Instructions for Preparing the MPI Standard Document

Message Passing Interface Forum

March 15, 2014

1 Introduction

This document provides guidance on editing the MPI standard documents. The MPI standard uses LaTeX, a powerful markup language where items are marked based on the content, rather than low-level control of individual formatting options as in WYSIWYG (what you see is what you get) markup. LaTeX is a high level interface over TeX, a powerful and general purpose typesetting language. LaTeX permits the use of TeX, which gives it extra power but also makes it easy to introduce the sort of inconsistent formatting that is the bane of documents produced with WYSIWYG systems.

Sections 3 covers use of markup in the document. This includes formatting the text and, in Section 3.8, markup used to automate testing of code examples. Section 5 covers how to create the document. Section 6 covers the process to be followed in making changes to the document.

2 Accessing the Document

The MPI standard documents are maintained in an svn repository. To checkout the approved version of the MPI 3.1 standard, for example, use

```
svn co https://svn.mpi-forum.org/svn/mpi-forum-docs/approved/MPI-3.1
```

For a read-only copy, replace the `co` (`checkout`) command with `export`. Note that edit access to the standard is restricted to Forum members who have been given permission to update the document; typically, the chapter authors and document editor have access.

For working groups who would like to use the svn repository for their work, there is an area set aside for the working groups. For example, for

a chapter committee that is working on revisions to MPI 3.1, the following command (with `<chapter-name>` replaced with, for example, `one-sided` or `point-to-point`) will create a new repository:

```
svn copy \
  https://svn.mpi-forum.org/svn/mpi-forum-docs/approved/MPI-3.1 \
  https://svn.mpi-forum.org/svn/mpi-forum-docs/trunk/working-groups/mpi-3.1/<chapter-name>
```

Once this new repository is created, it can be checked out as usual.

need to provide instructions for adding read and read/write access

3 Formatting the Document

For compatibility with the widest variety of editors, text should be wrapped to fit with 80 columns. Edits should avoid reflowing text as this complicates identifying real changes in the document.

3.1 Basic Formatting

For the most part, text should not use TeX (or LaTeX) formatting commands. In particular, font or size changes should not be used. Formatting of MPI names and C and Fortran code is handled with the macros defined below. You may use `\emph` to *emphasize* text, as in `\emph{emphasize}`. The command `\mpicode` may be used for miscellaneous language-independent code for which there are no appropriate MPI macros. The command `\code` should be used for code in a specific language, such as C or Fortran. Note that most uses of text font selection, including "face" (such as bold or sans serif), are erroneous. That is, you should use the appropriate markup for the kind of content, rather than change the font. The commands `\textbf` and `\textsf` should almost never be used. See Section 3.3 for the LaTeX commands to be used with different MPI objects.

Spaces *are significant* in LaTeX. Attempting to make the LaTeX source look more like a C program (some MPI document authors have done that) introduces additional spacing that is jaring to the eye and can interfere with tools used to find changes in the document. See Section 7 for some examples.

LaTeX defines many environments and many others may be added to LaTeX. To preserve a uniform appearance, use only these environments or the ones specifically defined for the MPI standard in Section 3.2. Most of these are well-known; where there is some subtle feature, this is noted (e.g., between "center" and "centering").

array

center This environment should be used to center tables and text outside of figures.

centering This environment should be used to center figures (it does not add additional space around the figure).

description

displaymath

enumerate

eqnarray

example This environment should be used for example program fragments

figure Note that captions (with the `\caption` command) go below figures.

itemize

obeylines

tabbing

tabular

table Note that captions (with the `\caption` command) go *above* tables (tables themselves are created with the `tabular` environment).

verbatim

Verbatim An alternative form of verbatim that allows the use of TeX commands within the Verbatim environment. Note that this should *not* be used for code examples, as it interferes with the automated checks that the code examples compile.

To include a graphic image, use `\includegraphics`. This command can rescale the size of the image. A typical use (from the Collectives chapter) is

```
\includegraphics[width=4in]{figures/mycoll-fig2}
```

Figures should be provided in PDF (`pdf`) formats. Figures should be placed in the `figures` directory, not in the chapter's directory.

Explicit linebreaks should be avoided unless they are used where a linebreak is always required, such as at the end of a line of declarations. Use `\gb` (for "good break") to tell LaTeX where it may be better to break a

line. The reason for using this is that if subsequent edits to the text change the flow of the paragraph, the line breaks will be adjusted accordingly. If `\gb` doesn't work (it uses TeX linebreak penalties to suggest a good spot at which to break the line, but does not force it), try `\vlgb`. This is a version of `\gb` that may add a larger amount of whitespace to the line, making a break there more likely.

References to section, table, figure, or enumerated list items should not use the number that appears in the document. Instead, they should make use of the LaTeX command `\label` and `\sectionref` or `\ref`. The `\label` command creates a symbolic label for the most recent command that creates a number. For example,

```
\section{Communication Calls}
\label{sec:onesided-putget}
```

creates the label `sec:onesided-putget`. This section can then be refered to with the `\sectionref` command:

```
\sectionref{sec:onesided-putget} describes the ...
```

The command `\sectionref` will output the text "Section" followed by the section number, including the TeX commands to avoid splitting the section name and number across a line. The `\label` command may also be used after a `\caption` command to get the number of a figure or table, and after the `\item` command in an enumerated list to get the number of that item. This approach is used in the One-Sided Communications chapter to refer to the different RMA rules. Using the `\label` and `\sectionref` or `\ref` commands ensure that the references remain correct even if a new numbered item is inserted into the document.

When referencing another part of the document, refer to a section. For example, use

```
Consider the code fragment in Example~\ref{ex:1sided-fence}.
```

Previous versions of the standard sometimes also provided the page number; while that is somewhat helpful for printed copies, the section information is adequate and many users will use on-line versions, where the section reference will provide a direct link to the relevant page. Rather than use inconsistent style, the text standard is to only use the Section number. Do not use the section name either.

3.2 MPI Environments

There are several environments that are used for special comments to the reader (advice to user, implementors, and rationale) and for lists of constants and functions.

users Advice for user

rationale Rationale for a choice in the MPI standard.

implementors Advice for implementers

constlist A list of MPI constants.

funcdef The environment used for many MPI function definitions. There are related environments `funcdef2` (to force a line break in the argument list between the first and second arguments) and `funddefna` (for functions with no arguments). `funcdef` takes one argument, which is the language-independent declaration (complete with arguments), followed by one or more `\funcarg` arguments that define each parameter.

mpicodeblock Used for language-independent code examples

3.3 MPI Objects

These macros are used to mark MPI objects in the code. In some cases, they provide automatic indexing for the names; all of these are rendered in a consistent font.

`\mpifunc` Used to refer to MPI functions in normal text, when the abstract, language independent function is meant. Because the name is language independent, it should be in uppercase. Example use: `\mpifunc{MPI\_BCAST}`.

The name `\func` is a deprecated alias. Do not use it and consider replacing it with `\mpifunc`.

`\cfunc` Like `\mpifunc`, but for C functions.

`\ffunc` Like `\mpifunc`, but for Fortran functions.

`\mpiarg` Use this to refer to (language independent) function arguments in normal text. E.g.: `\mpiarg{array\_of\_handles}`.

`\carg` Like `\mpiarg` but for C functions

`\farg` Like `\mpiarg` but for Fortran functions.

`\type` Use this to refer to the type of (language independent) function arguments in normal text. E.g.: `\type{MPI\_INTEGER}`

`\ctype` Just like `\type`, but for C types. E.g.: `\ctype{double}`

`\ftype` Just like `\type`, but for Fortran types. E.g.: `\ftype{INTEGER}`

`\const` Use this to refer to mpi-defined constants, in a language independent way. Takes one argument, the name of the constant.
E.g.: `\const{MPI\_MINLOC}`

`\constskip` Like `\const` but does not put the name in the index.

Obsolete commands which may be encountered but should not be used include

`\mpifunci` Like `\mpifunc`, but meant to be used with functions defined in MPI-1.

`\func` Has the same effect as `\mpifunc`.

Ocassionally, the standard uses a shorthand to describe a number of similar functions, as in MPI_FILE_IXXX. Use the command `\XXX/` to indicate the XXX as in

```
\mpifunc{MPI\_FILE\_I\XXX/}
```

this ensures that a consistent style is used in the document.

3.4 Standard Names

The following ensure that the name "MPI" is in the proper font and size. The `/` that follows the name is used to ensure that spaces are preserved (otherwise, TeX removes the blanks after a TeX command from the output).

`\MPI/` **or** `\mpi/` Gives you MPI with the correct font. It is used to refer to MPI in general. Example usage:
`It is highly desirable that \MPI/ not use...`

`\MPIIII/` **or** `\mpiiii/` Use like `\MPI/` but when you want to specifically refer to MPI-3.

`\MPIIIIDOTI/` **or** `\mpiiiidoti/` Use like `\MPI/` but when you want to refer to MPI-3.1.

`\MPIIIIDOTO/` **or** `\mpiiiidoto/` Use like `\MPI/` but when you want to refer to MPI-3.0.

`\MPIII/` **or** `\mpiii/` Use like `\MPI/` but when you want to refer to MPI-2.

`\MPIIDOTO/` **or** `\mpiidoto/` Use like `\MPI/` but when you want to refer to MPI-1.0.

`\MPIIDOTI/` **or** `\mpiidoti/` Use like `\MPI/` but when you want to refer to MPI-1.1.

`\MPIIDOTII/` **or** `\mpiidotii/` Use like `\MPI/` but when you want to refer to MPI-1.2.

`\MPIJOD/` **or** `\mpijod/` Use like `\MPI/` but when you want to refer to MPI-JOD.

`\MPIRT/` Use like `\MPI/` but when you want to refer to MPI/RT (real time).

3.5 Defining MPI Functions

The environment `funcdef` (also mentioned about in the MPI environments) is used to define the language-independent form of an MPI function. `\begin{funcdef}` takes one additional argument, like this:

```
\begin{funcdef}{MPI\_FOO( bd\_handle, root, comm )}
```

This is then followed by one more more `\funcarg` commands. `\funcarg` takes three arguments: intent of the argument, the name of the argument, and a brief description of the argument. For example

```
\funcarg{\IN}{root}{rank of broadcast root}
```

The MPI notion of IN, OUT, and INOUT arguments is slightly different that in some programming language descriptions. See the description in Section 2.3 (Procedure Specification) of the MPI Standard for more details, but roughly, they are

`\IN` Argument (or the object to which the argument refers) is not changed.

`\OUT` Argument (or the object to which the argument refers) is an output result only.

`\INOUT` Argument (or the object to which the argument refers) is both input and output.

A complete example, simplified from that of `MPI_BCAST`, is

```
\begin{funcdef}{MPI\_FOO( bd\_handle, root, comm )}
\funcarg{\INOUT}{bd\_handle}{Handle to buffer descriptor.
   On root, this is the send buffer descriptor, elsewhere,
   this is the receive buffer descriptor.}
\funcarg{\IN}{root}{rank of broadcast root}
\funcarg{\IN}{comm}{communicator handle}
\end{funcdef}
```

3.6 Bindings

Bindings specify how an MPI operation or object is expressed in a particular programming language. These are best updated by following an example with a similar format.

3.7 Indexing

The MPI standard has five separate indices:

- Examples Index
- Constant and predefined Handle Index
- Declarations
- Callback Functions
- Functions

There is no separate "concept" index, though the examples index contains references to some important MPI concepts.

For the indices for constants, declarations, callbacks, and functions, most (if not all) of the entries are created by using the correct macros around the names:

Constants

`\const` MPI constants

`\type` MPI datatype handles

`\shorttype` Like `\type`, but does not include the TeX macro for a "good break". Use only if `\type` introduces excessive spacing.

`\error` MPI Error classes. Use `\errormain` for the definition of an error class.

`\info` Predefined MPI Info string

`\infokey` MPI Info key

`\infoval` MPI Info value

An obsolete command, `\consti`, was occassionally used for constants defined in MPI-1.

Declarations `\typedefindex` adds an MPI callback function `typedef` to the index.

Callbacks

`\mpitypedefbind` Typedef for an MPI callback that returns an `int`.

`\mpitypedefbindvoid` Typedef for an MPI callback that returns a `void`.

`\mpitypedefemptybind` Typedef for an MPI callback; the second argument is the return type.

Functions Environments that define functions (`funcdef`, `funcdef2`, `funcdefna`). `\mpiemptybindidx` is for C binding definitions for functions that do not return an `int` and need to be in the index.

In places where the above macros cannot be used (such as within verbatim environments, for examples, or in certain tables), the index entries may be added explicitly with the following macros (which add *only* the index entry, and do not add any text at the point where they are used):

Constants `\cdeclindex` and `\cdeclmanindex` for C language declarations such as MPI_Comm.

Callbacks `\typedefindex` for C typedefs (function callbacks)

Functions `\mpifuncindex` and `\mpifuncmainindex` for MPI function.

The command contains the word "main" if this is the location where the name being indexed is defined; it will be underlined in the index.

(The callbacks are separated from the declarations when the indices are created).

For the examples index, entries must be added explicitly with `\exindex`. This should normally start in the first column and have a TeX comment character immediately after it to prevent blanks from entering the document (which can mess up the formatting). For example,

```
\exindex{MPI\_SEND}%
```

For entries that have subitems, e.g., you want the index to look like

```
foo
   bar     27
   foobar  721
```

use an exclamation point, as in

```
\exindex{foo!bar}%
...
\exindex{foo!foobar}%
```

In most cases, you should not use either a hyphen or a comma in an index entry — instead, use the "!" form.

Many items should also have a primary index entry, particularly when the item has many index entries. The macros in `mpi-macs.tex` contain macros `\mpifuncmainindex` and `\cdelcmainindex` which do this. You can make an item a "primary" by adding `|uu` to the entry, as in

```
\index{foo|uu}
```

Do not use text formatting commands in index entries. There is a way to do that, but it must be done with some special commands (otherwise the sorting of the index entries may be done by the formatting command rather than the text of the item, which has happened to at least one publisher).

The best way to check the index entries for each chapter is to build the chapter separately (execute `make` in the chapter's directory) and check the index that is created for that chapter to ensure that all relevant entries are included. Inspection of the LaTeX file can also help, in particular where verbatim environments are used. A final check of the full index, looking for duplicates, misspelled routines, and missing entries, can help identify other problems. Specifically, check for the following:

- Each function defined in the chapter appears in the index and is marked as the main entry (page number is underlined).
- Each constant (including MPI handles) defined in the chapter appears in the index and is marked as the main entry.
- Each function used in an example appears in the examples index.

3.8 Code Examples

To reduce the number of errors in code examples, we use a tool that extracts code from the document and attempts to compile the code. Below is a code example (slightly modified) from the Point-to-point chapter.

```
\begin{example}
\label{pt2pt-exA}
\exindex{MPI\_SEND}%
\exindex{MPI\_RECV}%
\exindex{Datatypes!matching}%
Sender and receiver specify matching types.
%%HEADER
%%LANG: FORTRAN
%%FRAGMENT
%%DECL: integer comm, rank, ierr, tag, a(2), b(2)
%%DECL: integer status(MPI_STATUS_SIZE)
%%ENDHEADER

\begin{verbatim}

CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
  CALL MPI_SEND(a, 10, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
  CALL MPI_RECV(b, 15, MPI_REAL, 0, tag, comm, status, ierr)
END IF

\end{verbatim}
```

The `\exindex` command adds its argument to the index of examples. MPI routing names should use the language-independent (all uppercase) name format. TeX comments between `%%HEADER` and `%%ENDHEADER` are used to provide extra information needed to compile the code, such as specifying

the language (e.g., `%%LANG: C` or `%%LANG: FORTRAN`), declarations for variables (e.g., `%%DECL: int a;`), and whether the code is a complete routine or a fragment (with `%%FRAGMENT`). Look at other examples in the text or the file `README` in directory `mpicompilechk`.

To check the code examples in the MPI document, run `make check`.

3.9 Language-Independent Examples

In some cases, you need to show language-independent code for MPI. That is, the code is neither C nor Fortran. An example is in the decsription of some of the collective routines, where language-independent code is used to define the behavior (in terms of data moved) of the a collective routine in terms of point-to-point routines. Use the `mpicodeblock` environment.

3.10 Mathematics

TeX was originally designed to typeset mathematics and no system is as effective at correctly handling all of the requirements of mathematical typesetting. You may use any of the usual LaTeX or TeX mathematics formatting commands when typesetting mathematics (most of the mathematical formulas are in the Datatype sections). If you have specific questions, either consult any of the LaTeX documentation or the document master.

4 Interfacing with PDF

The MPI document is created as a PDF file. The use of the standard LaTeX macros automatically generates links within the document that permit quick navigation. However, the use of LaTeX macros within these macros can create problems. For example, using a macro within the section name will cause problems for the PDF link. The solution is to use the special macro, `\texorpdfstring`. This macro takes two arguments. The first is used in producing the document output; the second must have no LaTeX commands and is used in the PDF link. For example, instead of this:

```
\section{Embedding in \MPI/}
```

use this (taken from `topol.tex`):

```
\section{Embedding in \texorpdfstring{\MPI/}{MPI}}
```

In some cases, because of limitations in the way LaTeX constructs the tables of contents and list of figures, it may be necessary to ensure that spaces are

not eliminated. An easy way to protect a space is to put `\bhox{}` before or after the space, depending on what is needed. This approach has been used in a few updates to section titles.

5 Building the Standard

To build the standard, simply use `make`. There are additional targets that may be used to build special versions of the standard.

clean Clean the directory tree of auxiliary files created by running make.

veryclean Like clean, but cleans more created files, including converted graphics files.

distclean Like veryclean, but also removes the document PDF files.

check Runs the utility to check the code examples. This utility must have been configured using the `configure` command in the directory `mpicompilechk`.

eachchap Builds each chapter separately. This is good for editing individual chapters, particularly for index entries and bibliographic references.

cleandoc Produce a clean version of the document with no markup about the changes in the standard (no change in font color, no changebars, not old/new text).

cleanbwdoc Like `cleandoc`, but in black and white only.

bookprintingdoc Almost the official version, but define the TeX `\bookprintingtrue`.

allversions Use this to build all of the versions of the document that are made available

HTMLVERSION Create an HTML version of the standard. This uses the tool `tohtml`, which must be installed from `ftp.mcs.anl.gov/pub/sowing`. The better-known `latex2html` is (at last test) unable to handle a document of the size of the MPI standard.

When this target is used, check the file `latex.err` to see what problems were encountered in creating the HTML version.

BWHTMLVERSION Like `HTMLVERSION`, but in black and white only.

5.1 Building Individual Chapters

To build a single chapter, first build the full standard. This will provide the information necessary to include the proper values for references to sections in other chapters, and the correct chapter number. Then `cd` to the directory of the chapter and use `make`.

5.2 Build Configuration

There are a number of options for the build of the document that are controled by a configuration file. Most users need never deal with the configuration, as the `Makefile` for the document handles all configuration file settings for building each type of document.

The builds copy predefined configuration files, stored in the `maint` directory, to the file `mpi-report.cfg`. Builds of individual chapters and explicit builds of the report use whatever `mpi-report.cfg` file is present in the directory; builds of specific versions of the document (e.g., `make cleandoc`) replace the current `mpi-report.cfg` file with the appropriate choice from the `maint` directory.

6 Editing Process

The MPI Document is developed by chapter subcommittees. Each chapter has a chair and a committee of at least 4 (including the chair). The committee is responsible for editing their chapter. For MPI-3.1, the MPI Forum has simplified the process of editing the official document, relying now on automated tools to compare different versions of the document rather than requiring all changes to be marked up with special macros defined for the purpose. However, these macros are still available for use by chapter committees and may be used by replacing the command `\allowchangefalse` with `\allowchangetrue` from the `mpi-report.cfg` file before building a file that makes use of these change macros (see Section 5.2 for more on the configuration files).

Changes may be introduced in two ways. The first is with the ticket system; this system is akin to a bug report system against the document. These can range from minor errors such as misspellings to the introduction of new routines. A ticket provides very specific details about the change, including the specific locations where changes are to be made.

The second way to introduce a change is for the chapter subcommittee to edit the chapter directly. This is appropriate for more extensive changes,

which may range from thorough spelling and grammar corrections to introduction of significant new capability through new functions. The chapter committee may choose to have the MPI Forum vote on parts of this process separately, for example, a new function may be brought before the Forum for a vote. However, a final decision is based on a vote for the chapter as a whole (as well as a vote on the standard as a whole).

Note that all changes must be approved with the MPI Forum's process. At this writing, this requires a reading of the chapter, followed by two successful (majority) votes, each reading and vote held during a different meeting (this is intended to give adequate time for reflection and review).

This process differs from the MPI-2.1 and MPI-2.2 process because those versions were intended as minor edits to the standard. The process described above follows the process used for MPI-1 and MPI-2, where chapters were written by subcommittee, with frequent feedback from the MPI Forum in terms of straw votes or Forum votes on particular features, but without Forum votes for each individual change.

6.1 Update Macros

The macros in this section may be used to mark changes in the document during development of material. They must *not* be used in the final document; our experience has been that their extensive use is error prone and focuses attention on small changes rather than the broader context, leading to errors in the final document. However, they can be useful during the development of new material and are provided for that use. In addition, tools such as `latexdiff` may be used to create a version that highlights the differences between versions of the MPI standard document.

The most general form is this macro:

```
\MPIupdateBegin{3.1}{ticket-number}
... changed text
\MPIupdateEnd{3.1}
```

A short form,

```
\MPIupdate{3.1}{ticket-number}{update}
```

may be used for very short changes, and

```
\MPIreplace{3.1}{ticket-number}{old text}{new text}
```

for replacements of text.

Deletions should be marked (where possible) with

```
\MPIdeleteBegin{3.1}{ticket-number}
% ... deleted text, commented out in LaTeX
% % ... comment out comments as well
\MPIdeleteEnd{3.1}
```

Note that the text to be deleted is commented out using the TeX comment symbol. This is a pragmatic choice — it is possible to force TeX to ignore blocks of text, but if those blocks of text contain certain TeX or LaTeX commands, the deletion may not work properly. This approach, while less elegant, is more certain.

A short form,

```
\MPIdelete{3.1}{ticket-number}{text to delete}
```

may be used for short deletions.

If the `\MPIdelete...` form cannot be used, then removed or replaced text should be commented out if at all possible

While a chapter is being developed by a chapter committee, that committee may choose to mark updates, changes, or alternatives in other ways. This is acceptable as long as when the chapter is presented to the MPI Forum, the update macros described above are used.

Updates to examples are awkward to mark, and tend to make it harder, not easier, to read and verify the code. If the chapter committees are functioning properly, and the automated code checker is used, a better tradeoff is (as is often the case) for readability and to correct the code, adding a LaTeX comment if necessary to mark the change. More details about changes are always available through the svn logs.

7 Things Not To Do

This section provides some examples of what *not* to do in the MPI document, with examples of the correct approach.

7.1 Spaces

Remember, spaces are significant. Do not try to make the LaTeX source look like good C code. Here is an example of incorrect use drawn from the document:

```
\caption{
Here is my caption
}
```

The newlines at the end of the first two lines add additional space into the caption. Instead use

```
\caption{Here is my caption}
```

An alternate correct approach is to tell LaTeX to ignore the newlines by using the LaTeX comment character:

```
\caption{%
Here is my caption%
}
```

Leaving off the second comment character (at the end of the caption) is *incorrect*.

7.2 Font Changes

Avoid font changes as much as possible. Use the correct commands when you do need to make them. For example, to *emphasize* a word in a sentance by changing font style, use `\emph{word}` rather than `{\em word\/}` or the incorrect `{\em word}` (the latter case is incorrect because it fails to include the *italic correction* — a TeX command needed to ensure that the spacing between the end of the italic text and the next non-italic character is not too large).

Incorrect	Correct
`{\tt text}`	`\texttt{text}`
`{\bf text}`	`\textbf{text}`
`{\em text}`	`\emph{text}`
`{\sf text}`	`\textsf{text}`

Note that in many cases, the use of `\texttt` or `\textsf` is still incorrect, because the commands that are defined for various MPI terms or parameters should be used instead.

7.3 Spacing

Do not add vertical spacing to the document with the low-level spacing commands such as `\vskip` or `\vspace`. It is almost always wrong to add explicit spacing with these commands (this is akin to using inlined assembly instructions when programming in a higher level language). In the rare case where additional vertical space is necessary, use one of the predefined skip commands: `\smallskip`, `\medskip`, or `\bigskip`.

7.4 Dashes

There are three major dashes:

name	appearance
hyphen	-
en dash	–
em dash	—

An *en dash* is used between two numbers, as in 27–32. An *em dash* is used as punctuation in a sentance — as used here. It is incorrect to use an en dash as punctuation, and it is incorrect to use a hyphen in a number range.

7.5 And so on

The above are not the only things to avoid — the recommendation is to stick to the commands outline in this document and to contact the document master/editor if something else is needed.