The Fortran 95 Library μXML

—

A short reference manual

Guy Munhoven
Université de Liège, Belgium
http://www.astro.ulg.ac.be/~munhoven

Abstract

μXML is a Fortran 95 library offering minimalistic support to read and process information from EXtensible Markup Language (XML) files.

μXML is distributed for usage under the GNU Affero General Public License v3.0 (AGPL-3.0).

1 Introduction

Processing simple text files with arbitrary content (i. e., text lines of different and undetermined lengths) is already difficult in Fortran. Support for `CHARACTER` strings is mixed: the intrinsic commands for processing `CHARACTER` strings are quite diverse and powerful. The `CHARACTER` type itself offers, however, only very limited possibilities, at least if standard conforming code is a pre-requisite.

This was one of the first shortcomings to overcome during the development of μXML. There were, once upon a time, various modules available for this purpose, implementing an `ISO_VARYING_STRING` module along the lines of ISO/IEC 1539-2:2000. The reference implementation was known to suffer from memory leaks. Later other implementations were developed, but these now tend to become difficult to find. Currently, such modules can still be found at Rich Townsend's MAD STAR site `http://www.astro.wisc.edu/~townsend/static.php?ref=iso-varying-string`. Here, I decided to develop a new module for representing and handling `CHARACTER` type content of arbitrary undetermined length, tailored to the needs of this project.

The approach adopted for the second stage — reading and processing XML files follows a mixture of the two commonly used methods for processing XML content: SAX (Simple API for XML) and DOM (Document Object Model). Based upon the SAX method, an event based description of a given document is loaded into memory (by the `XMLSTRUCT` subroutine from the `MODMXM_STRUCTLOAD` module–see below). That information is then used by a second subroutine (`XMLLOAD`, from the same module) to load the file's contents and store it in an internal tree, where it can be parsed and processed.

1.1 Getting, Building and Installing μXML

1.1.1 Prerequisites

A Fortran 95 conforming compiler is all that is required. The source code of μXML is written in standard-conforming Fortran 95.

1.1.2 Getting the source code

The source code for μXML can be made available to you in two ways:

1. in a compressed archive (typically a `.tar.gz` file);
2. by custom access to the SVN repository that hosts the source code and allows you to manually download it to your computing platform where all the necessary tools (SVN client, ...) for this access means have been installed beforehand – you will have been informed about the required credential information to use and steps to carry out in this case.

The source code is located under `$MUXML/src`, where `$MUXML` stands for the root path of the library.

1.1.3 Possible customizations

μXML uses a few pre-defined logical unit numbers for I/O purposes. These can, however, be easily changed to avoid conflicts with a main application that would like to take advantage of the XML functionality of μXML. The adopted values are set via Fortran `PARAMETER`s or via pre-processor `#define`'s:

- there are two logical unit numbers declared in `modmxm_general.F90` (`p_unit = 10` and and `p_logunit = 11`);
- the logical unit numbers for `STDERR` and `STDDBG` are defined in `modmxm.h` by `#define MXM_STDERR 0` and `#define MXM_STDDBG 100`, resp.;

modmxm.h furthermore offers extensive debugging options for a fine-grained troubleshooting. Please notice that neither STDERR nor STDDBG are explicitly opened files. On most LINUX systems, e. g., the logical unit 0 is attached to the screen and output directed to unit 100 would then go to a file with the name fort.100 (or with ome other, compiler- or system-dependent name).

1.1.4 Building the library

The building process is based upon the classical make utility.

The default settings in the Makefile are for the GFORTRAN compiler, on a LINUX platform. For other compilers or platforms, the settings may have to be adapted. Once the required customisation has been done, it is sufficient to issue

```
make
```

from a terminal window, while in $MUXML/src.

1.1.5 Installing the library and MOD files

The library archive libmodmxm.a can now be copied or moved to a central location, such as /usr/local/lib. The generated MOD files that are required to USE the various modules included in μXML can also be copied to a central location, such as /usr/local/include/modmxm. Both can nevertheless also be left in place.

2 Library Overview

The μXML library is composed of a series of Fortran 95 modules. In the following sections, we present short summaries of the roles that the different module fulfil both for internal purposes, but also in the API (the Application Programming Interface), i. e., the way the sub-programs provided by μXML must be used to carry out the various tasks required to read in XML files, and how to access and extract specific content provided.

2.1 MODMXM_STRUCTLOAD

The MODMXM_STRUCTLOAD module is the main turntable for reading in content from an XML file. It hosts two FUNCTION sub-programs:

- XMLSTRUCT analyses the file's structure and collects information about its elements' delimitations in a STKXE linked list (see MODMXM_STKXE module description below);

- XMLLOAD uses the information from that STKXE structure to read in the file's contents and distribute them into a STKMX tree (see MODMXM_STKMX module description below).

The STKMX tree contents can then be parsed, interpreted and processed by the main application.

For the analysis of the XML file's structure, XMLSTRUCT follows a last-in–first-out (LIFO) approach of tasks, with support from the MODMXM_TASKS module.

2.2 MODMXM_TASKS

MODMXM_TASKS hosts the task management part of the XMLSTRUCT function sub-program from MODMXM_STRUCTLOAD and is also only used by that subprgram. It defines identifiers and meaningful names for the different tasks that XMLSTRUCT has to perform, such as processing comments, strings, tags and other structural elements of XML files. It furthermore provides the task stack management support (pushing and pulling tasks, ...).

2.3 MODMXM_STKRC

MODMXM_STKRC — STKRC stands for "STacK of Record Chunks" — is a support module to manage and process text content read from files with irregular and no determined line (records) lengths. Content may extend over several lines (file records); EOR characters are detected and tracked. Each chunk has a maximum length of 10 characters by default. This length is nevertheless parametrized and can be changed, but a lower limit of 10 is enforced, as the housekeeping overhead becomes too large with shorter lengths than this.

The module first of all provides TYPE definitions for the required structures, named stack_recchunks. It furthermore includes subroutines to

- create, extend, compact and deallocate part or all of such structures;
- determine their effective lengths and the number of terminated records it contains;
- compare the contents of two such structures against each other or to compare the content of such a structure against a normal character string;
- split up such structures into shorter sub-structures of the same type or into arrays of normal strings.

2.4 MODMXM_GENERAL

MODMXM_GENERAL is closely related to the previous one (MODMXM_STKRC). It provides first of all the central PARAMETER that defines the maximum chunk length used in the character chunk structures of MODMXM_STKRC. It also provides the IOSTAT return values for the Fortran READ command. As these are compiler specific, they get determined by the make process, using the provided CONFIGURE_MODMXM utility (source code in configure_modmxm.F90). Finally, MODMXM_GENERAL includes a subroutine to read one chunk for a stack_recchunks, with additional house-keeping and monitoring.

2.5 MODMXM_STKXE

MODMXM_STKXE and MODMXM_STKMX represent the central parts of μXML. STKXE stands for "STacK of XML Events". *Events* represent any particular feature that can be encountered while parsing an XML file, such as detecting tag opening (<) or closing (>) characters, comments starting (<!--) and closing (-->) sequences, single or double quotes opening or closing string content, attribute name starting and ending positions, etc. MODMXM_STKXE includes a collection of parametric identifiers for the events that are processed. A function sub-program may be used to retrieve meaningful names for the identifiers.

Information read from a file is first submitted to initial consistency checks and then collected in a double-linked list of INTEGER(:) arrays (a list of TYPE(stack_xmlevents), or STKXE list for short). The information recorded for each event documents

- its position in the file, in terms of the record/line number, the order of the chunk (first, second, etc.) in the record and the position of the triggering character in the chunk;
- its type;
- additional characteristics where useful, such as the length for a name, the number of attributes for an element, etc.;
- possibly the number of consecutive occurrences of the same event.

MODMXM_STKXE includes two types of SUBROUTINE sub-programs. The first ones are used to manage STKXE linked lists:

- create and extend them;
- dump their contents;

- gather statistical informations from the registered.

The second ones are used to walk through a `STKXE` linked list to seek for events (an individual one, the next one of several from a list, etc.)

2.6 `MODMXM_STKMX`

While `MODMXM_STKXE` holds the information about the structure and layout of an XML files, `MODMXM_STKMX` provides a means to represents the contents of the file in an organised tree. `STKMX` stands for "STacK of MiniXML" elements. The content of an XML file is stored in a multiply-linked list (a list of `TYPE(stack_minixml)`, or `STKMX` list for short) of complex structures, that holds as diverse information as the type of content (root or other elements with names and attributes, including attribute names and contents; Parsed Character DATA, PCDATA, or unparsed Character DATA, CDATA). `STKMX` lists are multiply linked as each of their nodes may have several children or siblings (siblings have a common parent node). Each node is linked to its parent node, to the first and the the last of its child nodes as well as to the preceding and the following of its sibling nodes. This extensive linking allows for a rather straightforward scanning of the tree.

`MODMXM_STKMX` includes subroutines to

- create, extend and deallocate `STKMX` tree lists;
- print out information
- find and extract information from the tree (e. g., locate element nodes by their name, retrieve attribute indices by their name, attribute contents, data content, etc.)

3 Module reference

In this section, we describe the relevant subroutine and function interfaces that are required to use μXML. We mostly focus on functionality that is required to use the library. Functionality that is of internal usage only will only be shortly summarised.

3.1 `MODULE MODMXM_STRUCTLOAD`

The `MODMXM_STRUCTLOAD` module provides the main commands to read in the contents of an XML file into an organised tree including meta-information on the various types of content. The data transfer from the file to the memory is

carried out in a two-step process. Prerequisites are the usage of the following modules :

- MODMXM_STKXE
- MODMXM_STKMX
- MODMXM_STRUCTLOAD

The call sequence is then

```
1 stkxe_root => XMLSTRUCT(str_xmlfilename [, ...])
2 stkxm_root => XMLLOAD(str_xmlfilename, stkxe_root [, ...])
```

The resulting `stkxm_root` then holds the contents of the XML file and can be parsed to extract the required information.

3.1.1 FUNCTION XMLSTRUCT

The XMLSTRUCT fution uses the following modules:

- MODMXM_GENERAL
- MODMXM_TASKS
- MODMXM_STKXE

The fundamental interface of XMLSTRUCT is as follows:

```
FUNCTION XMLSTRUCT(str_xmlfilename,                          &
                   n_maxlen_eltname,                         &
                   n_maxlen_attname, n_maxlen_attcont) &
RESULT(stkxe_events)

TYPE(stack_xmlevents), POINTER     :: stkxe_events

CHARACTER(LEN=*),      INTENT(IN)  :: str_xmlfilename
INTEGER,               INTENT(OUT) :: n_maxlen_eltname
INTEGER,               INTENT(OUT) :: n_maxlen_attname
INTEGER,               INTENT(OUT) :: n_maxlen_attcont
```

XMLSTRUCT takes as arguments :

- str_xmlfilename, which holds the path to the XML file to process;
- n_maxlen_eltname, an optional argument that can be used to retrieve the maximum length of all the element names detected in the file;

- `n_maxlen_attname`, an optional argument that can be used to retrieve the maximum length of all the attribute names detected in any of the elements the file;
- `n_maxlen_attcont`, an optional argument that can be used to retrieve the maximum length of all the attribute contents found anywhere in the file.

The three optional arguments can be used to make sure that the subsequent transfer of content to size-limited character string variables can be made without loss.

`XMLSTRUCT` returns a `POINTER` to a `TYPE(stack_xmlevents)` list, to be used in a subsequent `XMLLOAD` call.

3.1.2 FUNCTION XMLLOAD

`XMLLOAD` uses the following modules:

- `MODMXM_GENERAL`
- `MODMXM_STKXE`
- `MODMXM_STKRC`
- `MODMXM_STKMX`

The fundamental interface of `XMLLOAD` is as follows:

```
FUNCTION XMLLOAD(str_xmlfilename, stkxe_root, n_maxdepth) &
RESULT(stkmx_root)

TYPE(stack_minixml),   POINTER     :: stkmx_root
CHARACTER(LEN=*),      INTENT(IN)  :: str_xmlfilename
TYPE(stack_xmlevents), POINTER     :: stkxe_root
INTEGER, OPTIONAL,     INTENT(OUT) :: n_maxdepth
```

`XMLLOAD` takes as arguments

- `str_xmlfilename` which holds the path to the XML file to process;
- `stkxe_root`, the `POINTER` to the `TYPE(stack_xmlevents)` list dressed before by `XMLSTRUCT` on the same XML file;
- `n_maxdepth`, an optional argument that can be used to retrieve the largest number of simultaneously opened elements plus one, to allow for possible data it might contain.

XMLLOAD returns a POINTER to a TYPE(stack_minixml) tree, which holds all of the contents of the XML file in an ordered tree. The optional argument can be used allow for a targeted dimensioning of work space during the processing of the tree.

3.2 MODULE MODMXM_STKRC

3.2.1 TYPE definitions

The MODMXM_STKRC module provides the fundamental TYPE definition for the linked list with the chunked representation of character strings of undetermined length.

```
TYPE stack_recchunks
  INTEGER                          :: n_chars
  CHARACTER(LEN=p_maxlen_chunk)    :: str_chunk
  LOGICAL                          :: l_eor
  TYPE(stack_recchunks), POINTER :: prev
  TYPE(stack_recchunks), POINTER :: next
END TYPE

TYPE stkrc_ptr
  TYPE(stack_recchunks), POINTER :: ptr
END TYPE
```

The additional TYPE stkrc_ptr is a container. It encapsulates a POINTER to TYPE(stack_recchunks), making possible the usage of arrays of pointers to TYPE(stack_recchunks) structures.

3.2.2 SUBROUTINE STKRC_createRoot

STKRC_createRoot allocates and initializes a new STKRC list.

```
SUBROUTINE STKRC_createRoot(stkrc_any,        &
                      nlay_stkrc, nlen_stkrc)

TYPE(stack_recchunks), POINTER     :: stkrc_any
INTEGER, OPTIONAL,     INTENT(OUT) :: nlay_stkrc
INTEGER, OPTIONAL,     INTENT(OUT) :: nlen_stkrc
```

The new STKRC list, accessible through the returned POINTER stkrc_any consists of one chunk, left empty, with %n_chars set to 0. Both %prev and %next pointers are initialised to NULL. The optional nlay_stkrc and

nlen_stkrc can be used for monitoring purposes. nlay_stkrc counts the number of chunks in the list, nlen_stkrc the total number of characters. If present, they are initialised to 1 and 0, resp.

3.2.3 SUBROUTINE STKRC_createNext

STKRC_createNext appends one chunk to a given STKRC list.

```
SUBROUTINE STKRC_createNext(stkrc_any,        &
                      nlay_stkrc, nlen_stkrc)

TYPE(stack_recchunks), POINTER       :: stkrc_any
INTEGER, OPTIONAL,     INTENT(INOUT) :: nlay_stkrc
INTEGER, OPTIONAL,     INTENT(INOUT) :: nlen_stkrc
```

The given stkrc_any must point to the tail chunk of an existing list (i.e., stkrc_any must be NULL). If stkrc_any itself is not associated, or if it is already linked to a chunk, STKRC_createNext aborts. If present, the optional nlay_stkrc and nlen_stkrc are incremented by 1 and 0, resp.

3.2.4 SUBROUTINE STKRC_deallocateStkrc

The STKRC_deallocateStkrc deletes the tail of an STKRC list starting at and including a given chunk

```
SUBROUTINE STKRC_deallocateStkrc(stkrc_any,           &
                      nlay_discarded, nlen_discarded)

TYPE(stack_recchunks), POINTER     :: stkrc_any
INTEGER, OPTIONAL,     INTENT(OUT) :: nlay_discarded
INTEGER, OPTIONAL,     INTENT(OUT) :: nlen_discarded
```

If stkrc_any is not associated, STKRC_deallocateStkrc returns silently. All of the list elements following and including the one pointed to by stkrc_any are deallocated; the pointer stkrc_any is nullified upon return. If stkrc_any does not point to the root (the first component) of the list, the preceding element also gets its %next pointer nullified. The optional nlay_discarded and nlen_discarded resp. report the number of chunks removed and the total number of characters they contained.

3.2.5 SUBROUTINE STKRC_compactStkrc

STKRC_compactStkrc compacts and shortens an existing STKRC list.

```
SUBROUTINE STKRC_compactStkrc(stkrc_any)

TYPE(stack_recchunks), POINTER :: stkrc_any
```

STKRC_compactStkrc scans the list starting from stkrc_any. Partially filled chunks that do not have an EOR marker set are filled up with content from the following chunks. Any resulting empty chunks are removed.

3.2.6 SUBROUTINE STKRC_copyStkrcToStr

STKRC_copyStkrcToStr concatenates the chunks of an STKRC list and copies the result into a CHARACTER string.

```
SUBROUTINE STKRC_copyStkrcToStr(stkrc_source,  &
                      str_dest, nlen_returned, &
                      l_pruneateor, l_spc4eor)

TYPE(stack_recchunks), POINTER     :: stkrc_source
CHARACTER(LEN=*),      INTENT(OUT) :: str_dest
INTEGER, OPTIONAL,     INTENT(OUT) :: nlen_returned
LOGICAL, OPTIONAL,     INTENT(IN)  :: l_pruneateor
LOGICAL, OPTIONAL,     INTENT(IN)  :: l_spc4eor
```

By default

- the complete content of stkrc_source is transcribed into str_dest, but transcripion is limited to LEN(str_dest) characters;
- EOR markers are ignored;
- str_dest is filled with space characters (SPC) to the right if necessary.

The default behaviour can be changed by usage of the optional arguments:

- by setting l_pruneateor = .TRUE., the transcription ends at the end of the first STKRC chunk that presents EOR
- by setting l_spc4eor = .TRUE., EOR markers are replaced by SPC
- if present, nlen_returned is set to the actual number of characters transcribed, or to -1, in case str_dest has not enough space to hold all the characters to be transcribed.

3.2.7 SUBROUTINE STKRC_writeStkrc

Similarly to STKRC_copyStkrcToStr, STKRC_writeStkrc concatenates the chunks of a STKRC list, but then prints them out.

```
 SUBROUTINE STKRC_writeStkrc(stkrc_source, i_unit,        &
                      l_pruneateor, l_spc4eor, l_advance, &
                      nlen_returned)

TYPE(stack_recchunks), POINTER     :: stkrc_source
INTEGER, OPTIONAL,     INTENT(IN)  :: i_unit
LOGICAL, OPTIONAL,     INTENT(IN)  :: l_pruneateor
LOGICAL, OPTIONAL,     INTENT(IN)  :: l_spc4eor
LOGICAL, OPTIONAL,     INTENT(IN)  :: l_advance
INTEGER, OPTIONAL,     INTENT(OUT) :: nlen_returned
```

By default,

- the printing destination is the standard output unit (*, STDOUT);
- each EOR marker triggers a linefeed at the end of the marked chunk;
- a linefeed is printed upon completion.

The default behaviour can be changed by usage of the optional arguments:

- if present, i_unit is used as the logical unit for the print destination instead of *;
- by setting l_pruneateor = .TRUE., printing ends with the first STKRC chunk that has an EOR marker;
- by setting l_spc4eor = .TRUE., EOR markers do not trigger new lines, but insertion of a SPC;
- by setting l_advance = .FALSE., no linefeed is printed upon completion;
- if present, nlen_returned is set to the actual number of characters printed (including SPCs that possibly replace EORs.

3.2.8 SUBROUTINE STKRC_dumpStkrc

The subroutine STKRC_dumpStkrc can be used to dump the complete information held by a STKRC list, node by node.

```
 SUBROUTINE STKRC_dumpStkrc(stkrc_source, i_unit)

TYPE(stack_recchunks), POINTER    :: stkrc_source
INTEGER, OPTIONAL,     INTENT(IN) :: i_unit
```

By default, the dump is sent to the standard output unit (*, STDOUT). The default destination can be overridden by using the optional i_unit to define a logical unit number as the destination.

For each chunk, its order in the list is given, the number of characters it holds, its actual content (delimited by >...<, the presence of an EOR marker, and the association status of the linking pointers (%prev and %next).

3.2.9 SUBROUTINE STKRC_lentrim

Similarly to the intrinsic function LEN_TRIM, STKRC_lentrim returns the number of characters held in the concatenated chunks of an STKRC list, after removal of trailing SPCs in the concatenated string.

```
 SUBROUTINE STKRC_lentrim(stkrc_source,           &
                      nlentrim_stkrc, nlen_stkrc)

TYPE(stack_recchunks), POINTER     :: stkrc_source
INTEGER,               INTENT(OUT) :: nlentrim_stkrc
INTEGER, OPTIONAL,     INTENT(OUT) :: nlen_stkrc
```

Upon return, nlentrim_stkrc holds the total number of characters in the STKRC list, where all the trailing SPCs in each chunk has been neglected. If present, the optional nlen_stkrc includes the total number of characters held by the STKRC list, obtained by summing the %n_chars information of all the chunks. Any trailing space is included.

3.2.10 FUNCTION STKRC_STKRC_EQ_STKRC

The STKRC_STKRC_EQ_STKRC function compares the contents of two STKRC lists.

```
LOGICAL FUNCTION STKRC_STKRC_EQ_STKRC(stkrc_1, stkrc_2)

TYPE(stack_recchunks), POINTER :: stkrc_1, stkrc_2
```

The distribution of the content is not considered significant: two lists stkrc_1 and stkrc_2 are assumed to be equal if the sequence of characters that they hold is exactly the same; EOR markers are disregarded. STKRC_STKRC_EQ_STKRC returns .TRUE. if the contents of the two lists are equal in this sense, .FALSE. otherwise.

3.2.11 FUNCTION STKRC_STKRC_EQ_STR

The STKRC_STKRC_EQ_STKRC function compares the content of an STKRC list to that of a CHARACTER string.

```
LOGICAL FUNCTION STKRC_STKRC_EQ_STR(stkrc_1, str_2)

TYPE(stack_recchunks), POINTER    :: stkrc_1
CHARACTER(LEN=*),      INTENT(IN) :: str_2
```

The lists stkrc_1 and the CHARACTER string str_2 are assumed to be equal if the sequence of characters stored in stkrc_1 is exactly the same as that in str_2; EOR markers are disregarded. STKRC_STKRC_EQ_STR returns .TRUE. if the contents of the two lists are equal in this sense, .FALSE. otherwise.

3.2.12 FUNCTION STKRC_createCopyStkrc

STKRC_createCopyStkrc prepares a copy of a given STKRC list.

```
FUNCTION STKRC_createCopyStkrc(stkrc_in) &
RESULT(stkrc_out)

TYPE(stack_recchunks), POINTER :: stkrc_in
TYPE(stack_recchunks), POINTER :: stkrc_out
```

If stkrc_in is not associated, STKRC_createCopyStkrc returns the NULL pointer as a result; else it returns the pointer to a one-to-one copy of stkrc_in.

3.2.13 FUNCTION STKRC_createSplitCopyStkrc

STKRC_createSplitCopyStkrc also produces a copy of given STKRC list. However, unlike STKRC_createCopyStkrc, a new list is started after each chunk that is marked with an EOR and the resulting set of sub-lists is returned as a 1D-array of TYPE(stkrc_ptr) pointer containers.

```
FUNCTION STKRC_createSplitCopyStkrc(stkrc_in) &
RESULT(stkrcp_out)
```

```
TYPE(stack_recchunks),          POINTER :: stkrc_in
TYPE(stkrc_ptr), DIMENSION(:), POINTER :: stkrcp_out
```

Each component of the resulting array thus points to the contents of a single record.

3.2.14 SUBROUTINE STKRC_createSplitCopyStr

Similarly to STKRC_createSplitCopyStkrc, STKRC_createSplitCopyStr produces a copy of an STKRC list split up into single records delimited by the EOR markers. The set of records is, however, not returned as an array of TYPE(stkrc_ptr) pointer containers, but as an array of CHARACTER strings.

```
SUBROUTINE STKRC_createSplitCopyStr(stkrc_in,    &
                      strarr_out, nlen_returned)

TYPE(stack_recchunks),           POINTER     :: stkrc_in
CHARACTER(LEN=*), DIMENSION(:), POINTER     :: strarr_out
INTEGER, OPTIONAL,               INTENT(OUT) :: nlen_returned
```

The lengths of the CHARACTER strings is defined by the calling program unit; the array of CHARACTER strings is allocated here. STKRC_ncountSections (see below) is used to determine the number of records (sections) included in stkrc_in.

3.2.15 FUNCTION STKRC_ncountSections

STKRC_ncountSections returns the number of sections (records) in an STKRC list, delimited by the EOR markers.

```
FUNCTION STKRC_ncountSections(stkrc_in) &
RESULT(n_sections)

TYPE(stack_recchunks), POINTER :: stkrc_in
INTEGER                        :: n_sections
```

3.3 MODULE MODMXM_STKMX

The MODMXM_STKMX provides a derived TYPE (a type of a multiply linked tree structure) to hold the complete content of an XML file and additional meta-data to allow for an efficient scanning and processing. The components of that structure are presented in section 3.3.1. The subroutines presented in sections 3.3.2 to 3.3.5 deal with the setting up, the construction and management of the tree itself (mainly for internal usage in XMLLOAD, e. g.) and only briefly addressed. The subroutines and function presented in sections 3.3.6 to 3.3.14 allow to locate and extract information from an STKMX tree and are thus the important interface sub-programs to get access to the information read in from an XML file.

3.3.1 TYPE definitions

The TYPE stack_minixml definition is cornerstone to the XML processing that can be carried out with μXML. It defines a multi-purpose structure to organise the information included in an XML file in different types (structural information and data content)

```
TYPE stack_minixml

  INTEGER                          :: i_type
                 ! %i_type
                 !   = -1: undetermined
                 !   =  0: root element, name and attributes
                 !   =  1: element name with attributes
                 !   =  2: PCDATA or CDATA of an element
                 !         (implies that
                 !         %n_children  = 0,
                 !         %str_eltname => NULL,
                 !         %str_attname => NULL,
                 !         %mxe_child_a => NULL,
                 !         %mxe_child_z => NULL)
                 !   =  3: PCDATA
                 !   =  4: CDATA

  INTEGER                          :: n_children
                 ! %n_children: only for elements

  INTEGER                          :: n_order
                 ! %n_order: order of the information
```

```
                !  = 0: root element name and its attributes
                !  = 1: sub-elements of the root element or
                !       data in the root element
                !  = 2: sub-elements of order-1-elements

INTEGER, DIMENSION(:), POINTER   :: i_chain
                ! %i_chain: integer array indicating the
                !           information sequence
                ! RANK(i_chain) = %n_order

TYPE(stack_recchunks), POINTER   :: stkrc_eltname
INTEGER                          :: nlen_eltname
                ! %stkrc_eltname: element name
                !                  (only for element nodes)
                ! %nlen_eltname:  element name length
                !                  (only for element nodes)

TYPE(stkrc_ptr), &
  DIMENSION(:), ALLOCATABLE      :: stkrc_attnames
INTEGER, &
  DIMENSION(:), ALLOCATABLE      :: nlen_attnames
                ! %stkrc_attnames: attribute names
                !  (only for element nodes, only
                !   allocated if any attributes present)
                ! %nlen_attnames: attribute name lengths
                !  (only for element nodes, only
                !   allocated if any attributes present)

TYPE(stkrc_ptr), &
  DIMENSION(:), ALLOCATABLE      :: stkrc_attcntts
INTEGER, &
  DIMENSION(:), ALLOCATABLE      :: nlen_attcntts
                ! %stkrc_attnames: attribute contents
                !  (only for element nodes, only
                !   allocated if any attributes present)
                ! %nlen_attnames: attribute content lengths
                !  (only for element nodes, only
                !   allocated if any attributes present)

TYPE(stack_recchunks), POINTER   :: stkrc_data
INTEGER                          :: nlen_data
```

```
                  ! %stkrc_data: data content
                  !  (only for data nodes)
                  ! %nlen_data: data content length
                  !  (only for data nodes)

                  ! pointer to parent element
  TYPE(stack_minixml), POINTER     :: parelt

                  ! pointer to previous sibling
  TYPE(stack_minixml), POINTER     :: prevsib
                  ! pointer to next sibling
  TYPE(stack_minixml), POINTER     :: nextsib
                  ! pointer to first child
  TYPE(stack_minixml), POINTER     :: frstchild
                  ! pointer to last child
  TYPE(stack_minixml), POINTER     :: lastchild

END TYPE



                  ! Encapsulate POINTER to TYPE(stack_minixml)
                  ! in order to use arrays of pointers to
                  ! TYPE(stack_minixml)
TYPE stkmx_ptr
  TYPE(stack_minixml), POINTER :: ptr
END TYPE

TYPE stkmx_ptrlist
  TYPE(stack_minixml), POINTER :: stkmx
  TYPE(stkmx_ptrlist), POINTER :: next
END TYPE
```

The complete content of an XML file is read in and distributed in a tree of multiply linked `TYPE stack_minixml` nodes: the root of the tree is the root element of the XML file. The content of each element is scanned and stored in subsequent child nodes of that element: one node for each sub-element, for each section of PCDATA (Parsable Character DATA) or CDATA (Character DATA), in the order in which they are found in the file. All theses nodes that have a common parent node are called siblings. Notice that only element nodes can have child nodes. Each child node is linked to its parent node (via `%parelt`) and to its preceding and its following sibling, if any (via `%prevsib`

and %nextsib, resp.). Each parent node is also linked to its first and its last child node (via %frstchild and %nextchild. This crosslinking allows for a fast and efficient navigation. Each node may also hold additional information (e. g., number of attributes, names and their respective contents for element nodes, etc.).

3.3.2 SUBROUTINE STKMX_CREATE_ROOT

STKMX_CREATE_ROOT allocates a root node of an STKMX tree and initialised the numerous components with default values.

```
SUBROUTINE STKMX_CREATE_ROOT(stkmx_root)

TYPE(stack_minixml), POINTER ::  stkmx_root
```

Please notice that STKMX_CREATE_ROOT only initiates a new STKMX tree, but does not initialise it as a root element node of an XML tree. The performed initialisation only puts it into a controlled undefined state.

3.3.3 SUBROUTINE STKMX_ADD_CHILD

The STKMX_ADD_CHILD subroutine adds a new child node to an existing element node of an STKMX tree.

```
SUBROUTINE STKMX_ADD_CHILD(stkmx_parelt, l_datachild)

TYPE(stack_minixml), POINTER    :: stkmx_parelt
LOGICAL, OPTIONAL,   INTENT(IN) :: l_datachild
```

The new child node is added as a child node to stkmx_parelt. Upon creation it can be accessed from stkmx_parelt via its %lastchild pointer. By default, the child node is pre-configured as an *element* node unless the optional l_datachild is present and set to .TRUE., in which case a *data* node is pre-configured.

3.3.4 SUBROUTINE STKMX_DEALLOCATE

The STKMX_DEALLOCATE subroutine removes the subtree attached to a given node of an STKMX tree, including that itself; if the node is a data node, only that node itself is removed.

```
SUBROUTINE STKMX_DEALLOCATE(stkmx_any)

TYPE(stack_minixml), POINTER :: stkmx_any
```

All the relevant pointers to and from the node are removed or by-passed. Finally, all the content of the nodes of the sub-tree is removed and the memory deallocated.

3.3.5 SUBROUTINE STKMX_INFO_NODE

STKMX_INFO_NODE prints out summary information about a given node of an STKMX tree.

```
SUBROUTINE STKMX_INFO_NODE(stkmx_node, i_unit)

TYPE(stack_minixml), POINTER    :: stkmx_node
INTEGER, OPTIONAL,   INTENT(IN) :: i_unit
```

By default, the information is printed to the standard output (*, STDOUT). If present, the optional i_unit is used to redirect the output to the logical unit number that it provides.

3.3.6 FUNCTION STKMX_getRootElement

The STKMX_getRootElement can be used to find the root element of the tree that a given node belongs to.

```
FUNCTION STKMX_getRootElement(stkmx_any) &
RESULT(stkmx_rootelt)

TYPE(stack_minixml), POINTER :: stkmx_any
TYPE(stack_minixml), POINTER :: stkmx_rootelt
```

STKMX_getRootElement takes a POINTER to a TYPE(stack_minixml) node as an argument and returns

- a NULL pointer if stkmx_any is not associated;
- a pointer to the root element node of stkmx_any.

3.3.7 FUNCTION STKMX_getElementNodeByName

The STKMX_getElementNodeByName provides a means to locate child element of a given node by their tag names.

```
FUNCTION STKMX_getElementNodeByName(stkmx_any, str_eltname) &
RESULT(stkmx_nodearr)

TYPE(stkmx_ptr), DIMENSION(:), POINTER    :: stkmx_nodearr
```

```
TYPE(stack_minixml),           POINTER    :: stkmx_any
CHARACTER(LEN=*),              INTENT(IN) :: str_eltname
```

As arguments, `STKMX_getElementNodeByName` takes a `TYPE(stack_minixml)` pointer to the parent node to analyse and the name of the element(s) to search for. As an element may have more than one child element with a given name, the result is returned as an pointer to an 1D-array of `TYPE(stkmx_ptr)` pointer containers. `STKMX_getElementNodeByName` returns

- a `NULL` pointer if `stkmx_any` does not have any child nodes at all;
- a `NULL` pointer if `stkmx_any` does not have any child node with that name;
- a 1D-array `stkmx_nodearr`, with as many components as there are child elements named `str_name` for `stkmx_any`, and whose pointer component point to the different children found.

3.3.8 FUNCTION STKMX_getUniqueChildEltByName

`STKMX_getUniqueChildEltByName` also allows to locate a child element of a node by its name. However, unlike Similarly to `STKMX_getElementNodeByName`, `STKMX_getUniqueChildEltByName` looks for unique childs only.

```
FUNCTION STKMX_getUniqueChildEltByName(stkmx_any,    &
                                       str_eltname) &
RESULT(stkmx_resnode)

TYPE(stack_minixml), POINTER    :: stkmx_any
CHARACTER(LEN=*),    INTENT(IN) :: str_eltname
TYPE(stack_minixml), POINTER    :: stkmx_resnode
```

As arguments, `STKMX_getUniqueChildEltByName` takes a pointer to the parent node (`TYPE(stack_minixml)`) to analyse and the name of the child element to search for. It returns a `TYPE(stack_minixml)` pointer to the child node of `stkmx_any` that is named `str_eltname`, if there is only one such child. Otherwise, it

- returns a `NULL` pointer if `stkmx_any` does not have any children;
- returns a `NULL` pointer if `stkmx_any` does not have any child named `str_eltname`;
- triggers an error and aborts if there is more than one child element named `str_eltname`.

3.3.9 FUNCTION STKMX_getChildElementNodes

STKMX_getChildElementNodes can be used to retrieve pointers to all the child nodes of a given STKMX node that are element nodes.

```
FUNCTION STKMX_getChildElementNodes(stkmx_any) &
RESULT(stkmx_nodearr)

TYPE(stack_minixml),           POINTER :: stkmx_any
TYPE(stkmx_ptr), DIMENSION(:), POINTER :: stkmx_nodearr
```

STKMX_getChildElementNodes only takes one argument, stkmx_any, a pointer to the parent node (TYPE(stack_minixml)) to scan. It then returns

- a NULL pointer if stkmx_any does not have any children;
- a NULL pointer if stkmx_any does not have any child nodes that are element nodes;
- a 1D-array of TYPE(stkmx_ptr) containers with pointers to the child nodes of stkmx_any that are element nodes, if there are any of them.

3.3.10 FUNCTION STKMX_getPCDatacntt

STKMX_getPCDatacntt retrieves pointers to the data contents of all PCDATA child nodes of a given STKMX node.

```
FUNCTION STKMX_getPCDatacntt(stkmx_any) &
RESULT(stkrc_datacntt)

TYPE(stack_minixml),           POINTER :: stkmx_any
TYPE(stkrc_ptr), DIMENSION(:), POINTER :: stkrc_datacntt
```

STKMX_getPCDatacntt only takes one argument: stkmx_any, a pointer to the parent node (TYPE(stack_minixml)) to scan. It then returns

- a NULL pointer if stkmx_any does not have any children;
- a NULL pointer if stkmx_any does not have any PCDATA child nodes;
- a 1D-array of TYPE(stkrc_ptr) containers with pointers to the data content of all the PCDATA child nodes of stkmx_any, by order of appearance, if there are any of them.

Notice that, unlike STKMX_getChildElementNodes, STKMX_getPCDatacntt does not provide pointers to the PCDATA child nodes themselves, but directly to their data content, which they store in a STKRC list.

3.3.11 FUNCTION STKMX_getCDatacntt

Similarly to STKMX_getPCDatacntt, STKMX_getCDatacntt retrieves data contents of all CDATA child nodes of a given STKMX node.

```
FUNCTION STKMX_getCDatacntt(stkmx_any) &
RESULT(stkrc_datacntt)

TYPE(stack_minixml),           POINTER :: stkmx_any
TYPE(stkrc_ptr), DIMENSION(:), POINTER :: stkrc_datacntt
```

The results of STKMX_getCDatacntt are completely analogous to those of STKMX_getPCDatacntt, except that they are for CDATA instead of PCDATA.

3.3.12 FUNCTION STKMX_getElementName

The STKMX_getElementName function retrieves the name of an element node that is part of an STKMX tree.

```
FUNCTION STKMX_getElementName(stkmx_any) &
RESULT(stkrc_eltname)

TYPE(stack_minixml),   POINTER :: stkmx_any
TYPE(stack_recchunks), POINTER :: stkrc_eltname
```

As an argument, STKMX_getElementName takes stkmx_any, a pointer to the node for which the name information is requested. It

- triggers an error and aborts if stkmx_any is not associated;
- returns the name of the XML element that the node stkmx_any relates to, if any;
- returns a NULL pointer if stkmx_any does not point to an element node (i. e., if it points to a PCDATA or a CDATA node, or if it is onlly pre-initialised).

3.3.13 FUNCTION STKMX_getAttIdxByName

STKMX_getAttIdxByName returns the index of an attribute stored in an element node of an STKMX tree.

```
INTEGER FUNCTION STKMX_getAttIdxByName(stkmx_node, &
                      str_attname)
```

```
TYPE(stack_minixml), POINTER    :: stkmx_node
CHARACTER(LEN=*),    INTENT(IN) :: str_attname
```

As arguments, `STKMX_getAttIdxByName` takes a `TYPE(stack_minixml)` pointer to the element node to analyse and the name of the attribute to search for. It then returns

- -1 if `stkmx_node` is not an element node (only element nodes can have attributes);
- 0 if `stkmx_node` is an element node, but does not have any attribute named `str_attname` (or no attributes at all);
- the index of the attribute named `str_attname` in the attribute arrays linked to the node if it exists.

3.3.14 FUNCTION STKMX_getAttcnttByIdx

Once the index of an attribute is known, `STKMX_getAttcnttByIdx` can be used to retrieve its contents.

```
FUNCTION STKMX_getAttcnttByIdx(stkmx_node, i_att) &
RESULT(stkrc_attcntt)

TYPE(stack_recchunks), POINTER    :: stkrc_attcntt
TYPE(stack_minixml),   POINTER    :: stkmx_node
INTEGER,               INTENT(IN) :: i_att
```

As arguments, `STKMX_getAttcnttByIdx` takes a `TYPE(stack_minixml)` pointer to the element node whose attributes are requested, and the index of the attribute whose content is required. It returns

- a `NULL` pointer if `stkmx_node` does not point to an element node;
- a `NULL` pointer if `stkmx_node` points to an element node, but does not have any attribute with index `i_att`;
- a pointer of `TYPE(stack_recchunks)` (an `STKRC` list) to the content of attribute `i_att` of the node pointed to by `stkmx_node`.

3.4 Others

3.4.1 PROGRAM CONFIGURE_MODMXM

This small standalone program (source code in `configure_modmxm.F90`) is used during the `make` process to generate the `modmxm_general.h` header file

that is #include'd in modmxm_general.F90. configure_modmxm determines the IOSTAT status values that the Fortran READ command returns upon successful reading, encountering an end-of-record (EOR) marker, and end-of-file (EOF) marker, or when trying to continue to read even past an EOF marker. The obtained status values are used to set up PARAMETER declaration statements that are then written to modmxm_general.h.

3.4.2 MODULE MODMXM_GENERAL

The MODMXM_GENERAL module centralises file related parameter values (logical unit numbers for common usage of all of the μXML related subroutines and functions; IOSTAT return values from READ commands). MODMXM_GENERAL also holds the central PARAMETER that defines the chunk length of the STKRC chunks.

Finally, MODMXM_GENERAL contains a subroutine called READ_NEXTCHUNK that is extensively used in the XMLSTRUCT and XMLLOAD subroutines in the MODMXM_STRUCTLOAD module.

3.4.3 MODULE MODMXM_STKXE

The MODMXM_STKXE module contains nine sub-programs (functions and subroutines), that are, however, only useful within the XMLSTRUCT and XMLLOAD subroutines from the MODMXM_STRUCTLOAD module. For common usage, is only necessary to use for the TYPE stack_xmlevents definition required to call these two subroutines :

```
INTEGER, PARAMETER                :: p_ndescs = 6

TYPE stack_xmlevents
  INTEGER, DIMENSION(p_ndescs)    :: idesc
  TYPE(stack_xmlevents), POINTER  :: prev
  TYPE(stack_xmlevents), POINTER  :: next
END TYPE
```

3.4.4 MODULE MODMXM_TASKS

The MODMXM_TASKS module is only used by the XMLSTRUCT subroutine from the MODMXM_STRUCTLOAD module. It provides the task management system for that subroutine.

4 Limitations

The functionality of μXML remains fairly basic at this stage. Any standard conforming XML document can be reliably read in. The processing is currently limited by the fact that only the most basic character encoding (`us-ascii`) is supported.[1] Standard conforming comments are correctly recognized and filtered out; syntax and nesting errors are detected. Attribute content can be delimited by simple or by double quotes. CDATA and PCDATA sections are recognized. Entity processing is, however, not yet supported (as of SVN revision 38 of μXML), not even for the five predefined entities `&quot;` ("), `&apos;` ('), `&` (&), `<` (<) and `>` (>). It is nevertheless planned to introduce the processing of these five in the future. For the common usage of XML that is currently made of μXML (code generation of the sediment model MEDUSA), not even these are indispensable. Entity processing in XML is unfortunately a recursive process requiring a lot of overhead.

Finally, μXML does not provide any XML writing functionality, although this should be rather straightforward to implement (if visual formatting is deemed of secondary importance).

[1] This limitation is to some extent due to the limited support of more extensive character sets by Fortran 95. Actually, although Fortran 2003 now provides the intrinsic commands (such as `SELECTED_CHAR_KIND()`) for managing international character sets and even reserves specific keywords (such as `'ISO_10646'`), compilers are not required to support any character sets besides the `'DEFAULT'` one. Even the Fortran 2008 standard does not yet require this.