(print_byte, uchar_t);
        DEFINEOP(print_int8, int8_t);
        DEFINEOP(print_uint8, uint8_t);
        DEFINEOP(print_int16, int16_t);
        DEFINEOP(print_uint16, uint16_t);
        DEFINEOP(print_int32, int32_t);
        DEFINEOP(print_uint32, uint32_t);
        DEFINEOP(print_int64, int64_t);
        DEFINEOP(print_uint64, uint64_t);
        DEFINEOP(print_double, double);
        DEFINEOP(print_string, char *);
        DEFINEOP(print_hrtime, hrtime_t);
        DEFINEOP(print_nvlist, nvlist_t *);
        DEFINEARROP(print_boolean_array, boolean_t *);
        DEFINEARROP(print_byte_array, uchar_t *);
        DEFINEARROP(print_int8_array, int8_t *);
        DEFINEARROP(print_uint8_array, uint8_t *);
        DEFINEARROP(print_int16_array, int16_t *);
        DEFINEARROP(print_uint16_array, uint16_t *);
        DEFINEARROP(print_int32_array, int32_t *);
        DEFINEARROP(print_uint32_array, uint32_t *);
        DEFINEARROP(print_int64_array, int64_t *);
        DEFINEARROP(print_uint64_array, uint64_t *);
        DEFINEARROP(print_string_array, char **);
        DEFINEARROP(print_nvlist_array, nvlist_t **);
};
```

_____*unchanged_portion_omitted_*

```
NVLIST_PRTFUNC(boolean, int, int, "%d")
NVLIST_PRTFUNC(boolean_value, boolean_t, int, "%d")
NVLIST_PRTFUNC(byte, uchar_t, uchar_t, "0x%2.2x")
NVLIST_PRTFUNC(int8, int8_t, int, "%d")
NVLIST_PRTFUNC(uint8, uint8_t, uint8_t, "0x%x")
NVLIST_PRTFUNC(int16, int16_t, int16_t, "%d")
NVLIST_PRTFUNC(uint16, uint16_t, uint16_t, "0x%x")
NVLIST_PRTFUNC(int32, int32_t, int32_t, "%d")
NVLIST_PRTFUNC(uint32, uint32_t, uint32_t, "0x%x")
NVLIST_PRTFUNC(int64, int64_t, longlong_t, "%lld")
NVLIST_PRTFUNC(uint64, uint64_t, u_longlong_t, "0x%llx")
NVLIST_PRTFUNC(double, double, double, "0x%f")
NVLIST_PRTFUNC(double, double, double, "0x%llf")
NVLIST_PRTFUNC(string, char *, char *, "%s")
NVLIST_PRTFUNC(hrtime, hrtime_t, hrtime_t, "0x%llx")

/*
 * Generate functions to print array-valued nvlist members.
 */

#define NVLIST_ARRPRTFUNC(type_and_variant, vtype, ptype, vfmt) \
static int \
nvaprint_##type_and_variant(nvlist_prtctl_t pctl, void *private, \
    nvlist_t *nvl, const char *name, vtype *valuep, uint_t count) \
{ \
        FILE *fp = pctl->nvprt_fp; \
        uint_t i; \
        NOTE(ARGUNUSED(private)) \
        NOTE(ARGUNUSED(nvl)) \
        for (i = 0; i < count; i++) { \
                if (i == 0 || pctl->nvprt_btwnarrfmt_nl) { \
                        indent(pctl, 1); \
                        (void) fprintf(fp, pctl->nvprt_nmfmt, name); \
                        if (pctl->nvprt_btwnarrfmt_nl) \
                                (void) fprintf(fp, "[%d]: ", i); \
```

```
237                 } \
238                 if (i != 0) \
239                         (void) fprintf(fp, pctl->nvprt_btwnarrfmt); \
240                 (void) fprintf(fp, vfmt, (ptype)valuep[i]); \
241         } \
242         return (1); \
243 }
```

*unchanged_portion_omitted_*

```
934 /*
935  * ========================================================================
936  * |                                                                      |
937  * | Misc private interface.                                              |
938  * |                                                                      |
939  * ========================================================================
940  */

942 /*
943  * Determine if string 'value' matches 'nvp' value.  The 'value' string is
944  * converted, depending on the type of 'nvp', prior to match.  For numeric
945  * types, a radix independent sscanf conversion of 'value' is used. If 'nvp'
946  * is an array type, 'ai' is the index into the array against which we are
947  * checking for match. If nvp is of DATA_TYPE_STRING*, the caller can pass
948  * in a regex_t compilation of value in 'value_regex' to trigger regular
949  * expression string match instead of simple strcmp().
950  *
951  * Return 1 on match, 0 on no-match, and -1 on error.  If the error is
952  * related to value syntax error and 'ep' is non-NULL, *ep will point into
953  * the 'value' string at the location where the error exists.
954  *
955  * NOTE: It may be possible to move the non-regex_t version of this into
956  * common code used by library/kernel/boot.
957  */
958 int
959 nvpair_value_match_regex(nvpair_t *nvp, int ai,
960     char *value, regex_t *value_regex, char **ep)
961 {
962         char    *evalue;
963         uint_t  a_len;
964         int     sr;

966         if (ep)
967                 *ep = NULL;

969         if ((nvp == NULL) || (value == NULL))
970                 return (-1);            /* error fail match - invalid args */

972         /* make sure array and index combination make sense */
973         if ((nvpair_type_is_array(nvp) && (ai < 0)) ||
974             (!nvpair_type_is_array(nvp) && (ai >= 0)))
975                 return (-1);            /* error fail match - bad index */

977         /* non-string values should be single 'chunk' */
978         if ((nvpair_type(nvp) != DATA_TYPE_STRING) &&
979             (nvpair_type(nvp) != DATA_TYPE_STRING_ARRAY)) {
980                 value += strspn(value, " \t");
981                 evalue = value + strcspn(value, " \t");
982                 if (*evalue) {
983                         if (ep)
984                                 *ep = evalue;
985                         return (-1);    /* error fail match - syntax */
986                 }
987         }

989         sr = EOF;
990         switch (nvpair_type(nvp)) {
```



```
991         case DATA_TYPE_STRING: {
992                 char    *val;

994                 /* check string value for match */
995                 if (nvpair_value_string(nvp, &val) == 0) {
996                         if (value_regex) {
997                                 if (regexec(value_regex, val,
998                                     (size_t)0, NULL, 0) == 0)
999                                         return (1);     /* match */
1000                        } else {
1001                                if (strcmp(value, val) == 0)
1002                                        return (1);     /* match */
1003                        }
1004                }
1005                break;
1006        }
1007        case DATA_TYPE_STRING_ARRAY: {
1008                char **val_array;

1010                /* check indexed string value of array for match */
1011                if ((nvpair_value_string_array(nvp, &val_array, &a_len) == 0) &&
1012                    (ai < a_len)) {
1013                        if (value_regex) {
1014                                if (regexec(value_regex, val_array[ai],
1015                                    (size_t)0, NULL, 0) == 0)
1016                                        return (1);
1017                        } else {
1018                                if (strcmp(value, val_array[ai]) == 0)
1019                                        return (1);
1020                        }
1021                }
1022                break;
1023        }
1024        case DATA_TYPE_BYTE: {
1025                uchar_t val, val_arg;

1027                /* scanf uchar_t from value and check for match */
1028                sr = sscanf(value, "%c", &val_arg);
1029                if ((sr == 1) && (nvpair_value_byte(nvp, &val) == 0) &&
1030                    (val == val_arg))
1031                        return (1);
1032                break;
1033        }
1034        case DATA_TYPE_BYTE_ARRAY: {
1035                uchar_t *val_array, val_arg;


1038                /* check indexed value of array for match */
1039                sr = sscanf(value, "%c", &val_arg);
1040                if ((sr == 1) &&
1041                    (nvpair_value_byte_array(nvp, &val_array, &a_len) == 0) &&
1042                    (ai < a_len) &&
1043                    (val_array[ai] == val_arg))
1044                        return (1);
1045                break;
1046        }
1047        case DATA_TYPE_INT8: {
1048                int8_t val, val_arg;

1050                /* scanf int8_t from value and check for match */
1051                sr = sscanf(value, "%"SCNi8, &val_arg);
1052                if ((sr == 1) &&
1053                    (nvpair_value_int8(nvp, &val) == 0) &&
1054                    (val == val_arg))
1055                        return (1);
1056                break;
```

```
1057         }
1058         case DATA_TYPE_INT8_ARRAY: {
1059                 int8_t *val_array, val_arg;

1061                 /* check indexed value of array for match */
1062                 sr = sscanf(value, "%"SCNi8, &val_arg);
1063                 if ((sr == 1) &&
1064                     (nvpair_value_int8_array(nvp, &val_array, &a_len) == 0) &&
1065                     (ai < a_len) &&
1066                     (val_array[ai] == val_arg))
1067                         return (1);
1068                 break;
1069         }
1070         case DATA_TYPE_UINT8: {
1071                 uint8_t val, val_arg;

1073                 /* scanf uint8_t from value and check for match */
1074                 sr = sscanf(value, "%"SCNi8, (int8_t *)&val_arg);
1075                 if ((sr == 1) &&
1076                     (nvpair_value_uint8(nvp, &val) == 0) &&
1077                     (val == val_arg))
1078                         return (1);
1079                 break;
1080         }
1081         case DATA_TYPE_UINT8_ARRAY: {
1082                 uint8_t *val_array, val_arg;

1084                 /* check indexed value of array for match */
1085                 sr = sscanf(value, "%"SCNi8, (int8_t *)&val_arg);
1086                 if ((sr == 1) &&
1087                     (nvpair_value_uint8_array(nvp, &val_array, &a_len) == 0) &&
1088                     (ai < a_len) &&
1089                     (val_array[ai] == val_arg))
1090                         return (1);
1091                 break;
1092         }
1093         case DATA_TYPE_INT16: {
1094                 int16_t val, val_arg;

1096                 /* scanf int16_t from value and check for match */
1097                 sr = sscanf(value, "%"SCNi16, &val_arg);
1098                 if ((sr == 1) &&
1099                     (nvpair_value_int16(nvp, &val) == 0) &&
1100                     (val == val_arg))
1101                         return (1);
1102                 break;
1103         }
1104         case DATA_TYPE_INT16_ARRAY: {
1105                 int16_t *val_array, val_arg;

1107                 /* check indexed value of array for match */
1108                 sr = sscanf(value, "%"SCNi16, &val_arg);
1109                 if ((sr == 1) &&
1110                     (nvpair_value_int16_array(nvp, &val_array, &a_len) == 0) &&
1111                     (ai < a_len) &&
1112                     (val_array[ai] == val_arg))
1113                         return (1);
1114                 break;
1115         }
1116         case DATA_TYPE_UINT16: {
1117                 uint16_t val, val_arg;

1119                 /* scanf uint16_t from value and check for match */
1120                 sr = sscanf(value, "%"SCNi16, (int16_t *)&val_arg);
1121                 if ((sr == 1) &&
1122                     (nvpair_value_uint16(nvp, &val) == 0) &&
```



```
1123                     (val == val_arg))
1124                         return (1);
1125                 break;
1126         }
1127         case DATA_TYPE_UINT16_ARRAY: {
1128                 uint16_t *val_array, val_arg;

1130                 /* check indexed value of array for match */
1131                 sr = sscanf(value, "%"SCNi16, (int16_t *)&val_arg);
1132                 if ((sr == 1) &&
1133                     (nvpair_value_uint16_array(nvp, &val_array, &a_len) == 0) &&
1134                     (ai < a_len) &&
1135                     (val_array[ai] == val_arg))
1136                         return (1);
1137                 break;
1138         }
1139         case DATA_TYPE_INT32: {
1140                 int32_t val, val_arg;

1142                 /* scanf int32_t from value and check for match */
1143                 sr = sscanf(value, "%"SCNi32, &val_arg);
1144                 if ((sr == 1) &&
1145                     (nvpair_value_int32(nvp, &val) == 0) &&
1146                     (val == val_arg))
1147                         return (1);
1148                 break;
1149         }
1150         case DATA_TYPE_INT32_ARRAY: {
1151                 int32_t *val_array, val_arg;

1153                 /* check indexed value of array for match */
1154                 sr = sscanf(value, "%"SCNi32, &val_arg);
1155                 if ((sr == 1) &&
1156                     (nvpair_value_int32_array(nvp, &val_array, &a_len) == 0) &&
1157                     (ai < a_len) &&
1158                     (val_array[ai] == val_arg))
1159                         return (1);
1160                 break;
1161         }
1162         case DATA_TYPE_UINT32: {
1163                 uint32_t val, val_arg;

1165                 /* scanf uint32_t from value and check for match */
1166                 sr = sscanf(value, "%"SCNi32, (int32_t *)&val_arg);
1167                 if ((sr == 1) &&
1168                     (nvpair_value_uint32(nvp, &val) == 0) &&
1169                     (val == val_arg))
1170                         return (1);
1171                 break;
1172         }
1173         case DATA_TYPE_UINT32_ARRAY: {
1174                 uint32_t *val_array, val_arg;

1176                 /* check indexed value of array for match */
1177                 sr = sscanf(value, "%"SCNi32, (int32_t *)&val_arg);
1178                 if ((sr == 1) &&
1179                     (nvpair_value_uint32_array(nvp, &val_array, &a_len) == 0) &&
1180                     (ai < a_len) &&
1181                     (val_array[ai] == val_arg))
1182                         return (1);
1183                 break;
1184         }
1185         case DATA_TYPE_INT64: {
1186                 int64_t val, val_arg;

1188                 /* scanf int64_t from value and check for match */
```

```
1189                 sr = sscanf(value, "%"SCNi64, &val_arg);
1190                 if ((sr == 1) &&
1191                     (nvpair_value_int64(nvp, &val) == 0) &&
1192                     (val == val_arg))
1193                         return (1);
1194                 break;
1195         }
1196         case DATA_TYPE_INT64_ARRAY: {
1197                 int64_t *val_array, val_arg;

1199                 /* check indexed value of array for match */
1200                 sr = sscanf(value, "%"SCNi64, &val_arg);
1201                 if ((sr == 1) &&
1202                     (nvpair_value_int64_array(nvp, &val_array, &a_len) == 0) &&
1203                     (ai < a_len) &&
1204                     (val_array[ai] == val_arg))
1205                                 return (1);
1206                 break;
1207         }
1208         case DATA_TYPE_UINT64: {
1209                 uint64_t val_arg, val;

1211                 /* scanf uint64_t from value and check for match */
1212                 sr = sscanf(value, "%"SCNi64, (int64_t *)&val_arg);
1213                 if ((sr == 1) &&
1214                     (nvpair_value_uint64(nvp, &val) == 0) &&
1215                     (val == val_arg))
1216                         return (1);
1217                 break;
1218         }
1219         case DATA_TYPE_UINT64_ARRAY: {
1220                 uint64_t *val_array, val_arg;

1222                 /* check indexed value of array for match */
1223                 sr = sscanf(value, "%"SCNi64, (int64_t *)&val_arg);
1224                 if ((sr == 1) &&
1225                     (nvpair_value_uint64_array(nvp, &val_array, &a_len) == 0) &&
1226                     (ai < a_len) &&
1227                     (val_array[ai] == val_arg))
1228                         return (1);
1229                 break;
1230         }
1231         case DATA_TYPE_BOOLEAN_VALUE: {
1232                 int32_t val_arg;
1233                 boolean_t val;
1232                 boolean_t val, val_arg;

1235                 /* scanf boolean_t from value and check for match */
1236                 sr = sscanf(value, "%"SCNi32, &val_arg);
1237                 if ((sr == 1) &&
1238                     (nvpair_value_boolean_value(nvp, &val) == 0) &&
1239                     (val == val_arg))
1240                         return (1);
1241                 break;
1242         }
1243         case DATA_TYPE_BOOLEAN_ARRAY: {
1244                 boolean_t *val_array;
1245                 int32_t val_arg;
1243                 boolean_t *val_array, val_arg;

1247                 /* check indexed value of array for match */
1248                 sr = sscanf(value, "%"SCNi32, &val_arg);
1249                 if ((sr == 1) &&
1250                     (nvpair_value_boolean_array(nvp,
1251                     &val_array, &a_len) == 0) &&
1252                     (ai < a_len) &&
```



```
1253                         (val_array[ai] == val_arg))
1254                             return (1);
1255                 break;
1256         }
1257         case DATA_TYPE_HRTIME:
1258         case DATA_TYPE_NVLIST:
1259         case DATA_TYPE_NVLIST_ARRAY:
1260         case DATA_TYPE_BOOLEAN:
1261         case DATA_TYPE_DOUBLE:
1262         case DATA_TYPE_UNKNOWN:
1263         default:
1264                 /*
1265                  * unknown/unsupported data type
1266                  */
1267                 return (-1);            /* error fail match */
1268         }

1270         /*
1271          * check to see if sscanf failed conversion, return approximate
1272          * pointer to problem
1273          */
1274         if (sr != 1) {
1275                 if (ep)
1276                         *ep = value;
1277                 return (-1);            /* error fail match  - syntax */
1278         }

1280         return (0);                     /* fail match */
1281 }
```

_____***unchanged_portion_omitted_***