Symbol Table Loading From Copybooks

CML00039-01

Code Magus Limited (England reg. no. 4024745)
Number 6, 69 Woodstock Road
Oxford, OX2 6EY, United Kingdom
www.codemagus.com

December 15, 2020

Contents

This document describes a library and its interface that can be used to parse copybooks and create symbol table data structures suitable for fast look ups of symbol attributes.

At present only a C interface is supported and only COBOL copybooks can be parsed into the symbol table data structures. The portions of COBOL that can appear in the copybooks are only the descriptions of the data items. This means that copybooks with, for example, COBOL `FD` entries in them are not accepted. The targeted level of COBOL is ANSI-85.

1 Usage

In order to use the library, the header file `symbols.h` should be included in the program. This header defines a number of structures that describe symbol table entries. Also defined are a number of prototypes that are used to create symbol table data structures and to perform look up requests for symbols by name. Duplicate symbols are permitted, but symbol look up must always be qualified by the 01-level name or the copybook member name (the scope of the name). Duplicates, either within an 01-level or within a copybook member cannot be located, only the last defined entry can be located and this entry displaces all the previous entries within the same scope.

The header file `symbols.h`, includes the required data structures required to describe the picture clauses and the hash table data structures required to locate symbols.

1.1 Initialising the interface

When "opening" a symbol table data structure using the `symbols_open` function, the path or file mask of where to find the copybooks to be processed is included in the initial call:

```
symbols_t *symbols_open(char *file mask, unsigned char flags);
```

When the structure is opened, a file name mask is supplied. This is either the name of the file which should be processed and it is assumed that this file will contain all the copy books whose symbols must be made available. This is indicated by the default setting of the flags value (zero). Alternatively, the file name mask may contain an embedded "%s" sequence. The presence of this sequence is indicated by setting the bit `SYMFL_MASKING` on in the flags. In this case, whenever a structure is not found, an attempt will be made to process the file name indicated by the structure name by inserting the name of the structure in the string (via sprintf). If this does not result in the the structure being made available or if the symbol is subsequently found not to be a member of the structure, then a `NULL` is returned.

The SYMFL_MASKING option is available only when the SYMFL_LAZY option is also selected. When this option is selected, the symbols are made available only when demanded. In this case the structure should be the name that completes the file name of SYMFL_MASKING is used. This name will be treated as an alias of the first field found in the file.

If the SYMFL_MASKING option is not used but SYMFL_LAZY is indicated then the file name passed on the symbols_open call will be processed on the first symbols_lookup call and the SYMFL_LAZY flag will be switched off.

If the SYMFL_REMDASH option is used, then a flag is set to inform the lexical analysis all dashes within identifiers should be converted to the under-score character. This is done so that the name can be used in environments other than COBOL. In such environments the dash would be confused with the arithmetic operator for subtraction.

If the SYMFL_EBCDIC or SYMFL_ASCII option is used, then the default character set which is assigned as the representing character set of an item is overridden. The default value is chosen when a data item which is represented as characters does not have an explicitly assigned collating sequence. The normal value of the default is the character set of the computer on which the library is being used. You should not use the SYMFL_EBCDIC flag together with the SYMFL_ASCII flag.

If the SYMFL_ENDSMALL or SYMFL_ENDBIG option is used, then the default endian of binary items is overridden by the corresponding value. The default value is chosen when a data item which is represented as a binary data item does not have an explicitly assigned endianness. The normal value of the default endianness is that of the computer on which the library is being used. You should not use the SYMFL_ENDSMALL and SYMFL_ENDBIG flags together.

If the SYMFL_NORAWBUF option is used, then a flag is set then the buffer print routines ignore portions of a buffer which were not selected as being formatted because the detection of the layout of the buffer. If no record matches a buffer, then the entire buffer is printed as a hexadecimal dump regardless. This option refers to those portions of a selected buffer which are not chosen. For example, because of no applicable overlay.

If the SYMFL_DATAONLY option is used, then a level indicator is set for the lexical analyser which forces it to interpret reserved words or keywords that are not required for data item definition to be recognised as identifiers. This is in addition to the use of the colon character in front of a symbol to force it to be interpreted as an identifier.

```
#include "symbols.h"

symbols_t *symbols;
int flags;

flags = SYMFL_LAZY|SYMFL_MASKING;
symbols = symbols_open("/home/stephen/copybooks/%s",flags);
```

The same code works on OS/390 where copybooks are stored in PDSs or PDSEs. The only changes required would be to the file name mask:

```
symbols = symbols_open("`XXXXXXXX.YY.COPY(%s)'",flags);
```

1.2 Locating symbols

Once a symbol table structure has been opened, symbols can be located using the `symbols_lookup` function:

```
data_item_t *symbols_lookup(symbols_t *symbols,
   unsigned char *structure, unsigned char *fieldname);
```

The `structure` parameter is value returned from the corresponding `symbols_open` function call (many symbol table structures can be open at the same time). The `structure` parameter is the qualifying name. This qualifying name is either the copybook name or the name of the 01-level (in this case, the copybook must have already been parsed).

The `symbols_lookup` function call returns the address of the symbol table entry for the qualified symbol or a `NULL` should the process fail to parse a required copybook or the symbol is not found in the copybook.

```
data_item_t *field;
char *structure;
char *fieldname;

field = symbols_lookup(symbols,structure,fieldname);
```

The pointer returned form the `symbols_lookup` can be dereferenced from code outside the library. The following attributes of the symbol are made available in this manner (see the `symbols_t` in `actions.h`):

attribute	description
*name	name of data item
level	data item level number
llevel	data item logical level number
offset	data item offset
place	value from OFFSET phrase
length	number of bytes of element
storage	number of bytes occupied
cover	field is a covering field
	zero for elementary field, one for group item
*type	type information
*type	see type_info_t in picture.h
sync	indicates synchronised item
usage	final usage assigned
usage_specified	from usage clause
catagory	bits indicating abstract type
	CATAGORY_BITS 0x1f
	ALPHA 0x01
	NUMERIC 0x02
	EDIT 0x04
	NUMERIC_EDIT (NUMERIC|EDIT)
	ALPHANUMERIC (ALPHA|NUMERIC)
	ALPHANUMERIC_EDIT (ALPHA|NUMERIC|EDIT)
	DBCS 0x08
	FLOAT 0x10
*redefines	possible item redefined
sign	bits for sign clause
	zero for no sign clause specified
	0x01 bit for sign leading
	0x02 bit for sign trailing
	0x12 bit for sign separate
blank_when_zero	blank when zero clause used
justified	justified clause used
*value	value clause specified
	see value_t in actions.h
*conditions	list of 88-level items
	these are also data_item_t structures
	iand are chained off the link field
*occurs	details from occurs clause
	see occurs_t in actions.h
*rename	details from rename clause
	see rename_t in actions.h
hidden	hidden by HIDDEN or HIDE phrase
chilren	count of children from this field
pack	function to convert display to value
	see item_pack_t in actions.h
unpack	function to convert value to display
	see item_unpack_t in actions.h
*attributes	attribute value pairs
	see the hash table library for interface

If the symbol is indexed (the `occurs` attribute will refer to a structure if the `OCCURS` clause was used in the definition of the data item and/or the data item is nested within group level items defined with `OCCURS` clauses), then the offset attribute refers to first occurrence of the item (possibly within the first occurrence within a covering occurring field, etc.). There are two functions which take a set of indices and a symbol table entry returned from `symbols_lookup` and return the actual offset of the indexed field:

```
int symbols_vindex(data_item_t *symbol, int levels, ...);
int symbols_index(data_item_t *symbol, int levels, int *index);
```

The only difference between the two functions is that in the first (`symbols_vindex`) expects all the index values to be passed as parameters and the second (`symbols_index`) expects all the indexes passed with a single parameter. This parameter is an array containing all the index values. The second is more useful as it does not require the hard-coding of the number of indexes:

```
int offset;
int levels;
int index[20];

offset = symbols_index(field,levels,index);
```

If the number of indexes do not match the symbol (and the symbols containing the symbol), then the function returns -1 as the offset (zero is a valid offset—it is the offset of the first item in an 01-level or the offset of the 01-level symbol itself). If the `SYMFL_VERBOSE` flag is set when the symbol table structure was created, then an error message will also be printed. The verbose option can also be selected from shell by defining the `SYMFL_VERBOSE` as an environment variable and setting the value appropriately:

```
[stephen@nomad symbols]$ export SYMFL_VERBOSE=1
```

The data structure returned from the `symbols` library shown in Figure 1 can be navigated in various ways. This is useful, for example, for iterating over the fields of a copybook.

The `symbols_t` data structure returned from the `symbols_open` comprises of a number of linked lists and trees. The fields `head` and `tail` form a linked list of all the fields defined in the copybook. The fields can be iterated over by following the `next` field of the `data_item_t` elements. The following code fragment shows an iteration over all the fields defined in a copybook regardless of which record the field belongs to and regardless of the relationship between the fields:

```
for (parent = symbols->head; parent; parent = parent->next)
   if (parent->level == 01 || lengths > 1)
      {
      if (parent->level == 01) adjust = 0;
```

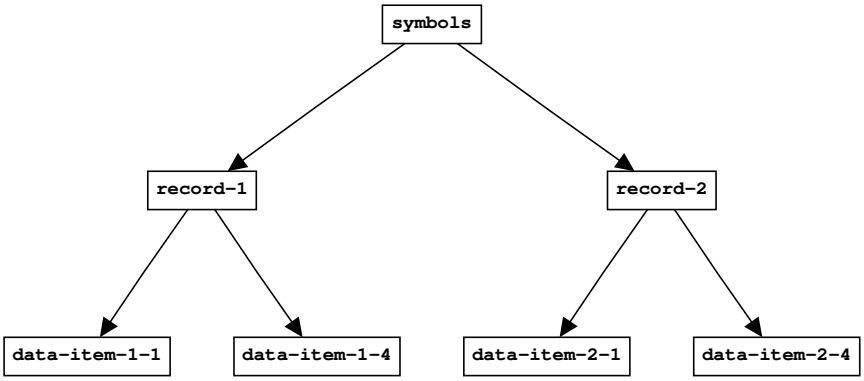

Figure 1: Simplified tree view of the `symbols` data structure

```
if (parent->offset+adjust != parent->place)
   {
   misalign = 1;
   adjust -= parent->offset+adjust - parent->place;
   }
else
   misalign = 0;

if (parent->name[0] == '-')
   printf("FILLER %d %d %d %d %d\n",parent->length,
      parent->storage,parent->offset,parent->place,
      misalign);
else
   printf("%s %d %d %d %d %d\n",parent->name,
      parent->length,parent->storage,parent->offset,
      parent->place,misalign);
}
```

The `symbols_t` fields `record_head` and `record_tail` form a linked list of `record_t` structures. The records can be iterated over by following the `link` field in the `record_t` structure. Each of these data structures defines a grouping of the fields defined in a single 01-level or 77-level item. The following code fragment shows an iteration over all the records that might have been defined in a single copybook:

```
for (record = symbols->record_head; record; record = record->link)
   {
   record_map = record;
   for (field = record->head; field; field = field->link)
```

```
      {
      if (record->cover[field->offset] == 1
            && !symbols_eval_value(field,length,buffer))
         {
         record_map = NULL;
         break;
         }
      }
   if (record_map) break;
   }
```

The `record_t` structure in turn contains a linked list of the fields under the 01-level record. This linked list is formed by the fields `head` and `tail` in the `record_t` structure. When iterating over the fields in a `record_t` structure, the `link` field should be followed. The code fragment above shows an example of iterating over all the fields defined in a 01-level record.

The relationship between the fields of a single record also forms a tree hierarchy. This tree is also represented in the data structure at the record level. Because a node in the tree (i.e. a field) can have any number of children, the tree is represented by a child-peer data structure. In this data structure the first child of a node is pointed to by the `child` field. The the next sibling of this child is pointed to by the previous child itself (and not the parent), and so on. This allows for an arbitrary tree structure without wasting space and admits a simple recursive navigation for iterating over all of the field nodes in a record. The following code fragment shows an example of a recursive traversal of the tree data structure:

```
  /* Function symbols_has_values() evaluates whether or not a subtree h
   * value clauses.
   */

int symbols_has_values(data_item_t *field)
   {
   data_item_t *child;

   if (field->cover)
      {
      for (child = field->child; child; child = child->peer)
         if (symbols_has_values(child)) return 1;
      return 0;
      }

   if (field->value || field->conditions) return 1;

   return 0;
```

```
} /* symbols_has_values */
```

2 Printing the contents of buffers

The symbols library can also be used to print the contents of a buffer in hand using a collection of records (01-level items) loaded from a copybook. There are a number of exported functions which support buffer printing.

Function `symbols_printbuf` will format a buffer using a supplied `record_t` structure. The contents of the buffer will be printed according to the definition of the fields described in the supplied record. The prototype for function `symbols_printbuf` is:

```
void symbols_printbuf(symbols_t *symbols, record_t *record,
      int len, unsigned char *buffer);
```

Function `symbols_formatbuf` works similarly except that it is not given the record which should be used to format the buffer. Instead, the buffer contents are examined and compared with the contents of any `value` clauses present. Both the `value` clauses on items at levels 01 to 49 are examined and the `value` clauses on 88-level items. The prototype for function `symbols_formatbuf` is:

```
void symbols_formatbuf(symbols_t *symbols, int length, unsigned char *b
```

When considering which record to choose, only the `value` clauses associated withe items that describe an unambiguous portions of the buffer are taken into account. For example, if there were a `value` clause on an item which was a redefinition of a portion of the record, then the `value` clause would not be taken into account when considering the suitability of the record for describing the buffer in hand. All unambiguous `value` clauses on field items must match the inorder for the record to be chosen as a layout. Where there is a list of `value` clauses (for example a sequence of 88-level items or a list of values in the definition of a single data item), on one of the values needs to match in order to consider that that field has matched the buffer contents.

The function `symbols_choosemap` is used to decide on a record for a given buffer. This function is also exported and can be used independently of the use described here. The prototype for this function is:

```
record_t *symbols_choosemap(symbols_t *symbols, int length,
            unsigned char *buffer);
```

3 Associating attributes withe data items

There may be a case for associating arbitrary attributes with a given data item. This can be done as an appropriately placed embedded comment in the copybook. These at-

tributes take the form of name-value pairs. Both the name and the value are represented as quoted strings. In the following example, the attribute ENCRYPTED is associated with a data item called ACCOUNT-NUMBER and is given a value of DES-OP-KEY:

```
000000001111111111
1234567890123456789
           05 ACCOUNT-NUMBER           PIC X(19).
      *attribute set ACCOUNT-NUMBER["ENCRYPTED"] = "DES-OP-KEY".
```

The values of attributes can be set and retrieved programmatically as well. This is accomplished by the functions symbols_set_attribute and symbols_get_attribute, respectively. These functions have the following prototypes:

```
char *symbols_get_attribute(data_item_t *field, char *attribute);
char *symbols_set_attribute(data_item_t *field, char *attribute, char *
```

Functions to route to the corresponding pack and unpack routines. These routines are determined by the data items type and are set at the time when the symbol table is created. The functions return zero if the translation occurred correctly, else -1 is returned.

The function symbols_pack converts the item in the buffer described by the field into a string and the function symbols_unpack takes a string and converts the item to the type described by the field and stores it into the buffer. For some of the conversion either the character set encoding or the endian of the binary items need to be taken into account. In these cases the attributes of the local machine are used unless they have been specified on the symbols_open call; or unless they have been explicitly provided at the field or group or covering field level.

```
int symbols_pack(data_item_t *field, unsigned char *buf,
      unsigned char *value, int levels, int *index);
int symbols_unpack(data_item_t *field, unsigned char *buf,
      unsigned char *value, int levels, int *index);
```

3.1 Disconnecting and cleanup

Once the interface is no longer needed, the symbols_close function can be used to free the data structures occupied by the symbol tables:

```
int symbols_close(symbols_t *symbols);
```

The symbols argument is the symbol table handle returned from the symbols_open function when the interface was initialised.

3.2 Complete example

The following C program demonstrates the use of the API. This sample, `testprog.c` is part of the source distribution and is used for testing the C interface.

```
/*
 * File: testprog.c
 *
 * This program is a sample using the symbols library.  The program
 * prompts for qualified field names. The copybooks corresponding the
 * the symbols are lazily loaded. The path to the copybooks is hard-
 * coded in the program.
 *
 * Author: Stephen Donaldson.
 */

/*
 * $Author: hayward $
 * $Date: 2009/11/30 13:47:15 $
 * $Id: symref.tex,v 1.9 2009/11/30 13:47:15 hayward Exp $
 * $Name:  $
 * $Revision: 1.9 $
 * $State: Exp $
 *
 * $Log: symref.tex,v $
 * Revision 1.9  2009/11/30 13:47:15  hayward
 * Change to new title page.
 *
 * Revision 1.8  2008/12/18 08:59:36  hayward
 * Change to new header format.
 *
 * Revision 1.7  2004/10/19 11:45:12  stephen
 * Allow F and FIL as abbreviations of FILLER
 *
 * Revision 1.6  2002/12/26 10:45:23  stephen
 * Add C++ wrapper and document pack and unpack functions
 *
 * Revision 1.5  2002/08/15 20:31:49  stephen
 * Maintenance June/July/Aug 2002 JHB
 *
 * Revision 1.12  2002/06/06 05:45:39  stephen
 * Include memdebug.h for possible memory debug build
 *
 * Revision 1.11  2002/05/29 18:50:15  stephen
 * Resource cleanup on close and maintain attributes for data items
 *
```

```
   * Revision 1.10  2002/01/13 15:08:21  cvs
   * Changes to support metadata processing
   *
   * Revision 1.9  2001/06/30 10:01:39  stephen
   * Fix flag dependency and build lib archive
   *
   * Revision 1.8  2000/12/27 10:08:29  stephen
   * Add additional options for testing
   *
   * Revision 1.7  2000/12/26 15:57:52  stephen
   * Cleanup and introduce V1R0M0 tag
   *
   * Revision 1.6  2000/12/20 07:54:52  stephen
   * Option to convert dashes to underscores
   *
   * Revision 1.5  2000/12/19 17:26:12  stephen
   * Changes to use storage and occurs
   *
   * Revision 1.4  2000/12/19 13:06:17  stephen
   * Definable symbol LAZY_OPEN for lazy open
   *
   * Revision 1.3  2000/12/15 15:12:45  stephen
   * Enlarge default hash table size. Hash table malloc
   * fixed (should be size times sizeof pointer---and not size
   * times size of structure).
   *
   * The sample testprog.c loads all the books in a non-lazy
   * fashion from allbooks. (Just for testing.)
   *
   * Revision 1.2  2000/12/14 20:08:27  stephen
   * Memory debugging header added files changed to unconditionnaly include
   * the header (memdebug.h)
   *
   * Cleanup code.
   *
   * If the structure name (member containing copybooks) and the first field
   * name are different then add all the symbols defined in the member (from
   * all the records) under a single structure name (the name of the member).
   *
   * Revision 1.1.1.1  2000/12/14 13:25:14  stephen
   * Add file for symbols module
   *
   *
   */

static char *cvs =
```

```
   "$Id: symref.tex,v 1.9 2009/11/30 13:47:15 hayward Exp $";

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "memdebug.h"
#include "copynote.h"
#include "symbols.h"

int main(int argc, char *argv[])
   {
   symbols_t *symbols;
   char fullname[200];
   char *structure;
   char *fieldname;
   char *indices;
   char *indstart;
   char *delim;
   data_item_t *field;
   int index[20];
   int levels;
   int offset;
   int flags;
   char *type;

   COPYNOTE;

   # ifdef LAZY_OPEN

      flags = SYMFL_LAZY|SYMFL_MASKING|SYMFL_VERBOSE;
      # ifdef USE_DASHES
         flags |= SYMFL_REMDASH;
      # endif /* USE_DASHES */

      # ifdef ASK_PATH_MASK
         printf("Enter path mask: ");
         fgets(fullname,sizeof(fullname),stdin);

         delim = strchr(fullname,'\n');
         if (delim) *delim = 0;
         delim = strchr(fullname,' ');
         if (delim) *delim = 0;

         symbols = symbols_open(fullname,flags);
      # else /* !ASK_PATH_MASK */
```

```
      symbols = symbols_open("/home/stephen/nedcor/copybooks/%s",flags);
   # endif /* ASK_PATH_MASK */

# else /* !LAZY_OPEN */
   symbols = symbols_open("/home/stephen/nedcor/copybooks/allbooks",0);
# endif /* LAZY_OPEN */

if (!symbols)
   {
   fprintf(stderr,"Failed to create symbol table structures.\n");
   exit(0);
   }

while (1)
   {
   printf("Fully qualified Symbol: ");
   delim = fgets(fullname,sizeof(fullname),stdin);
   if (!delim || fullname[0] == '.') break;

   delim = strchr(fullname,'\n');
   if (delim) *delim = 0;
   delim = strchr(fullname,' ');
   if (delim) *delim = 0;

   delim = strchr(fullname,'.');
   if (!delim)
      {
      fprintf(stderr,"Field qualification error in %s\n",fullname);
      continue;
      }

   structure = fullname;
   *delim = 0;
   fieldname = delim+1;

   delim = strchr(fieldname,'(');
   if (delim)
      {
      *delim = 0;
      indices = delim+1;
      }
   else
      indices = 0;

   indstart = indices;
   for (levels = 0; indices && strlen(indices);)
```

```
         {
         if (sscanf(indices,"%d",&index[levels]) == 1)
            {
            levels++;
            for (; strlen(indices) && isdigit(*indices); indices++);
            for (; strlen(indices) && ispunct(*indices); indices++);
            }
         }

      field = symbols_lookup(symbols,structure,fieldname);

      if (field)
         {
         print_data_item(stdout,field);

         offset = symbols_index(field,levels,index);

         if (indstart && offset >= 0)
            printf("   %s (%s has offset %d.\n",field->name,indstart,offset);

         type = symbols_get_attribute(field,"TYPE");
         if (type)
            printf("\n   Symbol has override type of %s\n",type);

         printf("\n");
         }

      } /* while */

   symbols_close(symbols);
   exit(0);

   } /* main */
```

Assume that the directory `/home/stephen/copybooks` contains the following copybook under the file name `XX`:

```
000080 01  AAAAAAAA.
000090     05 BBBBBBBB                        PIC X(88).
000100     05 CCCCCCCC OCCURS 200.
              10 DDDDDDDD OCCURS 2.
                 15 EEEEEEEE OCCURS  10      PIC X(10).
           05 WWWWWWWW.
000090        15 FFFFFFFF                     PIC X.
000090        15 GGGGGGGG                     PIC X.
```

The program `testprog` can be used to query the symbols in the copybooks in the

directory /home/stephen/copybooks:

```
[stephen@nomad symbols]$ ./testprog
[./testprog] $Id: symref.tex,v 1.9 2009/11/30 13:47:15 hayward Exp $
Copyright (c) 2000 by Stephen Donaldson [stephen@codemagus.com].
Fully qualified Symbol: XX.FFFFFFFF

   FFFFFFFF
      level = 15/4, offset = 40088, length = 1, cover = 0
      sync = 0, usage = display, sign = 0, bwz = 0
      storage = 1, occurances = 1
      Type info:
         Pic = X, Cat type = alpha numeric[03],
         usage_allowed = default, length = 1

Fully qualified Symbol: AAAAAAAA.EEEEEEEE

   EEEEEEEE
      level = 15/4, offset = 88, length = 10, cover = 0
      sync = 0, usage = display, sign = 0, bwz = 0
      storage = 100, occurances = 10
      Type info:
         Pic = X(10), Cat type = alpha numeric[03],
         usage_allowed = default, length = 10

Fully qualified Symbol: AAAAAAAA.EEEEEEEE(190,2,6)

   EEEEEEEE
      level = 15/4, offset = 88, length = 10, cover = 0
      sync = 0, usage = display, sign = 0, bwz = 0
      storage = 100, occurances = 10
      Type info:
         Pic = X(10), Cat type = alpha numeric[03],
         usage_allowed = default, length = 10
   EEEEEEEE (190,2,6) has offset 38038.

Fully qualified Symbol: AAAAAAAA.GGGGGGGG

   GGGGGGGG
      level = 15/4, offset = 40089, length = 1, cover = 0
      sync = 0, usage = display, sign = 0, bwz = 0
      storage = 1, occurances = 1
      Type info:
         Pic = X, Cat type = alpha numeric[03],
         usage_allowed = default, length = 1

Fully qualified Symbol: AAAAAAAA.EEEEEEEE(200,2,10)
```

```
   EEEEEEEE
      level = 15/4, offset = 88, length = 10, cover = 0
      sync = 0, usage = display, sign = 0, bwz = 0
      storage = 100, occurances = 10
      Type info:
         Pic = X(10), Cat type = alpha numeric[03],
         usage_allowed = default, length = 10
   EEEEEEEE (200,2,10) has offset 40078.

Fully qualified Symbol: AAAAAAAA.WWWWWWWW

   WWWWWWWW
      level = 5/2, offset = 40088, length = 2, cover = 1
      sync = 0, usage = display, sign = 0, bwz = 0
      storage = 2, occurances = 1
      Type info:

Fully qualified Symbol: .
[stephen@nomad symbols]$
```

Notice that after the use of the member name as the qualifying name (`XX`) the name of the 01-level data item becomes available as the qualifying name.

4 Extensions

one important omission needs to be added as soon as possible:

- the `SYNCHRONIZED` clause does not adjust offsets so that the items are aligned on the necessary boundaries.