D

Parser and Builder Example

This appendix contains example field parser and builder functions for SIDF. The macros and type defines used by the functions are listed below and the support routines for these functions are listed after the examples.

Compiler

The compiler used for the examples in this appendix is Watcom version 9.0. The following compiler flags are used:

CFLAGS= /3s /ez /mf /olt /s /w4 /we /zp1/zq

3s	= Use 386 stack calling convention (as opposed to register calling convention).
ez	= Generate PharLap EZ-OMF object files.
mf	= Use flat memory model.
olt	= Optimize loops and favor execution speed over code size.
s	= Do not check for stack overflow.
w4	= Turn all warnings on.
we	= Do not produce object file if warnings are present.
zp1	= No alignment of structure members.
zq	= Suppress messages to screen.

Macros and Type Defines

```
#define CheckUnderflow(n) \
   if (bufferSize < (n))\
      { ccode = UNDERFLOW; goto Return; }\
   else bufferSize -= (n)

#define FixedFid(fid)((longFid) ? (((fid) & 0xF000) == 0xF000) : ((fid) & 0x40))

#define FixedSize(fid) \
   (1L << ((longFid) ? (*((UINT8 *)&(fid) + 1) AND 0x0F) : ((fid) AND 0x0F)))

#define GT(a, b)(*((UINT32 *)a + 1) or *(UINT32 *)a > b)

#define PutUINT64(a, v)(*((UINT32 *)(a) + 1) = 0, *(UINT32 *)(a) = (v))

#define ZeroUINT64(a)(*(UINT32 *)(a) = *((UINT32 *)(a) + 1) = 0)

typedef  unsigned char* BUFFERPTR;
typedef  unsigned long  CCODE;
typedef  unsigned char* PSTRING;
typedef  unsigned char  UINT8;
typedef  unsigned int   UINT16;
typedef  unsigned long  UINT32;

typedef struct
{
   UINT16 v[4];
} UINT64;
```

```
typedef struct
{
   UINT32   fid;
   UINT64   dataSize;
   void   *data;
   UINT32   bytesTransfered;
   UINT64   dataOverflow;
} FIELD_DATA;
```

Example Field Parser Function

GetNextField parses a buffer containing SIDF fields. The calling routine should consider the following:

- The calling routine must make sure that the returned fields are the fields it expects.
- The function does not move the buffer pointer after getting a field. If this function is called repeatedly, the calling function must readjust *buffer* to point to the next field and decrement *bufferSize* to indicate how much data is left each time before this function is called.
- It must not write into the buffer containing the data.
- If *dataOverflow* is non-zero, the rest of the data is in the next buffer. The calling routine must retrieve this data manually.
- The routine does not handle FIDs that spans a buffer.

The function finds the end of the FID, copies it into field->fid backwards, gets the data's size, and points to the data. This function uses the following algorithm:

```
Get the first byte
If we have a long FID
    Get the next byte
    if we have a short or long FID
        if we have an extended FID
            move the buffer pointer to the byte that tells us
                if we have a short or long FID
        else we have a standard or developer FID, move the
            buffer pointer to the byte that tells us if we
            have a short or long FID
        next we move the buffer pointer the appropriate
            number of bytes to the end of the FID
    else we have a small FID, move the pointer 1 byte to
        the end of the FID
    Copy the FID into "field->fid" backwards
    Get the data's size and data
```

```
CCODE GetNextField(
   BUFFERPTR   buffer,     /* A pointer to the field to be parsed. */
   UINT32      bufferSize, /* The size of the buffer containing the field. */
   FIELD_DATA *field)

   /* on return field contains the following:
      fid               Contains the value of the Field ID parsed.
      dataSize          Contains the size of the data.
      data              Points to the data contained in buffer.  Do not pass a
                        buffer with data when calling this function.
      bytesTransfered   Contains the number of bytes read from buffer.
      dataOverflow      Contains the number of bytes not transferred. */
{
   BUFFERPTR tempFieldPtr = buffer, ptr1, ptr2 = (BUFFERPTR)&field->fid;
   CCODE ccode = 0;
   UINT32 sizeOfData = 0;
   UINT8 longFid = FALSE;
   ZeroUINT64(&field->dataOverflow);

   /* There should be at least one byte in the buffer.  If there isn't, then we have
      a buffer-underflow condition, see above macro */

   CheckUnderflow(sizeof(UINT8));

   /* The following statements, up until "field->fid = 0," move the buffer pointer
      to the end of the FID */
   /* Get the next byte. If bit 7 is set, we have a short or long FID; otherwise we
      have a small FID */
   if (*tempFieldPtr & 0x80)
   {
      /* We have a short or long FID. Next we check bit six to see if we have an
         extended FID */
      if (*tempFieldPtr & 0x40)
      {
         /* We have an extended FID.  Check for underflow condition and move the
            buffer pointer to the byte that tells us if we have a short or long FID
            */
         CheckUnderflow(sizeof(UINT8));
         tempFieldPtr += sizeof(UINT16);
      }

      else /*  We have a standard or developer FID. We now move the pointer to the
               byte that tells us if we have a short or long FID. */
         tempFieldPtr++;

      /* next we move the buffer pointer to the end of the FID */
      /* Check for underflow condition and check if we have a short or long FID */
      CheckUnderflow(sizeof(UINT8));
      if (*tempFieldPtr & 0x80)
      {
         /* We have a long FID.  Check for bufferunderflow and move the pointer to
            the end of the FID */
         CheckUnderflow(sizeof(UINT8));
         tempFieldPtr += sizeof(UINT16);
         longFid = TRUE;
      }

      else /*  We have a short FID, move the pointer to the end of the FID */
         tempFieldPtr++;
   }

   else /*  We have a small FID, move the pointer to the end of the FID */
      tempFieldPtr++;

   field->fid = 0;
```

```
/* Convert the FID back to small endian (low-high byte order).  SIDF uses a byte
      stream for the FIDs and data size descriptors. */
for (ptr1 = tempFieldPtr - 1; ptr1 >= buffer; *ptr2++ = *ptr1--)
   ;

/* Get the data's size and data */
if (!field->fid)
{
   /* We have a NULL FID (see page 1-27) */
   field->data = NULL;
   sizeOfData = 0;
   SetUINT64(&field->dataSize, &sizeOfData, 4);
   goto Return;
}

else if (FixedFid(field->fid))
{
   /* We have a fixed size FID.  Point to the data and get its size */
   field->data = (void *)tempFieldPtr;
   sizeOfData = FixedSize(field->fid);
   SetUINT64(&field->dataSize, &sizeOfData, 4);

   if (sizeOfData > bufferSize)
      goto DataOverflow;

   else /* we are done */
      goto Return;
}

/* We have a variable size FID. tempFieldPtr is pointing to the size descriptor
   byte(s). Check for buffer underflow condition and find out what kind of size
   descriptor we have. */
CheckUnderflow(sizeof(UINT8));
if (*tempFieldPtr & 0x80)
{
   /* We have a format 2 or 3 size descriptor. */
   if (*tempFieldPtr & 0x40)
   {
      /* We have a format 3 size descriptor (see page 1-29). The lower 6 bits of
         tempFieldPtr contains the data. */
      field->data = (void *)tempFieldPtr;  /*   caution: the upper two bits are
                                                not cleared */
      sizeOfData = 1;
      SetUINT64(&field->dataSize, &sizeOfData, 4);
   }

   else  /* We have a format 2 size descriptor (see page 1-28). */
   {
      UINT16 sizeOfSize;
      /* The lower two bits contains the exponent "n" of the data's size.  The
         number of size bytes that follows is 2^n.  Get the size and point to the
         data. */

      sizeOfSize = 1 << (*tempFieldPtr++ & 3);
      CheckUnderflow(sizeOfSize);

      SetUINT64(&field->dataSize, (void *)tempFieldPtr, sizeOfSize);
      tempFieldPtr += sizeOfSize;
      field->data = (void *)tempFieldPtr;

      if (GT(&field->dataSize, bufferSize))
         goto DataOverflow;

      sizeOfData = *(UINT32 *)&field->dataSize;
   }
```

```
   }

   else  /* We have a size format 1 descriptor (see page 1-28).  The data's size is
            between 0 and 127 bytes inclusive. Get the size and point to the data.*/
   {
      sizeOfData = *tempFieldPtr++;
      field->data = sizeOfData ? (void *)tempFieldPtr : NULL;
      SetUINT64(&field->dataSize, &sizeOfData, 4);

      if (sizeOfData > bufferSize)
         goto DataOverflow;
   }

Return:
   if (!ccode)
   {
      tempFieldPtr += sizeOfData;
      field->bytesTransfered = tempFieldPtr - buffer;
   }
   return (ccode);

DataOverflow:
   field->bytesTransfered = tempFieldPtr - buffer + bufferSize;
   field->dataOverflow = field->dataSize;
   DecrementUINT64(&field->dataOverflow, bufferSize);
   return (ccode);
}
```

Example Field Builder Function

PutNextField puts one field into a buffer. The calling function must figure out the FID values and data's size. This function takes care of placing the data into the buffer appropriately. The function does not deal with buffer overflow conditions or minimum buffer size needs.

```
CCODE PutNextField(
   BUFFERPTR   buffer,     /* Points to the next location that will contain the
                              field. */
   UINT32      bufferSize, /* The size of the buffer to put field in */
   FIELD_DATA *field,      /* field contains the following:
                                 fid:  A FID value, range:
                                    0x00 - 0x7F,
                                    0x8000 - 0xFFFF,
                                    0x808000 - 0xBFFFFF, and
                                    0xC00000000 - 0xFFFFFFFF.

                                 dataSize: The number of data bytes to put into
                                    buffer.  The validity of this parameter is
                                    determined by dataSizeMap.

                                 data:  The address of the data to be put in the
                                    buffer.

                                 bytesTransfered:  The number of bytes of data
                                    written to the buffer.

                                 dataOverflow:  The number of data bytes not written
                                 to the buffer. */
```

```
   UINT8       dataSizeMap,/* The data size:
                              0 - 0x7F (0 to 127 bytes, dataSize is undefined),
                              0x80 (dataSize has data size, PutNextField will
                                 format the size information),
                              0xC0 - 0xFF (bit data, dataSize is undefined, data
                                 is in the lower size bits of dataSizeMap). */

   UINT32      sizeOfData) /* The size of the data passed in to be put in the
                              buffer. */
{
   BUFFERPTR tempFieldPtr = buffer, tempPtr;
   CCODE ccode = 0;
   UINT32 size;
   UINT8 longFid = FALSE;

   /* Find out if we have a long FID */
   size = SizeOfUINT32Data0(field->fid);
   if (size == 4 || (size == 3 && !(field->fid & 0x400000L)))
      longFid = TRUE;

   /* Convert the FID from a byte stream to small endian??).  SIDF specifies that
      FIDs and data size descriptors must be in this order. */
   for (tempPtr = (BUFFERPTR)&field->fid + size - 1; size; --size)
      *tempFieldPtr++ = *tempPtr--;

   /* Set dataSize and transfer the data */
   ZeroUINT64(&field->dataOverflow);

   if (FixedFid(field->fid))
   {
      if ((size = FixedSize(field->fid)) < sizeOfData)
      {
         ccode = INVALID_PARAMETER;
         goto Return;
      }

      SetUINT64(&field->dataSize, &size, 4);
   }

   else /* we have a variable size FID */
   {
      if (dataSizeMap & 0x80)
      {
         /* We have a format 2 or 3 data size descriptor */
         if (dataSizeMap & 0x40)
         {
            /* We have a format 3 data size descriptor. The data are bits. One byte
               contains the size descriptor and data. */
            *tempFieldPtr++ = dataSizeMap;
            goto Return;
         }

         else /* We have a format 2 data size descriptor */
         {
            UINT16 sizeOfSize;

            /* Figure out the size of the data size descriptor, and move it into the
               buffer. */
            sizeOfSize = SizeOfFieldData(field->dataSize, dataSizeMap);
            if (!dataSizeMap)
               dataSizeMap = *(UINT8 *)&field->dataSize;
```

```
            *tempFieldPtr++ = dataSizeMap;
            if (sizeOfSize)
            {
               memcpy((PSTRING)tempFieldPtr, &field->dataSize, sizeOfSize);
               tempFieldPtr += sizeOfSize;
            }

            /* Compare sizeOfData and dataSize */
            if (LT(&field->dataSize, sizeOfData))
            {
               ccode = INVALID_PARAMETER;
               goto Return;
            }
         }
      }

      else  /* We have a format 1 data size descriptor */
      {
         /* Data size is 1 byte */
         *tempFieldPtr++ = dataSizeMap;
         if (sizeOfData > dataSizeMap)
         {
            ccode = INVALID_PARAMETER;
            goto Return;
         }
      }
   }

   bufferSize -= (tempFieldPtr - buffer);

   /* Move the data to the buffer */
   if (sizeOfData > bufferSize)
   {
      UINT32 dataOverflow;

      dataOverflow = sizeOfData - bufferSize;
      SetUINT64(&field->dataOverflow, &dataOverflow, 4);
      size = bufferSize;
   }

   else
      size = sizeOfData;

   if (size)
      memcpy(tempFieldPtr, field->data, size);

   tempFieldPtr += size;

Return:
   if (!ccode)
      field->bytesTransfered = tempFieldPtr - buffer;

   return (ccode);
}
```

Support Functions

```
CCODE Borrow(
   UINT64   *value,
   INT16  index)
{
   if (index >= 4)
      return (UNDERFLOW);

   if (!value->v[index])
   {
      value->v[index] = 0xFFFF;
      return (Borrow(value, index + 1));
   }

   --value->v[index];
   return (0);
}



CCODE DecrementUINT64(  /* a -= b */
      UINT64   *a,
      UINT32    b)
{
   UINT64 x, y;

   x = *a;
   PutUINT64(&y, b);
   return (SubUINT64(&x, &y, a));
}



CCODE SubUINT64(  /* dif = a - b */
      UINT64 *a,
      UINT64 *b,
      UINT64 *dif)
{
   UINT32 borrow;
   INT16 index;

   for (index = 0; index < 4; index++)
   {
      if (a->v[index] < b->v[index])
      {
         borrow = 0x10000L;
         if (Borrow(a, index + 1))
            break;
      }

      else
         borrow = 0;

      dif->v[index] = borrow + a->v[index] - b->v[index];
   }

   return (borrow ? UNDERFLOW : 0);
}
```

```
CCODE SetUINT64(
      UINT64   *a,
      void  *buffer,
      UINT16    bufferSize)
{
   CCODE ccode = 0;

   ZeroUINT64(a);
   if (buffer)
   {
      if (n > 8)
         ccode = OVERFLOW;

      else if (n)
         memcpy((void *)a, buffer, n);
   }

   return (ccode);
}
```