Joaquín Ataz-López

Une courte (?) introduction à ConTeXt Mark IV

Une courte (?) introduction à ConTEXt Mark IV
Version 1.6 [2 janvier 2021]

Titre original: Una introducción (no demasiado breve) a ConTEXt Mark IV
Traduction française: A bon ami qui souhaite rester anonyme.

Historique des versions :

- 18 août 2020 : Version 1.0 (Uniquement en espagnol) : Document original.
- 23 août 2020 : Version 1.1 (Uniquement en espagnol) : Correction de petites erreurs de frappe et de malentendus de l'auteur.
- 3 septembre 2020 : Version 1.15 (Uniquement en espagnol) : Autres corrections de petites erreurs de frappe et de malentendus.
- 5 septembre 2020 : Version 1.16 (Uniquement en espagnol) : Autres corrections de petites erreurs de frappe et de malentendus ainsi que des petites modifications qui améliorent la clarté du texte (je crois).
- 6 septembre 2020 : Version 1.17 (Uniquement en espagnol) : C'est incroyable le nombre de petites erreurs que je trouve. Si je veux m'arrêter, je dois arrêter de relire le document.
- 21 octobre 2020 : Version 1.5 (Uniquement en espagnol) : Introduction de suggestions et correction des erreurs signalées par les utilisateurs de la liste de diffusion ntg-context.
- 2 janvier 2021: Version 1.6 : Corrections suggérées après une nouvelle lecture du document, à l'occasion de sa traduction en anglais

Table des matières

Préface 5

I What is ConTEXt and how do we work with it 14

1 ConTEXt: a general overview 15

1.1 What is ConTEXt then? 15
1.2 Typesetting texts 16
1.3 Markup languages 18
1.4 TEX and its derivatives 19
1.5 ConTEXt 23

II Global aspects of the document 32

2 Pages and document pagination 33

2.1 Page size 33
2.2 Elements on the page 38
2.3 Page layout (`\setuplayout`) 41
2.4 Page numbering 46
2.5 Forced or suggested page breaks 48
2.6 Headers and footers 50
2.7 Inserting text elements in page edges and margins 54

III Particular issues 57

3 Characters, words, text and horizontal space 58

3.1 Getting characters not normally accessible from the keyboard 58
3.2 Special character formats 68
3.3 Character and word spacing 72
3.4 Compound words 76
3.5 The language of the text 77

Appendices 86

A Installing, configuring and updating ConTEXt 87
1 Installing and configuring « ConTEXt Standalone » 88
2 Installing LMTX 93
3 Using several versions of ConTEXt on the same system (only for Unix-type systems) 97

Préface*

Noble lecteur, voici un document à propos de ConTEXt un système de composition de document dérivé de TEX, qui, à son tour, est un système de composition créé entre 1977 et 1982 par Donald E. Knuth à l'Université de Stanford.

ConTEXt a été conçu pour la création de documents de très haute qualité typographique – soit des documents papier, soit des documents destinés à être affichés sur écran informatique. Il ne s'agit pas d'un traitement de texte ou d'un éditeur de texte, mais, comme je l'ai dit précédemment, d'un *système*, ou dit autrement *une suite d'outils*, destinés à la composition de documents, c'est-à-dire à la mise en page et à la visualisation des différents éléments du document sur la page ou à l'écran. ConTEXt, en résumé, vise à fournir tous les outils nécessaires pour donner aux documents la meilleure apparence possible. L'idée est de pouvoir générer des documents qui, en plus d'être bien écrits, sont également « beaux ». A cet égard, nous pouvons mentionner ici ce que Donald E. Knuth a écrit lors de la présentation de TEX(le système sur lequel est basé ConTEXt) :

> *Si vous voulez simplement produire un document passablement bon – quelque chose d'acceptable et essentiellement lisible mais pas vraiment beau – un système plus simple suffira généralement. Avec TEX l'objectif est de produire la* meilleure *qualité ; cela nécessite plus d'attention aux détails, mais vous ne trouverez pas cela beaucoup plus difficile, et vous pourrez être particulièrement fier du produit fini.*

Lorsque nous préparons un document avec ConTEXt, nous indiquons exactement comment celui-ci doit être transformé en pages (ou écrans) avec une qualité typographique et une précision de composition comparable à celle que l'on peut obtenir avec les meilleures imprimeries du monde. Pour ce faire, une fois que nous

---

* Cette préface a commencé avec l'intention d'être une traduction/adaptation à ConTEXt de la préface de « The TEXBook », le document qui explique *tout ce que vous devez savoir sur le TEX*. En fin de compte, j'ai dû m'en écarter ; cependant, j'en ai conservé quelques éléments qui, je l'espère, pour ceux qui le connaissent, feront écho.

avons appris le système, nous n'avons guère besoin de plus de travail que ce qui est normalement nécessaire pour taper le document dans n'importe quel traitement de texte ou éditeur de texte. En fait, une fois que nous avons acquis une certaine facilité de manipulation de ConTEXt, notre travail total est probablement moindre si nous gardons à l'esprit que les principaux détails de formatage du document sont décrits globalement dans ConTEXt, et que nous travaillons avec des fichiers texte qui sont – une fois que nous nous y sommes habitués – une façon beaucoup plus naturelle de traiter la création et l'édition de documents ; d'autant plus que ces types de fichiers sont beaucoup plus légers et plus faciles à traiter que les lourds fichiers binaires appartenant aux traitements de texte.

Il existe une documentation considérable sur ConTEXt, presque excmusivement en anglais. Par exemple, ce que l'on considère comme étant la distribution *officielle* de ConTEXt– appelée « ConTEXt Standalone » [1] – contient une documentation de quelques 180 fichiers PDF (la majorité en anglais, mais certains en néerlandais et en allemand) avec notamment des manuels, des exemples et des articles techniques ; et sur le site web de Pragma ADE (la société qui a donné naissance à ConTEXt) il y a (le jour où j'ai fait le décompte en mai 2020) 224 documents librement téléchargeables, dont la plupart sont distribués avec la « ConTEXt Standalone » mais quelques autres aussi. Cependant, cette énorme documentation n'est pas particulièrement utile pour durant la phase d'apprentissage de ConTEXt, car, en général, ces documents ne s'adressent pas à un lecteur novice, qui ne connaît rien du système, mais désireux d'apprendre. Sur les 56 fichiers PDF que « ConTEXt Standalone » appelle « manuels », un seul suppose que le lecteur ne connait rien sur ConTEXt. Il s'agit du document intitulé « ConTEXt Mark IV, an Excursion » ou en français « ConTEXt Mark IV, une escapade ». Mais ce document, comme son nom l'indique, se limite à présenter le système et à expliquer comment faire certaines choses qui peuvent être faites avec ConTEXt. Ce serait une bonne introduction s'il était suivi d'un manuel de référence un peu plus structuré et systématique. Mais un tel manuel n'existe pas et l'écart entre le document « ConTEXt Mark IV, an Excursion » et le reste de la documentation est trop important.

En 2001, un manuel de référence a été rédigé et peut être trouvé sur le site web Pragma ADE web ; mais malgré ce titre, d'une part il n'a pas été conçu pour être

[1] Au moment de la première version de ce texte, ceci était vrai ; mais au printemps 2020, le Wiki ConTeXt a été mis à jour et nous devons supposer qu'à partir de ce moment la distribution « officielle » de ConTeXt est devenue LMTX. Cependant, pour ceux qui entrent dans le monde de ConTeXt pour la première fois, je recommande quand même d'utiliser « ConTEXt Standalone » puisque c'est une distribution plus stable. L'annexe A explique comment installer l'une ou l'autre des distributions.

un manuel complet, et d'autre part il était (est) destiné à la version précédente de ConTEXt(appelé Mark II) et intégre des éléments obsolètes, ce qui perturbe l'apprentissage.

En 2013, le manuel a été partiellement mis à jour mais beaucoup de ses sections n'ont pas été réécrites et il contient des informations relatives à la fois à ConTEXt Mark II et ConTEXt Mark IV (la version actuelle), sans toujours préciser clairement quelles informations se rapportent à chacune des versions. C'est peut-être la raison pour laquelle ce manuel ne se trouve pas parmi les documents inclus dans « ConTEXt Standalone ». Pourtant, malgré ces défauts, le manuel reste le meilleur document pour commencer à apprendre ConTEXt une fois lu « ConTEXt Mark IV, an Excursion ». Autres informations également très utiles pour démarrer avec ConTEXt, celles contenues sur son wiki, qui, au moment où nous écrivons ces lignes, est en cours de remaniement et présente une structure beaucoup plus claire, bien qu'elles mélangent également des explications qui ne fonctionnent que dans Mark II avec d'autres pour Mark IV ou pour les deux versions. Ce manque de différenciation se retrouve également dans la liste officielle des commandes de ConTEXt[1] qui ne précise pas quelles commandes ne fonctionnent que dans l'une des deux versions.

Fondamentalement, cette introduction a été rédigée en s'inspirant des quatre sources d'information énumérées ici : La ConTEXt
[`<Excursion>`], le manuel 2013, le contenu du wiki et la liste officielle des commandes qui comprend, pour chacune d'elles, les options de configuration possibles ; en plus, bien sûr, de mes propres essais et tribulations. Ainsi, cette introduction est en fait le résultat d'un effort d'investigation, et durant un temps j'ai été tenté de l'appeler « ce que je sais sur ConTEXt Mark IV » ou « Ce que j'ai appris sur ConTEXt Mark IV ». Finalement, j'ai écarté ces titres car, aussi vrais qu'ils puissent être, j'ai pensé qu'ils risquaient de dissuader certains de s'investir dans ConTEXt ; il est certain, malgré que la documentation ait (à mon avis) certaines lacunes, que ConTEXt est un outil vraiment utile et polyvalent, pour lequel l'effort d'apprentissage vaut sans aucune doute la peine. Grâce à ConTEXt, nous pouvons manipuler et configurer des documents texte pour réaliser des choses que ceux qui ne connaissent pas le système ne peuvent tout simplement pas imaginer.

> Because of who I am, I cannot help the fact that my complaints about the lack of information will appear from time to time throughout this document. I would not like this to be misunderstood: I am immensely grateful to the creators of ConTEXt for having designed such a powerful tool and for having made it available to the public. It is

[1] Pour la liste, voir section ??.

> simply that I cannot avoid thinking that this tool would be much more popular if its documentation were improved: one has to invest a lot of time into learning it, not so much because of its intrinsic difficulty (which it has, but no greater than other similar tools – to the contrary in fact), but due to the lack of clear, complete and well-organised information that differentiates between the two versions of ConTEXt, explaining the functions in each of them and, above all, clarifying what each command, argument and option does.
>
> It is true that this kind of information demands great time investment. But given that many commands share options with similar names, perhaps a kind of *glossary* of options could be provided that would also help to detect some inconsistencies resulting from when two options with the same name do different things, or when, to do the same thing, one uses the names of different options in different commands.
>
> As for the reader who is approaching ConTEXt for the first time, let my complaints not dissuade you, because although it may be true that deficient information increases the time needed to learn it, at least for the material dealt with in this introduction I have already invested this time so that the reader does not have to do so. And just with what can be learned from this introduction, readers will have at their disposal a tool that will allow them to produce documents with an ease that they could never have suspected.

Since what is explained in this document comes to a large extent from my own conclusions, it is likely that even though I have personally tested most of what I explain, some statements or opinions may be neither correct nor very orthodox. I will, of course, appreciate any correction, nuance or clarification readers can offer me, and these can be sent to joaquin@ataz.org. However, to reduce the occasions where I am likely to be wrong I have tried not to enter into matters about which I have found no information and that I have not been able (or have not wanted) to personally try out. At times this is the case because the results of my tests were not conclusive, and at other times because I have not always tested everything: the number of commands and options ConTEXt has is impressive, and if I had to try everything out I would never have finished this introduction. There are occasions, however, when I cannot avoid *assuming* something, i.e. making a statement that I see as probable but that I am not completely sure about. In these cases, a «conjecture» image has been placed in the left margin of the paragraph where I am making such an assumption. The image aims to graphically represent the assumption.[1] At other times, I have no choice but to admit that I don't know something and I don't even have a reasonable assumption about it: in this case, the image visible to the immediate left in the margin is meant to represent more than just conjecture or ignorance.[2] But as I have never been very good with graphic

[1] I did not draw the image, but downloaded it from the internet (https://es.dreamstime.com/), where it says that it is a royalty-free image.

[2] Also found on the internet (https://www.freepik.es/) where its free use is authorised.

representations, I am not sure that the images I have selected really manage to convey so many nuances.

This introduction, on the other hand, has been written from the point of view of a reader who knows nothing about either TEX or ConTEXt, although I hope that it can also be useful to those coming from TEX or LATEX (the most popular of the TEX derivatives) who are approaching ConTEXt for the first time. Just the same, I am aware that in trying to please so many different kinds of reader, I run the risk of satisfying nobody. Therefore, in case of doubt, I have always been clear that the principal addressee of this document is the newcomer to ConTEXt, the newcomer who has just come to this fascinating ecosystem.

Being a newcomer to ConTEXt does not imply also being a newcomer to using computer tools; and although in this introduction I am not assuming any particular level of computer literacy in readers, I do presume a certain « reasonable literacy » that implies, for example, having a general understanding of the difference between a word processor and a text editor, knowing how to create, open and manipulate a text file, knowing how to install a program, knowing how to open a terminal and execute a command... and little else.

> Reading through the previous parts of this introduction as I write these lines, I realise that sometimes I get carried away and get into computer issues that are not necessary for learning ConTEXt and that could scare the newcomer off, while at other times I am busy explaining quite obvious things that could bore the experienced reader. I beg the indulgence of both. Rationally, I know that it is very difficult for a complete beginner in computerised text management to even know that ConTEXt exists, but from another point of view, in my professional environment I am surrounded by people who are constantly struggling with texts when they used word processors, and they do so reasonably well, but never having worked with text files as such they ignore such basic issues as, for example, what encoding text files use or what the difference is between a word processor and a text editor.

The fact that this manual is designed for people who know nothing about ConTEXt or TEX, implies that I have included information that clearly is not about ConTEXt but TEX; but I have understood that it is not necessary to burden readers with information that is not relevant for them, as could be the case if a certain command that *in fact* works, is really a ConTEXt command or belongs to TEX; so only on some occasions, when it seems to me to be useful, do I clarify that certain commands really belong to TEX.

With regard to the organisation of this document, the material is grouped into three blocks:

- **The first part**, comprising the first four chapters, offers a global overview of ConTEXt, explaining what it is and how we work with it, showing a first example of how to transform a document so as to be able, later, to explain some fundamental concepts of ConTEXt along with certain questions relating to ConTEXt source files.

  As a whole, these chapters are intended for readers who up until now have only known how to work with word processors. A reader who already knows about working with markup languages could forgo these early chapters; and if the reader already knows TEX, or LATEX, they could also skip much of the content in Chapters 3 and 4. Just the same, I would recommend at least reading:

  – The information relating to ConTEXt commands (Chapter 3), and in particular how it functions, how it is configured, because this is where the principal difference lies between the conception and syntax of LATEX and ConTEXt. Since this introduction refers only to the latter, these differences are not expressly indicated as such, but someone reading this chapter who knows how LATEX works will immediately understand the difference in syntax of the two languages, as also the way that ConTEXt allows us to configure and customise the way almost all of its commands work.

  – The information relating to multifile ConTEXt projects (Chapter 4), which is not so similar to the way of working with other TEX-based systems.

- **The second part**, that includes Chapters 5 to 9, focuses on what we consider to be the main global aspects of a ConTEXt document:

  – The two aspects that mainly affect the appearance of a document are the size and layout of its pages and the font used. Chapters 5 and 6 are dedicated to these matters.

    ⋆ The first focuses on pages: size, the elements that make up a page, its layout (meaning, how the page elements are distributed), etc. For systematic reasons, more specific aspects are also dealt with here, such as those relating to pagination and the mechanisms that allow us to influence it.

    ⋆ Chapter 6 explains commands related to the fonts and their handling. Also included here is a basic explanation of the use and management of colours since, although these are not strictly a *characteristic* of fonts,

they are just as much an influence on the external appearance of the document.

– Chapters 7 and 8 focus on the structure of the document and the tools that ConTEXt makes available to the author for writing well-structured documents. Chapter 7 focuses on structure properly so called (structural divisions of the document) and Chapter 8 on how this is reflected in the Table of Contents; although, in line with the explanation of this, we use the opportunity to also explain how to generate various kinds of indexes with ConTEXt, since for ConTEXt these all come under the notion of « lists ».

– Finally, Chapter 9 focuses on references, an important global aspect of any document when we need to refer to something in another part of the document (internal references) or to other documents (external references). In the case of the latter, we are only interested for the moment in references (links) that mean going to an external document. These *links* (that can also occur in internal references) make our document *interactive*, and in this chapter we explain some of the features of ConTEXt for creating these kinds of documents.

These chapters do not need to be read in any particular order except for Chapter 8, which may be easier to understand if Chapter 7 has been read first. In any case, I have tried to ensure that when a question arises in a chapter or section that is dealt with elsewhere in this introduction, the text includes a mention of that together with a hyperlink to the point where the question is dealt with. However, I am not in a position to guarantee that this will always be the case.

- Finally **the third part** (Chapters 10 and following) focuses on more detailed aspects. They are independent not only of each other, but even of their sections (except, perhaps, in the last chapter). Given the large number of utilities that ConTEXt incorporates, this part could be very extensive; but since my understanding is that by the time they arrive here readers will already be prepared to dive into ConTEXt documentation of their own accord, I have only included the following chapters:

  – Chapters 10 and 11 deal with what we could call the *core elements* of any text document: the text is made up of characters which make up words that are grouped on lines, which in turn make up paragraphs separated from one another by vertical space... Clearly, all these issues could have been included in a single chapter, but as this would be too long, I have divided this matter into two chapters, one that deals with characters, words and

horizontal space and another that deals with lines, paragraphs and vertical space.

  - Chapter 12 is a kind of *mishmash* dealing with elements and constructions commonly found in documents; for the most part academic or scientific or technical documents: footnotes, structured lists, descriptions, numbering, etc.

  - Finally, Chapter 13 focuses on floating objects especially the most typical of these: images inserted into documents, and tables.

- The introduction closes with three **appendices**. One is about installing ConTEXt, a second appendix contains several dozen commands that allow the generation of various symbols – mainly but not only for mathematical use, and a third appendix contains an alphabetical list of ConTEXt commands explained or mentioned in the course of this text.

There are many issues that remain to be explained: dealing with quotes and bibliographic references, writing specialised texts (maths, chemistry...), the connection with XML, the interface with Lua code, modes and processing based on modes, working with MetaPost for designing graphics, etc. This is why, since I am not including a complete explanation of ConTEXt, nor am I pretending to do so, I have called this document « An introduction to ConTEXt Mark IV »; and I have added the fact that the introduction is none too short, because obviously this is the case: a text that has left so many things still in the pipeline but that has already gone beyond 300 pages is not by any means a short introduction. This is because I want the reader to understand the logic of ConTEXt, or at least the logic as I have understood it. It does not claim to be a reference manual, but rather a guide for self-learning that prepares the reader to produce documents of medium complexity (and this includes most of the likely documents) and that above all teaches the reader to *imagine* what can be done with this powerful tool and find out how to do it in the documentation available. Nor is this document a *tutorial*. Tutorials are designed to progressively increase the level of difficulty, so that what is to be learned is taught step by step; in this respect I have preferred to begin with the second part instead of ordering material according to the level of difficulty, in order to be more systematic. But while it is not a tutorial, I have included very many examples.

It is possible that for some readers this document's title reminds them of a text written by Oetiker, Partl, Hyna and Schlegl available on the internet and one of the better documents for introducing oneself to the LaTeX world. I am talking about « *The Not So Short Introduction to LaTeX* 2_ϵ ». This is no coincidence, but a tribute and

act of appreciation: thanks to the generous work of those who write texts like that, it is possible for many people to begin to work with useful and powerful tools like LaTeX and ConTeXt. These authors helped me to start out with LaTeX; I am hoping to do the same with someone who wants to start out with ConTeXt, even though in the original Spanish version of this text I stuck exclusively to the Spanish-speaking world who have lacked so much documentation in their language. I hope this document fulfils this expectation, and in the meantime, others have generously offered to translate it into other languages, hence this English edition. Thank you.

Joaquín Ataz-López
Summer 2020

I
What is ConTEXt and how do we work with it

Chapitre 1

ConTEXt: a general overview

Table of Contents: **1.1 What is ConTEXt then?;** **1.2 Typesetting texts;** **1.3 Markup languages;** **1.4 TEX and its derivatives;** 1.4.1 TEX engines; 1.4.2 Formats derived from TEX; **1.5 ConTEXt;** 1.5.1 A short history of ConTEXt; 1.5.2 ConTEXt versus LATEX; 1.5.3 A good understanding of the dynamics of working with ConTEXt; 1.5.4 Getting help with ConTEXt;

1.1 What is ConTEXt then?

ConTEXt is a *typesetting system*, or in other words: an extensive set of tools aimed at giving the user absolute and complete control over the appearance and presentation of a specific electronic document intended for print on paper or to be shown on screen. This chapter explains what this means. But first, let us highlight some of the characteristics of ConTEXt.

- There are two *flavours* of ConTEXt known as Mark II and Mark IV respectively. ConTEXt Mark II is frozen, i.e. it is considered to be an already fully-developed language that is not intended to have further changes or new things added. A new version would appear only in the case where some error needs to be corrected. ConTEXt Mark IV, on the other hand, continues to be developed so that new versions appear from time to time that introduce some improvement or additional utility. But, although still in development, it is a very mature language in which changes introduced by the new versions are quite subtle and exclusively affect the low level operation of the system. For the average user these changes are totally transparent; it is as if they did not exist. Although both *flavours* have much in common, there are also some incompatible features between them. Hence this introduction focuses only on ConTEXt Mark IV.

- ConTEXt is software *libre* (or free software, but not just in the sense of *gratis*). The program properly speaking (that is, the complex of computer tools that make up ConTEXt), is distributed under the *GNU General Public Licence*. The documentation is offered under the « *Creative Commons* » licence that allows it to be freely copied and distributed.

- ConTEXt is neither a word processor program nor a text editing program, but a collection of tools aimed at *transforming* a text we have previously written in our favourite text editor. Therefore, when we work with ConTEXt:
  - We begin by writing one or more text files with any kind of text editor.
  - In these files, along with the text that makes up the contents of the document, there is a range of instructions that tell ConTEXt about the appearance that the final document generated from the original text files must have. The complete set of ConTEXt instructions, in fact, is a *language*; and since this language allows one to *program* the typographical transformation of a text, we can say that ConTEXt is a *typographical programming language*.
  - Once we have written the source files, these will be processed by a program (also called « `context` »[1]), which will generate a PDF file from them ready to be sent to a print shop or to be shown on screen.
- In ConTEXt, therefore, we must differentiate between the document we are writing, and the document that ConTEXt generates. To avoid any doubts, in this introduction I will call the text document that contains formatting instructions the *source file*, and the PDF document generated by ConTEXt from the source file I will call the *final document*.

The above basic points will be further developed below.

1.2 Typesetting texts

Writing a document (book, article, chapter, leaflet, print out, paper ...) and putting it together typographically are two very different activities. Writing the document is much the same as drafting it; this is done by the author who decides on its content and structure. The document created directly by the author, just as he or she wrote it, is called the *manuscript*. By its very nature, only the author or those permitted to read it have access to the manuscript. Its dissemination beyond this

[1] ConTEXt is both a language and a program at the same time (besides being other things). This fact, in a text like the current one, poses the problem that at times we have to distinguish between these two aspects. This is why I have adopted the typographical convention of writing « ConTeXt » with its logo (ConTEXt) when I want to refer exclusively to the language, or to both the language and the program. However, when I want to refer exclusively to the program, I will write « `context` » all in lower case and in the monospaced type that is typical of computer terminals and typewriters. I will also use this type for examples and mentions of commands belonging to this language.

intimate group requires the manuscript to be *published*. Today, publishing something – in the etymological sense of making it « accessible to the public » – is as simple as putting it on the internet, available to anyone who finds it and wants to read it. But until relatively recently, publication was a cost-intensive process dependent on certain professionals specialised in it, only accessed by those manuscripts which, because of their content, or because of their author, were considered to be particularly interesting. And even today we tend to reserve the word *publication* for this kind of *professional publication* where the manuscript undergoes a series of transformations in its appearance aimed at improving the *legibility* of the document. This series of transformations is what we call *typesetting*.

The aim of typesetting is – generally speaking, and leaving aside advertisement-type texts that seek to attract the reader's attention – to produce documents with the greatest *legibility*, meaning the quality of the printed text that invites or facilitates its reading and ensures that the reader feels comfortable with it. Many things contribute to this; some, of course, have to do with the document's *contents*: (quality, clarity, organisation...), but others depend on things like the type and dimensions of the font used, the use of white space in the document, visual separation between paragraphs, etc. In addition, there are other kinds of resources, not so much of the graphic or visual kind, such as the presence or otherwise in the document of specific aids to the reader like page headers and footers, indexes, glossaries, use of bold type, margin headings, etc. The knowledge and correct handling of all the resources available to a typesetter could be called the « art of typesetting » or the « art of printing ».

Historically, and until the advent of the computer, the tasks and roles of writer and typesetter were kept quite distinct. The author wrote by hand or on a 19th century machine called a typewriter, the typographical resources of which were even more limited than those who wrote by hand; and then the writer gave the originals to the publisher or printer who transformed them to obtain the printed document.

Today, computer science has made it easier for the author to decide on the composition down to the last detail. However, this does not do away with the fact that the qualities that a good author needs are not the same as those needed by a good typesetter. Depending on the kind of document being dealt with, the author needs an understanding of the subject matter being written about, clarity of exposition, well-structured thinking that allows for the creation of a well-organised text, creativity, a sense of rhythm, etc. But the typesetter has to combine a good knowledge of the conceptual and graphical resources at his or her disposal, and sufficient good taste to be able to use them harmoniously.

With a good word processing program[1] it is possible to achieve a reasonably good typographically prepared document. But word processors, generally speaking, are not designed for typesetting and the results, although they may be correct, are not comparable to the results obtainable with other tools designed specifically to control the composition of the document. In fact, word processors are how typewriters evolved, and their use, to the extent that these tools mask the difference between the authorship of the text and its typesetting, tends to produce unstructured and typographically inadequate texts. On the contrary, tools like ConTEXt have evolved from the printing press; they offer many more composition possibilities and above all, it is not possible to learn how to use them without also acquiring, along the way, many notions relating to typesetting. This is the difference from word processors, which someone can use for many years without learning a single thing about typography.

1.3 Markup languages

In the days before computers, as I said before, the author prepared the manuscript by hand or typewriter and handed it to the publisher or printer who was responsible for the transformation of the manuscript into the final printed text. Although the author had relatively little involvement in the transformation, he or she did maintain some intervention by pointing out, for example, that certain lines of the manuscript were the titles of its various parts (chapters, sections ...), or by indicating that certain things should be highlighted typographically in some way. These indications were made by the author in the manuscript itself, sometimes expressly, and at other times through certain conventions that continued to develop over time. For example, the chapters always began on a new page by inserting several blank lines before the title, underlining it, writing it in capital letters, or framing the text to be highlighted between two underscores, increasing the indentation of a paragraph, etc.

To put it briefly, the author *marked up* the text in order to provide indications relating to how it should be typeset. Then later, the editor would handwrite other indications on the text for the printer, such as, for example, the font to be used, and its size.

[1] According to a rather old convention, we make a distinction between *text editors* and *word processors*. The early kinds of text editing programs dealt with unformatted text files, while the other kind worked with binary files of formatted text.

Today, in a computerised world, we can continue to do this to generate electronic documents through what is called a *markup language*. These kinds of languages use a series of *marks* or indications that the program processing the file containing them knows how to interpret. Probably the best known markup language today is HTML, since most web pages today are based on it. An HTML page contains the text of a web page, along with a series of marks that tell the browser program that loads the page how it should display it. The HTML markup that web browsers understand, together with instructions about how and where to use them, is called the « HTML language », which is a *markup language*. But as well as HTML, there are many other markup languages; in fact they are booming, and so XML, which is the markup language *par excellence*, is found everywhere today and is in use for pretty much everything: for database design, for the creation of specific languages, transmission of structured data, application configuration files, etc. There are also markup languages intended for graphic design (SVG, TikZ or MetaPost), maths formulas (MathML), music (Lilypond and MusicXML), finance, geomatics, etc. And of course there are also markup languages aimed at the typographical transformation of text, and among these, TEX and its derivatives stand out.

With regard to the *typographical* markup that indicates how a text should look, there are two kinds that we can refer to: *purely typographical markup* and *conceptual markup* or, if you prefer, *logical markup*. Purely typographical markup is limited to indicating precisely what typographical resource should be used to display a certain text; such as when, for example, we indicate that certain text should be in bold or italics. Conceptual markup, on the other hand, indicates what function complies with certain text in the document as a whole, such as when we indicate that something is a title, or a subtitle, or a quote. In general, documents that prefer to use this second kind of markup are more consistent and easier to compose, since once again they point out the difference between authorship and composition: the author indicates that such and such a line is a title, or that such and such a fragment is a warning, or a quote; and the typesetter decides how to typographically highlight all titles, warnings or quotations; thus, on the one hand, consistency is guaranteed, as all the fragments that fulfil the same function will look the same, and, on the other hand it saves time, because the format of each type of fragment only needs to be indicated once.

1.4 TEX and its derivatives

TEX was developed towards the end of the 70s by Donald E. Knuth, a professor (now emeritus professor) of theoretical computer programming at Stanford Uni-

versity, who implemented the program to produce his own publications and as an example of a systematically developed and annotated program. Along with TEX, KNUTH developed an additional programming language called METAFONT, created for designing typographical fonts, and he used it to design a font he baptised as *Computer Modern*, which, along with the usual characters of any font, also included a complete set of « glyphs »[1] designed for writing mathematics. To all this he added some additional utilities and thus the typesetting system called TEX was born, which, due to its power, quality of results, flexibility of use and broad possibilities, is considered one of the best computerised systems for text composition. It was designed for texts in which there was a lot of mathematics, but it soon became clear that the system's possibilities made it suitable for all kinds of texts.

Internally, TEX functions in the same way as the former compositors would do in a print shop. For TEX, everything is a *box*: The letters are contained in boxes, the blank spaces are also boxes, several letters (the boxes containing several letters) form a new box that encloses the word, and several words, along with the blank space between them, form a box containing a line, several lines become a box containing the paragraph ... and so on. All this, moreover, with extraordinary precision in the handling of measurements. Consider that the smallest unit that TEX deals with is 65.536 times smaller than the typographical point with which characters and lines are measured, which is usually the smallest unit handled by most word processing programs. This means that the smallest unit handled by TEX is approximately 0.000005356 millimetres.

The name TEX comes from the root of the Greek word $\tau\epsilon\chi\nu\eta$, written in upper case letters (TÉXNH). Therefore, the final letter of the word TEX is not a Latin «X», but the Greek «χ», pronounced – apparently – like the Scottish «ch» in *loch*. So TEX should be pronounced as *Tech*. This Greek word, on the other hand, meant both « art » and « technology », and this is the reason why KNUTH chose it to name his system. The purpose of this name – he wrote – « is to remind you that TEX is primarily concerned with high quality technical manuscripts. Its emphasis is on art and technology, as in the underlying Greek word ».

Using the convention established by KNUTH, TEX is to be written:

- In typographically formatted texts like this one, using the logo that I have been using until now: the three letters in upper case, with the central «E» slightly

[1] In typography, a glyph is the graphical representation of a character, a number of characters or part of a character, and is today's equivalent of the letter type (the bit engraved with the letter or movable type).

displaced below to facilitate a closer alignment between the «T» and the «X»; or in other words: « TEX ».

To facilitate the writing of this logo, Knuth included an instruction in TEX for writing it in the final document: `\TeX`.

- In unformatted texts (such as an email, or a text file), with the «T» and the «X» in upper case, and the central «e» in lower case; so: « TeX ».

This convention continues to be used in all derivatives of TEX that include its proper name, as is the case with ConTEXt. When writing it in text mode we need to write « ConTeXt ».

1.4.1 TEX engines

The TEX program is free *libre* software: its source code is available to the public and anyone can use it or modify it as they wish, with the only condition that, if modifications are made, the result cannot be called « TEX ». This is why, over time, certain adaptations of the program have emerged, introducing different improvements to it, and which are generally referred to as *TEX engines*. Apart from the original TEX program, the main engines are, in chronological order of appearance, pdfTEX, ε-TEX, XƎTEX and LuaTEX. Each of them is supposed to incorporate the improvements of the previous one. These improvements, on the other hand, up until the appearance of LuaTEX, did not affect the language itself, but only the input files, the output files, handling of sources and low level operation of macros.

The question of which TEX engine to use is a much debated one within the TEX universe. I will not develop this question here since ConTEXt Mark IV only works with LuaTeX. In reality, in the ConTEXt world, discussion on TEX *engines* becomes a discussion on whether to use Mark II (that works with PdfTeX and XeTeX) or Mark IV (that only works with LuaTeX).

1.4.2 Formats derived from TEX

The core or heart of TEX only understands a set of approximately 300 very basic instructions, called *primitives*, which are suitable for typesetting operations and programming functions. The great majority of these instructions are of a very *low level*, which, in computer terminology, means that they are more easily understandable by the computer than by human beings, since they concern very elementary operations of the « shift this character 0.000725 millimetres upward » kind. Hence Knuth saw that TEX would be extensible, meaning that there should be a mechanism that allows instructions to be defined at a higher level, more easily understandable by human beings. These instructions, that are broken down into other

simpler instructions at the time of execution, are called *macros*. For example, the TEX instruction that prints the (`\TeX`) logo, is broken down as follows at the time of execution:

```
T
\kern -.1667em
\lower .5ex
\hbox {E}
\kern -.125em
X
```

But for the human being, it is much easier to understand and remember that the simple command « `\TeX` » carries out the typographical operations needed to print the logo.

> The difference between what is a *macro* and what is a *primitive*, really only has importance from the perspective of the TEX developer. From the user's perspective they are *instructions* or, if you prefer, *commands*. Knuth called them *control sequences*.

This possibility of extending the language through *macros* is one of the characteristics that turned TEX into such a powerful tool. In fact, Knuth himself created approximately 600 macros that, along with the 300 primitives, make up the format called « Plain TEX ». It is quite common to confuse TEX properly so called, with Plain TEX and, in fact, almost everything usually written or said about TEX, is really a reference to Plain TEX. Books that claim to be about TEX (including the foundational « *The TEXBook* »), really refer to Plain TEX; and those who believe they are directly working with TEX are in reality working with Plain TEX.

Plain TEX is what, in TEX terminology, is called a *format*, consisting of a broad set of macros, together with certain rules of syntax concerning how and in what way to use them. As well as Plain TEX, with the passing of time other *formats* have been developed, among which it is worth mentioning LATEX, a broad set of macros for TEX created in 1985 by Leslie Lamport and which is probably the TEX derivative that is most in use in the academic, technological and mathematical world. ConTEXt is (or has begun to be), on a par with LATEX as a format derived from TEX.

Normally these *formats* are accompanied by a programme that loads the macros that make them up into memory before calling on « `tex` » (or the actual engine being used for processing) to process the source file. But even though all these formats are actually running TEX, as each of them has different instructions and different syntax rules from the user's point of view, we can think of them as *different languages*. They all draw their inspiration from TEX, but are different from TEX and also different from each other.

1.5 ConTEXt

In reality ConTEXt, which started out as a *format* of TEX, is much more than that today. ConTEXt includes:

1. A very broad set of TEX macros. If Plain TEX has around 900 instructions, ConTEXt has around 3500; and if we add up the names of the different options that these commands support, we are talking about a vocabulary of around 4000 words. The vocabulary is this large because of the ConTEXt strategy to facilitate its learning, and this strategy means the inclusion of any number of synonyms for commands and options.

   > The intention is that if a certain effect is to be achieved, then for each of the ways an English speaker would call that effect there is a command or option that achieves it – which is supposed to make the use of the language easier. For example, to simultaneously get a bold and italic letter, ConTEXt has three instructions all of which achieve the same result: `\bi`, `\italicbold` and `\bolditalic`.

2. A likewise broad set of macros for MetaPost, a graphical programming language derived from METAFONT, which in turn is a language for typeface design that KNUTH developed jointly with TEX.

3. Various *scripts* developed in PERL (the oldest), RUBY (some also old, others not so old) and LUA (the most recent).

4. An interface that integrates TEX, MetaPost, LUA and XML, allowing one to write and process documents in any of these languages, or to mix elements from some of them.

   > Perhaps you did not understand much of the previous explanation? Don't worry about it. I used a lot of computer jargon in it and mentioned many programs and languages. It is not necessary to know all the different components to use ConTEXt. The important thing, at this stage of learning, is to stay with the idea that ConTEXt integrates many tools from different sources that together make up a *typesetting system*.

It is because of this latter feature of integration of tools with different origins, that we say that ConTEXt is a « hybrid technology » intended for typesetting documents. My understanding is that this turns ConTEXt into an extraordinarily advanced and powerful system.

Even though ConTEXt is much more than a collection of macros for TEX, it continues to be based on TEX, and this is why this document, that I claim to be no more than an *introduction*, focuses on this.

ConTEXt, on the other hand, is rather more modern than TEX. When TEX was created, the emergence of computers was just at the beginning, and we were far from seeing what the internet and the multimedia world would be (would become). In this respect, ConTEXt naturally integrates some of the things that have always been something of a foreign body in TEX such as including external graphics, handling colour, hyperlinks in electronic documents, assuming a paper size suitable for a document intended for display on a screen, etc.

1.5.1 A short history of ConTEXt

ConTEXt was born approximately in 1991. It was created by Hans Hagen and Ton Otten in a Dutch document design and processing company called « *Pragma Advanced Document Engineering* », usually abbreviated as Pragma ADE. It began by being a collection of TEX macros that had Dutch names and was unofficially known as *Pragmatex*, aimed at the company's non-technical employees who had to manage the many details of editing typeset documents and who were not used to using markup languages or interfaces other than Dutch. Hence the first version of ConTEXt was written in Dutch. The idea was to create a sufficient number of macros with a uniform and consistent interface. Approximately in 1994 the *package* was stable enough for a user manual to be written in Dutch, and in 1996, through the initiative of Hans Hagen, reference to it began taking on the name « ConTEXt ». This name claims to mean « Text with TEX » (using the Latin preposition «con» meaning «with»), but at the same time a wordplay on the English (and Dutch) word « Context ». Behind the name, therefore, lies a triple play on words involving « TEX », « text » and « context ».

> Therefore, since the name is based on wordplay, ConTEXt should be pronounced «context» and not «contecht» since this would mean losing the play on words.

The interface began to be translated into English approximately in 2005, giving rise to the version known as ConTEXt Mark II, where the «II» is explained because in the mind of the developers, the previous version in Dutch was Version «I», even though it was never officially called that. After the interface was translated into English, the use of the system began to spread beyond the Netherlands, and the interface was translated into other European languages such as French, German, Italian and Romanian. The « official » documentation for ConTEXt, nevertheless, is normally based on the English version, and this is the version this document works with.

In its initial version, ConTEXt Mark II worked with the PdfTeX *TEX engine*. But later, at the appearance of the XETEX *engine*, ConTEXt Mark II was modified to allow the

use of this new engine that contributed a number of advantages by comparison with PdfTeX. But when LuaTEX came along some years later, the decision was made to internally reconfigure how ConTEXt functioned in order to integrate all the new possibilities that this new engine offered. And so, ConTEXt Mark IV was born, and it was presented in 2007, immediately after the presentation of LuaTEX. Very probably, one of the influencing factors in the decision to reconfigure ConTEXt to adapt it to LuaTEX was that two of the three main developers of ConTEXt, Hans Hagen and Taco Hoekwater, were also part of the main team developing LuaTEX. This is why ConTEXt Mark IV and LuaTEX were born at the same time and developed in unison. There is a synergy between ConTEXt and LuaTEX that does not exist in any other derivative of TEX; and I doubt that any of the others can avail themselves of the advantages of LuaTEX as ConTEXt can.

There are many differences between Mark II and Mark IV, although most of them are *internal*, that is, they have to do with how the macro actually works at a lower level, such that from the user's perspective the differences are not noticeable: the name and parameters of the macro remain the same. There are, however, some differences that affect the interface and force one to do things differently depending on which version one is working with. These differences are relatively few, but they do affect very important aspects such as for example, the coding of the input file, or the handling of fonts installed in the system.

It would, however, be very welcome if somewhere there were a document that explained (or listed) the appreciable differences between Mark II and Mark IV. In the ConTEXt wiki, for example, for each ConTEXt command there are *two kinds of syntax* (very often identical). I presume one belongs to Mark II and the other to Mark IV; and based on this assumption, I also presume that the *first version* is from Mark II. But the truth is that the wiki tells us nothing about this.

The fact that the differences, at a language level, are relatively few, means that on many occasions rather than speaking of two « flavours » of ConTEXt. But whether you call them one or the other, the fact is that a document prepared for Mark II cannot normally be compiled with Mark IV and vice versa; and if the document mixes both versions, it will most likely not compile well with either of them; which implies that the author of the source file has to start by deciding whether to write for Mark II or for Mark IV.

If we work with the different versions of ConTEXt, a good trick for differentiating at first sight between files intended for Mark II and those intended for Mark IV is to use a different extension for the file names. Thus, for example, for any files I have written for Mark II, I put « `.mkii` » as the extension, and « `.mkiv` » instead for those written for Mark IV. It is true that ConTEXt expects all source files to have the extension « `.tex` »,

but we can change the file extension as long as we expressly indicate the file extension when applying ConTEXt to the file.

The ConTEXt distribution installed on the wiki, « ConTEXt Standalone », includes both versions, and to avoid confusion – I assume – uses a different command for each of them to compile a file. Mark II compiles with the command « `texexec` » and Mark IV with the command « `context` ».

In fact both commands, « `context` » and « `texexec` », are *scripts* with different options that run « `mtxrun` », which in turn is a Lua *script*.

Today, Mark II is frozen and Mark IV continues to be developed, which means that new versions of the former are only published when errors or faults are discovered that need to be corrected, while new versions of Mark IV continue to be published regularly; sometimes two or three times a month, even though in most of these cases the « new versions » do not introduce perceptible changes in the language but are limited to somehow improving implementation of a command at low level, or updating some of the many manuals included with the distribution. Even so, if we have installed the development version – which is what I would recommend and which is the one installed by default with « ConTEXt Standalone » – it makes sense to update our version from time to time (See Appendix A for the way of updating the installed version of « ConTEXt Standalone »).

LMTX and other alternative implementations of Mark IV

The developers of ConTEXt are naturally restless, and therefore have not ceased development of ConTEXt with Mark IV; new versions are still being tested and experimented with, although in general these differ from Mark IV in very few ways, and do not have the incompatibility in compiling that exists between Mark IV and Mark II.

Thus, certain minor variants of Mark IV called, respectively, Mark VI, Mark IX and Mark XI have been developed. Of these, I have only been able to find a small reference to Mark VI in the ConTEXt wiki where it says that the only difference with Mark IV lies in the possibility of defining commands by assigning the parameters not a number, as is traditional in TEX, but a name, as is usually done in almost all programming languages.

More important than these small variations, I believe, is the appearance in the ConTEXt universe (ConTEXtverse?) of a new version called LMTX, a name which is an acronym of LuaMetaTeX: a new TEX *engine* that is a simplified version of LuaTEX, developed with a view to saving computer resources; which means that LMTX requires less memory and less processing power than ConTEXt Mark IV.

LMTX was presented in spring 2019 and one assumes that it will not imply any external change to the Mark IV language. For the author of the document there would be no difference at the time of working with it; but when compiling it, one would need to choose between doing so with LuaTEX, or doing so with LuaMetaTeX. In Appendix A,

relating to the installation of ConTEXt, a procedure is shown for assigning a different command name to each of the installations (section 3).

1.5.2 ConTEXt versus LATEX

Given that the most popular format derived from TEX is LATEX, a comparison between this and ConTEXt is inevitable. Clearly we are talking about different languages although in some way related to each other since they both derive from TEX; the relationship is similar to that which exists, for example, between Spanish and French: languages that have a common origin (Latin) which means that their syntax is *similar* and many of the words in each of these languages is mirrored by a word in the other. But apart from this *family resemblance*, LATEX and ConTEXt differ in their philosophy and implementation, since the initial aims of both, are, to some degree, the opposite. LATEX claims to facilitate the use of TEX, isolating the author from the concrete typographical details to help focus on content, leaving the typesetting details in the hands of LATEX. This means that simplifying the use of TEX takes place at the expense of limiting the immense flexibility of TEX, by predefining basic formats and limiting the number of typographical issues that the author has to decide on. In contrast to this philosophy, ConTEXt was born within a company dedicated to typesetting documents. Therefore, far from wanting to isolate the author from typesetting details, the aim is to give the author absolute and complete control over them. To achieve this, ConTEXt provides a uniform and consistent interface which is much closer to the original spirit of TEX than LATEX.

This difference in philosophy and founding objectives then translates, in turn, into a difference in implementation. LATEX, that tends to simplify things as much as possible, does not need to use all of TEX's resources. In some way, its core is rather simple. So when there is a need to broaden its possibilities, it is necessary to expressly write a *package* to do so. This *packaging* associated with LATEX is both a virtue and a defect: a virtue, because the tremendous popularity of LATEX, together with the generosity of its users, means that almost any need we are likely to have has been met by someone before, and that there is a package to achieve it; but it is also a defect because these packages are often incompatible with each other, and their syntax is not always uniform. This means that working with LATEX requires one to constantly be searching through thousands of already existing packages to fulfil one's needs and ensure that they all work together.

By contrast with the simplicity of the LATEX core, which is complemented by its extensibility through packages, ConTEXt is designed to have within it all – or almost all – the typographical possibilities of TEX, so its conception is much more monolithic, but at the same time it is also more modular. The ConTEXt core allows

us to do almost everything, and we are guaranteed that there will be no incompatibilities between its different utilities, no need to investigate extensions for what we need, and the syntax of the language does not change just because we need a particular utility.

It is true that ConTEXt has what are called extension *modules* that some might consider as carrying out a function similar to the LATEX packages, but in real terms they both work differently: ConTEXt modules are designed exclusively to include additional utilities that, because they are still in an experimental stage, have not yet been incorporated into the core, or to allow access to extensions authored by someone outside the ConTEXt development team.

I do not believe that either one of these two *philosophies* is preferable to the other. The question depends rather on the user's profile and what he or she wants. If the user does not want to deal with typographical issues but simply produce very high quality standardised documents, it would probably be preferable to opt for a system like LATEX; on the other hand, the user who likes to experiment, or who needs to control every last detail of the document, or someone who has to devise a special layout for a document, would probably be better off using a system like ConTEXt, where the author has all the control in their hands; with the risk, of course, of not knowing how to use this control correctly.

1.5.3 A good understanding of the dynamics of working with ConTEXt

When we work with ConTEXt, we always begin by writing a text file (which we call a *source file*), in which, along with the actual content of our final document, we will include the instructions (in ConTEXt-speak) that indicate exactly how we want the document to be formatted: the general appearance we want its pages and paragraphs to have, the margins we want to apply to certain paragraphs, the font we want to display, the snippets we want shown in a different font, etc. Once we have written the source file, we apply the « `context` » program from a terminal, which will process it, and will generate a different file from it in which the contents of our document will be formatted in accordance with the instructions included in the source file for this purpose. This new file could be sent to a (commercial) printer, displayed on screen, placed on the internet or distributed among contacts, friends, clients, teachers, pupils ... or in other words, to anyone for whom we wrote the document.

This means that when working with ConTEXt the author is working with a file whose appearance has nothing to do with the final document: the file the author is

directly working on is a text file that is not formatted typographically. So ConTEXt works in a different way than do programs known as *word processors* that show the final appearance of the edited document at the same time we are writing it. For those accustomed to word processors, the way of working with ConTEXt will initially feel strange, and it may even take some time to get used to it. However, once one gets used to it, one understands that in reality this other way of working, differentiating between the work file and the final result, is actually an advantage for many reasons, among which I will highlight here, without following any particular order, the following:

1. Because text files are «lighter» to handle than word processor binary files, and editing them requires less computer memory, they are less likely to be corrupted, and they do not become unintelligible when we change the version of the program we are creating them with. They are also compatible with any operating system, and can be edited with many text editors, so that in order to work with them it is not necessary for the computer system to have the program the file was created with installed on it: any other editing program will do; and in every computer system there is always some text editing program.

2. Because differentiating between the working document and the final document helps to distinguish what the actual content of the document is from what its appearance will be, allowing the author to concentrate on the content in the creation phase, and to focus on the appearance in the typesetting phase.

3. Because it allows one to quickly and accurately change the appearance of the document, since this is determined by ConTEXt commands that can be easily identified.

4. Because this facility for changing the appearance, on the other hand, allows us to easily generate two (or more) different versions from a single content: Perhaps one version optimised for printing on paper, and another designed to be displayed on screen, adjusted to the size of the latter and perhaps including hyperlinks that make no sense in a paper document.

5. Because typographical errors (typos) that are common in word processors, such as extending the italics beyond the last character of a word, are also easily avoided.

6. Because while the work file is not distributed and is «for our eyes only», it is possible to incorporate annotations and observations, comments and warnings for ourselves for subsequent revisions or versions, with the peace of

mind in knowing that these will not appear in the formatted file to be distributed.

7. Because the quality that can be obtained by processing the whole document simultaneously is much higher than that which can be achieved with a program that has to make typographical decisions as the document is being written.

8. Etcetera.

All of the above means that on the one hand when working with ConTEXt, once we have got the hang of it, we are more efficient and productive, and that on the other hand, the typographical quality we can obtain is much superior to what can be obtained with so-called *word processors*. And although it is true that the latter are easier to use, in point of fact they are not that *much* easier to use. Because while it is true that ConTEXt, as we have said before, contains 3500 instructions, a normal user of ConTEXt will not need to know them all. To do what is usually done with word processors, we only need to know the instructions that allow us to indicate the structure of the document, a few instructions concerning common typographical resources, such as bold or italics, and perhaps how to generate a list, or a footnote. In total, no more than 15 or 20 instructions will allow us to do almost all the things that are done with a word processor. The rest of the instructions allow us to do different things that we normally cannot do with a word processor, or are very difficult to achieve. We can say that while learning to use ConTEXt is more difficult than learning to use a word processor, this is because we can do a lot more with ConTEXt.

1.5.4 Getting help with ConTEXt

While we are new to it, the best place for getting help with ConTEXt is, undoubtedly, on the wiki, which abounds in examples and has a good search engine, especially if one understands English well. We can also find help on the internet, of course, but here the play on words in the name ConTEXt will play tricks on us because searching on the word « context » will return millions of results most of which will have nothing to do with what we are looking for. To find information on ConTEXt you need to add something to the word « context »; for example, « tex », or « Mark IV » or « Hans Hagen » (one of the creators of ConTEXt) or « Pragma ADE », or something similar. It could also be useful to seek information using the wiki name: « contextgarden ».

When we have learned something more about ConTEXt, we can consult some of the many documents included in « ConTEXt Standalone », or even seek help in

TeX – LaTeX Stack Exchange, or on the mailing list for ConTEXt (NTG-context). The latter involves the people who know the most about ConTEXt, but the rules of good cyber-etiquette demand that before asking a question, we should have tried hard beforehand to find the answer ourselves.

II
Global aspects of the document

Chapitre 2
Pages and document pagination

Table of Contents: **2.1 Page size;** 2.1.1 Setting page size; 2.1.2 Using non-standard page sizes; 2.1.3 Changing the page size at any point in the document; 2.1.4 Adjusting the page size to its contents; **2.2 Elements on the page;** **2.3 Page layout (`\setuplayout`);** 2.3.1 Assigning a size to the different page components; 2.3.2 adapting the page layout; 2.3.3 Using multiple page layouts; 2.3.4 Other matters related to page layout; A Distinguishing between odd and even pages; B Pages with more than one column; **2.4 Page numbering;** **2.5 Forced or suggested page breaks;** 2.5.1 The `\page` command; 2.5.2 Joining certain lines or paragraphs to prevent a page break from being inserted between them; **2.6 Headers and footers;** 2.6.1 Commands for establishing the content of headers and footers; 2.6.2 Formatting headers and footers; 2.6.3 Defining specific headers and footers and linking them to section commands; **2.7 Inserting text elements in page edges and margins;**

ConTEXt transforms the source document into correctly formatted *pages*. What these pages look like, how the text and blank spaces are distributed and what elements they have, are all fundamental for good typesetting. This chapter is dedicated to all these questions, and to some other matters relating to pagination.

2.1 Page size

2.1.1 Setting page size

By default, ConTEXt assumes that documents will be of A4 size, the European standard. We can establish a different size with `\setuppapersize` that is the typical command found in the document preamble. The *normal* syntax of this command is:

```
\setuppapersize[LogicalPage][PhysicalPage]
```

where both arguments take symbolic names.[1] The first argument, that I have called *LogicalPage*, represents the page size to be taken into consideration for typesetting; and the second argument, *PhysicalPage*, represents the size of the page it will be printed on. Normally both sizes are the same, and the second argument can then be omitted; however, on occasions the two sizes can be different, as, for example, when printing a book in sheets of 8 or 16 pages (a common printing technique, especially for academic books until approximately the 1960s). In these cases, ConTEXt allows us to distinguish both sizes; and if the idea is that several pages are to be printed on a single sheet of paper, we can also indicate the folding scheme to be followed by using the `\setuparranging` command (which will not be explained in this introduction).

For typesetting size we can indicate any of the predefined sizes used by the paper industry (or by ourselves). This includes:

- Any of the A, B and C series defined by the ISO-216 (from A0 to A10, from B0 to B10 and from C0 to C10), generally in use in Europe.
- Any of the US standard sizes: letter, ledger, tabloid, legal, folio, executive.
- Any of the RA and RSA sizes defined by the ISO-217 standard.
- The G5 and E5 sizes defined by the Swiss SIS-014711 standard (for doctoral theses).
- For envelopes: any of the sizes defined by the North American standard (envelope 9 to envelope 14) or by the ISO-269 standard (C6/C5, DL, E4).
- CD, for CD covers.
- S3 – S6, S8, SM, SW for screen sizes in documents not intended to be printed but shown on screen.

Together with the paper size, with `\setuppapersize` we can indicate page orientation: « portrait » (vertical) or « landscape »(horizontal).

> Other options that `\setuppapersize` allows, according to the ConTEXt wiki, are « `rotated` », « 90 », « 180 », « 270 », « `mirrored` » and « `negative` ». In my own tests I have only noticed some perceptible changes with « `rotated` » that inverts the page,

[1] Recall that in section ?? I indicated that the options taken by ConTEXt commands are basically of two kinds: symbolic names, whose meaning is already known to ConTEXt, or variable that we must explicitly assign some value to.

although it is not exactly an inversion. The numerical values are supposed to produce the equivalent degree of rotation, on their own or in combination with « `rotated` », but I have been unable to get them to work. Nor have I exactly discovered what « `mirrored` » and « `negative` » are for.

The second argument of `\setuppapersize`, that I have already said can be omitted when the print size is no different from the typesetting size, can take the same values as the first, indicating paper size and orientation. It can also take « `oversized` » as a value that – according to ConTEXt wiki – adds a centimetre and a half to each corner of the paper.

According to the wiki there are other possible values for the second argument: « `undersized` », « `doublesized` » and « `doubleoversized` ». But in my own tests I have not seen any change after using any of these; nor does the official definition of the command (see section ??) mention these options.

2.1.2 Using non-standard page sizes

If we want to use a non-standard page size, there are two things we can do:

1. Use an alternative syntax of `\setuppapersize` that allows us to introduce the height and width of the paper as dimensions.
2. Define our own page size, assigning a name to it and using it as if it were a standard paper size.

Alternative syntax of `\setuppapersize`

Other than the syntax we have already seen, `\setuppapersize` allows us to use this other one:

`\setuppapersize`[Name][Options]

where *Name* is an optional argument that represents the name assigned to a paper size with `\definepapersize` (that we will look at next), and *Options* are of the kind where we assign an explicit value. Among the allowable options we can highlight the following:

- **`width, height`** that represent, respectively, the width and height of the page.
- **`page, paper`**. The first refers to the size of the page to be typeset, and the second to the size of the page to be physically printed on. This means that « `page` » is equivalent to the first argument of `\setuppapersize` in its normal syntax (explained above) and « `paper` » to the second argument. These options can take the same values indicated earlier (A4, S3, etc.).

- `scale`, indicates a scaling factor for the page.
- `topspace, backspace, offset,` additional distances.

Defining a customised page size

To define a customised page size, we use the `\definepapersize` command, whose syntax is

```
\definepapersize[Name][Options]
```

where *Name* refers to the name given to the new size and *Options* can be:

- Any of the allowable values for `\setuppapersize` in its normal syntax (A4, A3, B5, CD, etc).
- Measurements of width (of the paper), height (of the paper) and offset (displacement), or a scaled value (« `scale` »).

What is not possible is to mix the values allowed for `\setuppapersize` with measurements or scale factors. This is because in the first case the options are symbolic words and in the second, variables given an explicit value; and in ConTEXt, as I have already said, it is not possible to mix both kinds of options.

When we use `\definepapersize` to indicate a paper size that coincides with some of the standard measurements, in actual fact, rather than defining a new paper size, what we are doing is defining a new name for an already existing paper size. This can be useful if we want to combine a paper size with an orientation. So, for example, we can write

```
\definepapersize[vertical][A4, portrait]
\definepapersize[horizontal][A4, landscape]
```

2.1.3 Changing the page size at any point in the document

In most cases documents only have one page size and this is why `\setuppapersize` is the typical command we include in the preamble and use only once in each document. However, on some occasions it might be necessary to change the size at some point in the document; for example, if from a certain point onwards an annex is included in which the sheets are landscape.

In such cases we can use \setuppapersize at the precise point where we want the change to happen. But since the size would change immediately, to avoid unexpected results we would normally insert a forced page break before \setuppapersize.

If we only need to change the page size for an individual page, instead of using \setuppapersize twice, once to change to the new size, and the second to return to the original size, we can use \adaptpapersize that changes the page size, and, a page later, automatically resets the value prior to it being called. And just the same as we did with \setuppapersize, before using \adaptpapersize we should insert a forced page break.

2.1.4 Adjusting the page size to its contents

There are three environments in ConTEXt that generate pages of the exact size for storing their contents. These are \startMPpage, \startpagefigure and \startTEXpage. The first generates a page that contains a graphic generated with MetaPost, a graphic design language that integrates with ConTEXt, but which I will not deal with in this introduction. The second does the same with an image and perhaps some text beneath it. It takes two arguments: the first identifies the file containing the image. I will deal with this in the chapter dedicated to external images. The third (\startTEXpage) contains the text which is its contents on a page. Its syntax is:

```
\startTEXpage[Options] ... \stopTEXpage
```

where the options can be any of the following:

- **strut**. I am not sure about the usefulness of this option. In ConTEXt terminology, a *strut* is a character lacking width, but with maximum height and depth, but I don't quite see what that has to do with the overall usefulness of this command. According to the wiki this option allows for the values « yes », « no », « global » and « local », and where the default value is « no ».

- **align**. Indicates text alignment. This can be « normal »,« flushleft », « flushright », « middle », « high », « low » or « lohi ».

- **offset** to indicate the amount of white space around the text. It can be « none », « overlay » if an overlay effect is desired, or an actual dimension.

- **`width, height`** where we can indicate a width and height for the page, or the value « `fit` » so that the width and height are those required by the text that is included in the environment.

- **`frame`** that is « `off` » by default but can take the value « `on` » if we want a border around the text on the page.

2.2 Elements on the page

ConTEXt recognises different elements on pages, whose dimensions can be configured with `\setuplayout`. We will look at this immediately, but beforehand it would be best to describe each of the page elements, indicating the name that `\setuplayout` knows each of them by:

- **Edges:** white space surrounding the text area. Although most word processors call them « margins », using ConTEXt's terminology is preferable since it enables us to differentiate between edges as such, where there is no text (although in electronic documents there can be navigation buttons and the like), and margins where certain text elements can sometimes be located, like, for example, margin notes.

  – The height of the upper edge is controlled by two measurements: the upper edge itself (« `top` ») and the distance between the upper edge and the header (« `topdistance` »). The sum of these two measurements is called « `topspace` ».

  – The size of the left and right edges depends on the « `leftedge` » « `rightedge` » measurements respectively. If we want both to be of the same length we can configure them simultaneously with the « `edge` » option.

    In documents intended for double-sided printing, we don't talk about left and right edges but inner and outer ones; the first is the edge closest to the point where the sheets will be stapled or sewn, i.e. the left edge on odd-numbered pages and the right edge on even-numbered pages. The outer edge is the opposite of the inner edge.

  – The height of the lower edge is called « `bottom` ».

- **Margins** properly so called. ConTEXt only calls side margins (left and right) margins. Margins are located between the edge and the main text area and

are intended to host certain text elements such as, for example, margin notes, section titles or their numbers.

The dimensions that control margin size are:

- `margin`: used when we want to simultaneous set the margins at the same size.
- `leftmargin, rightmargin`: set the size of the left and right margins respectively.
- `edgedistance`: Distance between the edge and the margin.
- `leftedgedistance, rightedgedistance`: Distance between the edge and the left and right margins respectively.
- `margindistance`: Distance between the margin and the main text area.
- `leftmargindistance, rightmargindistance`: Distance between the main text area and right and left margins respectively.
- `backspace`: this measurement represents the space between the left corner of the paper and the beginning of the main text area. Therefore it is made up of the sum of « `leftedge` » + « `leftedgedistance` » + « `leftmargin` » + « `leftmargindistance` ».

- **Header and footer:** The header and footer of a page are two areas that are located, respectively, in the top or bottom of the written area of the page.They usually contain information that helps to contextualise the text, such as the page number, the name of the author, the title of the work, the title of the chapter or section, etc. In ConTEXt these areas on the page are affected by the following dimensions:
  - `header`: Height of the header.
  - `footer`: Height of the footer
  - `headerdistance`: Distance between the header and the page's main text area.

- `footerdistance`: Distance between the footer and the page's main text area.

- `topdistance, bottomdistance`: Both mentioned previously. They are the distance between the upper edge and header or the lower edge and footer, respectively.

- **Main text area**: this is the widest area on the page, holding the document's text. It size depends on the « `width` » and « `textheight` » variables. The « `height` » variable, for its part, measures the sum of « `header` », « `headerdistance` », « `textheight` », « `footerdistance` » and « `footer` ».

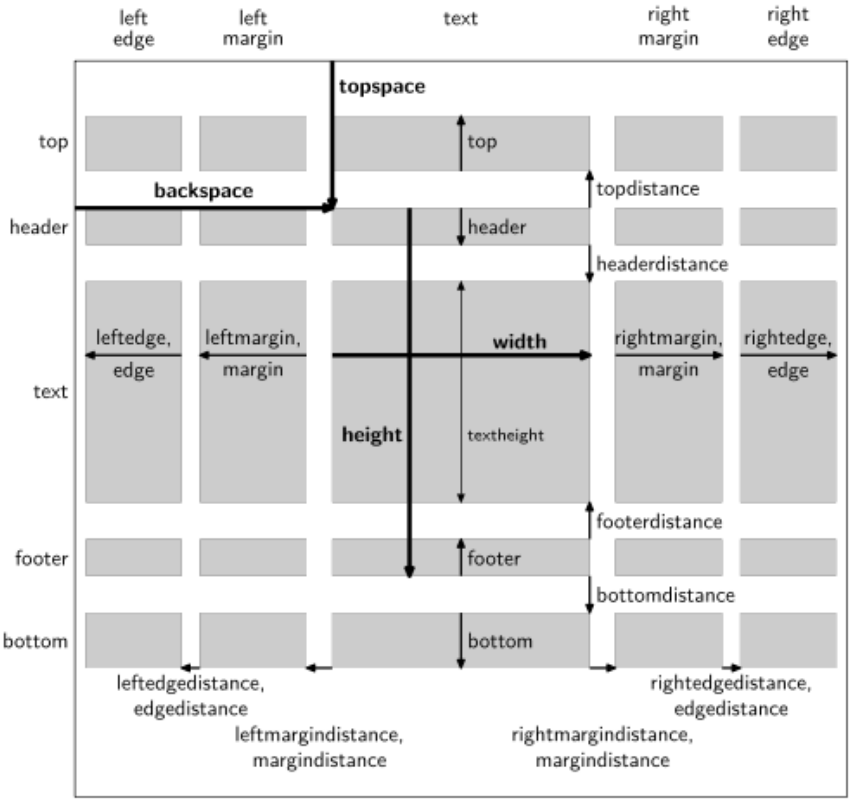


Figure 2.1 Areas and measurements on a page

We can see all these areas in image 2.1 along with the names given to the corresponding measurements, and arrows indicating their extent. The thickness of the arrows together with the size of the names of the arrows are intended to reflect the importance of each of these distances for the page layout. If we look closely, we will see that this image shows that a page can be represented as a table with 9 rows and 9 columns, or, if we do not take into account the separation values between

the different areas, there would be five rows and five columns of which there can only be text in three rows and three columns. The intersection of the middle row with the middle column constitutes the main text area and will normally take up the majority of the page.

In the layout phase of our document, we can see all the page measurements with `\showsetups`. To see the main outlines of text distribution indicated visually on the page, we can use `\showframe`; and with `\showlayouts` we can get a combination of the previous two commands.

2.3 Page layout (`\setuplayout`)

2.3.1 Assigning a size to the different page components

Page design involves assigning specific sizes to the respective areas of the page. This is done with `\setuplayout`. This command allows us to alter any of the dimensions mentioned in the previous section. Its syntax is as follows:

```
\setuplayout[Name] [Options]
```

where *Name* is an optional argument used only for the case where we have designed multiple layouts (see section 2.3.3), and the options are, besides others we will see later, any of the measurements previously mentioned. Bear in mind, however, that these measurements are inter-related since the sum total of the components affecting width, of those affecting height must coincide with the width and height of the page. In principle this will mean that when changing a horizontal length, we must adjust the remaining horizontal lengths; and the same when adjusting a vertical length.

By default, ConTEXt only carries out automatic adjustments of dimensions in some cases which, on the other hand, are not indicated in any complete or systematic way in its documentation. By carrying out several tests I was able to verify, for example, that a manual increase or decrease in the height of the header or footer entails an adjustment in « `textheight` »; however a manual change of some of the margins does not automatically adjust (according to my tests) the text width (« `width` »). This is why the most efficient way to not generate any inconsistency between the page size (set with `\setuppapersize`) and the size of its respective components, is:

- Regarding horizontal measurements:

  - By adjusting « backspace » (that includes « leftedge » and « leftmargin »).
  - By adjusting « width » (text width) not with a dimensions but with the « fit » or « middle » values:
    - fit calculates the width of the text on the basis of the width of the rest of the page's horizontal components.
    - middle does the same, but first makes right and left margins equal.

- Regarding vertical measurements:
  - By adjusting « topspace ».
  - By adjusting the « fit » or « middle » values to « height ». These work the same way as in the case of width. The first calculates the height based on the rest of the components, and the second first makes the upper and lower margins equal, and then calculates the text height.
  - Once « height » is adjusted, by adjusting the height of the header or footer if necessary, knowing that in such cases « textheight » will be automatically readjusted.

- Another possibility for indirectly determining the height of the main text area, is by indicating the number of lines that are to fit in it (bearing in mind the current interline space and font size). This is why \setuplayout includes the « lines » option.

Placing the logical page on the physical page

In the case where the logical page size is not the same as the physical page size (see section 2.1.1) \setuplayout allows us to configure some additional options affecting the placement of the logical page on the physical page:

- **location**: This option determines the place where the page will be placed on the physical page. Its possible values are left, middle, right, top, bottom, singlesided, doublesided or duplex.

- **scale**: Indicates a scaling factor for the page before placing it on the physical page.

- **marking**: Will print visual marks on the page to indicate where the paper is to be cut.

- **horoffset, veroffset, clipoffset, cropoffset, trimoffset, bleedoffset, artoffset**: A series of measures indicating different displacements in the physical page. Most of these are explained in the 2013 reference manual.

These \setuplayout options must be combined with indications from \setuparranging that indicates how logical pages are to be ordered on the physical sheet of paper. I will not explain these commands in this introduction, since I haven't carried out any tests on them.

Getting the width and heights of the text area

The \textwidth and \textheight commands return the width and height of the text area respectively. The values these commands offer cannot be directly shown in the final document, but they can be used for other commands to set their width or height measurements. So, for example, to indicate that we want an image whose width will be 60% of the width of the line, we need to indicate as the value of the image's « width » option: « width=0.6\textwidth ».

2.3.2 adapting the page layout

It could be that our page layout on a particular page produces an undesired result; like, for example, the final page of a chapter with only one or two lines, which is neither typographically or aesthetically desirable. To solve these cases, ConTEXt provides the \adaptlayout command that allows us to alter the size of the text area on one or more pages. This command is intended to be used only when we have already finished writing our document and are making some small final adjustments. Therefore, its natural location is in the preamble to the document. The command's syntax is:

```
\adaptlayout[Pages] [Options]
```

where *Pages* refers to the number of the page or pages whose layout we want to change. It is an optional argument that is to be used only when \adaptlayout is placed in the preamble. We can indicate just one page, or several pages, separating the numbers with comnmas. If we omit this first argument, \adaptlayout will exclusively affect the page on which it finds the command.

As for the options, they can be:

- **height**: Allows us to indicate, as a dimensions, the height the page in question should have. We can indicate an absolute height (e.g. “19cm”) or a relative height (e.g. “+1cm”, “-0.7cm”).

- **lines**: We can include the number of lines to add or subtract. To add lines the value is preceded by a +, and to subtract lines, by the − sign (not just a hyphen).

Consider that when we change the number of lines on a page, this could affect the pagination of the rest of the document. this is why it is recommended that we use `\adaptlayout` only at the end, when the document will not have further changes, and to do it in the preamble. Then we go to the first page we wish to adapt, do so and check how it affects the pages that follow; if it affects it in such a way that another page needs adapting, we add its number and compile once again, and so on.

2.3.3 Using multiple page layouts

If we need to use different layouts in different parts of the document, the best way is to begin by defining the *general* layout and then the various alternative ones, those that only change the dimensions that need to be different. These alternative layouts will inherit all the features of the overall layout which will not change as part of its definition. To specify an alternative layout and give it a name we can later call it with, we use the `\definelayout` command whose general syntax is:

```
\definelayout[Name/Number] [Configuration]
```

where *Name/Number* is the name associated with the new design, or the page number where the new layout will be automatically activated, and *Configuration* will contain the aspects of the layout that we wish to change by comparison with the overall layout.

When the new layout is associated with a name, to call it at a particular point in the document we use:

```
\setuplayout[LayoutName]
```

and to return to the general layout:

```
\setuplayout[reset]
```

If, on the other hand, the new layout was associated with a specific page number, it will be automatically activated when the page is reached. However, once activated,

to return to the general design we will have to explicitly indicate this, even though we can *semi-automate* this. For example, if we want to apply a layout exclusively to pages 1 and 2, we can write in the document's preamble:

```
\definelayout[1][...]
\definelayout[3][reset]
```

The effect of these commands will be that the layout defined in the first line is activated on page 1 and on page 3 another layout is activated the function of which is only to return to the general layout.

With `\definelayout[even]` we create a layout that is activated on all even pages; and with `\definelayout[odd]` the layout will be applied to all odd pages.

2.3.4 Other matters related to page layout

A. Distinguishing between odd and even pages

In double-sided printed documents it is often the case that the header, page numbering and side margins differ between odd and even pages. Even-numbered pages are also called left hand (verso) pages and odd pages, right hand (recto) pages. In these cases it is also usual for the terminology regarding margins to change, and we talk about inner and outer margins. The former is located at the closest point to where the pages will be sewn or stapled and the latter on the opposite side. On odd-numbered pages, the inner margin corresponds to the left margin and on even pages the outer margin corresponds to the right margin.

`\setuplayout` does not have any option expressly allowing us to tell it that we want to differentiate between the layout for odd and even pages. This is because for ConTEXt the difference between both kinds of pages is set with a different option: `\setuppagenumbering` that we will see in section 2.4. Once this has been set, ConTEXt assumes that the page described with `\setuplayout` was the odd page, and builds the even page by applying the inverted values for the odd page to it: the specifications applicable on the odd-numbered page apply to the left, on the even-numbered page they apply to the right; and vice versa: those applicable on the odd-numbered page on the right, apply to the even-numbered page on the left.

B. Pages with more than one column

With `\setuplayout` we can also see that the text of our document is distributed across two or more columns, in the way that newspapers and some magazines do,

for example. This is controlled by the « `columns` » option the value of which has to be a whole number. When there is more than one column, the distance between the columns is indicated by the « `columndistance` » option.

This option is intended for documents in which all the text (or most of it) is distributed across multiple columns. If, in a document that is mainly a one column document, we want a particular part to be two or three columns, we do not need to alter the page layout but simply use the « `columns` » environment (see section ??).

2.4 Page numbering

By default, ConTEXt uses Arabic numbers for page numbering and the number appears centred in the header. To alter these features, ConTEXt it has different procedures which, in my opinion, make it unnecessarily complex where this matter is concerned.

Firstly, the fundamental characteristics of numbering are controlled by two different commands: `\setuppagenumbering` and `\setupuserpagenumber`.

`\setuppagenumbering` allows the following options:

- **`alternative`**: This option controls whether the document is designed so that the header and footer are identical on all pages (« `singlesided` »), or whether they differentiate odd and even pages (« `doublesided` »). When this option takes the latter value, automatically the page layout values introduced by « `setuplayout` » are affected, so that it is assumed that what is indicated in « `setuplayout` » refers only to odd-numbered pages, and therefore what is arranged for the left margin actually refers to the inner margin (which on even-numbered pages is on the right) and that what is arranged for the right side actually refers to the outer margin, which on even-numbered pages is on the left.

- **`state`**: Indicates whether or not the page number will be displayed. It allows two values: start (page number will be displayed) and stop (page numbers will be suppressed). The name of these values (start and stop) could make us think that when we have « `state=stop` » pages stop being numbered, and when « `state=start` » numbering begins again. But this is not so: these values only affect whether the page number is shown or not.

- **`location`**: indicates where it will be displayed. Normally we need to indicate two values in this option, separated by a comma. First of all we need to specify

if we want the page number in the header (« `header` ») or the footer (« `footer` »), and then, where in the header or footer: it could be « `left` », « `middle` », « `right` », « `inleft` », « `inright` », « `margin` », « `inmargin` », « `atmargin` » or « `marginedge` ». For example: to show right-aligned numbering in the footer we should indicate « `location={footer,right}` ». See, on the other hand, how we have surrounded this option with curly brackets so ConTEXt can correctly interpret the separating comma.

- **`style`**: indicates font size and style to be used for page numbers.
- **`color`**: Indicates the colour to be applied to the page number.
- **`left`**: picks up the command or text to be executed to the left of the page number.
- **`right`**: picks up the command or text to be executed to the right of the page number.
- **`command`**: picks up a command to which the page number will be passed as a parameter.
- **`width`**: indicates the width taken up by the page number.
- **`strut`**: I am not so sure about this. In my tests, when « `strut=no` », the number is printed exactly on the upper edge of the header or on the bottom of the footer, while when « `strut=yes` » (default value) a space is applied between the number and the edge.

`\setupuserpagenumber`, allows these extra options:

- **`numberconversion`**: controls the kind of numbering that can be arabic (« `n` », « `numbers` »), lower case (« `a` », « `characters` »), upper case (« `A` »,« `Characters` »), small caps (« `KA` »), lower case roman (« `i` », « `r` », « `romannumerals` »), uppercase roman (« `I` », « `R` », « `Romannumerals` ») or small caps roman (« `KR` »).
- **`number`**: indicates the number to assign to the first page, on the basis of which the rest will be calculated.
- **`numberorder`**: if we assign « `reverse` » to this as a value, page numbering will be in decreasing order; this means the last page will be 1, the second-last 2, etc.
- **`way`**: allows us to indicate how numbering will proceed. It can be: byblock, bychapter, bysection, bysubsection, etc.

- **prefix**: allows us to indicate a prefix to page numbers.
- **numberconversionset**: Explained in what follows.

In addition to these two commands, it is also necessary to take into account the control of numbers involving the document's macrostructure (see section ??). From this point of view, \defineconversionset allows us to indicate a different kind of numbering for each of the macrostructure blocks. For example:

```
\defineconversionset
  [frontpart:pagenumber][][romannumerals]

\defineconversionset
  [bodypart:pagenumber][][numbers]

\defineconversionset
  [appendixpart:pagenumber][][Characters]
```

will see that the first block in our document (frontmatter) is numbered with lower case Roman numbers, the central block (bodymatter) with Arabic numbers and the appendices with upper case letters.

We can use the following commands to get the page number:

- \userpagenumber: returns the page number just as it was configured with \setuppagenumbering and with \setupuserpagenumber.
- \pagenumber: returns the same number as the previous command but still in Arabic numbers.
- \realpagenumber: returns the real number of the page in Arabic numbers without taking any of these specifications into account.

To get the number of the final page in the document there are three commands that are parallel to the previous ones. They are: \lastuserpagenumber,\lastpagenumber and \lastrealpagenumber.

2.5 Forced or suggested page breaks

2.5.1 The \page command

The algorithm for text distribution in ConTEXt is quite complex, and is based on a multitude of calculations and internal variables that tell the program where the

best possible point is for introducing an actual page break from the perspective of typographical correctness. The \page command allows us to influence this algorithm:

a. By suggesting certain points as the best or the most inappropriate place for including a page break.

   - **no**: indicates that the place where the command is located is not a good candidate for inserting a page break, so, as far as possible, the break should be made at another point in the document.
   - **preference**: tells ConTEXt that the point where it encounters the command is a *good place* for attempting a page break, although it will not force one there.
   - **bigpreference**: indicates that the point where it encounters the command is a *very good place* for attempting a page break, but it too does not go as far as forcing it.

   Note that these three options neither force nor prevent page breaks, but only tell ConTEXt that when looking for the best place for a page break, it should take into account what is indicated in this command. In the final instance, however, the place where the page break will happen will continue to be decided by ConTEXt.

b. By forcing a page break at a certain point; in this case we can also indicate how many page breaks should be made as well as certain features of the pages to be inserted.

   - **yes**: force a page break at this point.
   - **makeup**: similar to « yes », but the forced break is immediate, without first placing any floating objects whose placement is pending (see section ??).
   - **empty**: insert a completely blank page in the document.
   - **even**: insert as many pages as necessary to make the next page an even page.
   - **odd**: insert as many pages as necessary to make the next page an odd page.
   - **left, right**: similar to the two previous options, but applicable only to double-sided printed documents, with different headers, footers or margins depending on whether the page is odd or even.
   - **quadruple**: insert the number of pages needed for the next page to be a multiple of 4.

Along with these options which specifically control pagination, \page includes other options that affect the way this command functions. Especially the « disable » option that causes ConTEXt to ignore the \page commands it finds from there on, and the « reset » option that produces the opposite effect, restoring the effectiveness of future \page commands.

2.5.2 Joining certain lines or paragraphs to prevent a page break from being inserted between them

Sometimes, if we want to prevent a page break between several paragraphs, the use of the \page command can be laborious, as it would have to be written at every point where it was possible for a page break to be inserted. A simpler procedure for this is to place the material we want to keep on the same page in what TEX calls a *vertical box*.

> At the beginning of this document (on page 20) I indicated that internally, everything is a *box* for TEX. The box notion is fundamental in TEX for any kind of *advanced* operation; but managing it is too complex to include in this introduction. This is why I only make occasional references to boxes.

TEX boxes, once created, are indivisible, meaning that we cannot insert a page break that would split a box in two. This is why, if we put the material we want kept together in an invisible box, we avoid a page break being inserted that would split this material. The command for doing this is \vbox, the syntax for which is

```
\vbox{Material}
```

where *Material* is the text we want to keep together.

Some of ConTEXt's environments do put their contents in a box. For example, « framedtext », so if we frame the material we want kept together in this environment and also see that the frame is invisible (which we do with the frame=off option), we will have achieved the same thing.

2.6 Headers and footers

2.6.1 Commands for establishing the content of headers and footers

If we have assigned a certain size to the header and footer in page layout, we can include text in them with the \setupheadertexts and \setupfootertexts commands. The two commands are similar, the only difference being that the former

activates header content and the latter the footer content. Both have from one to five arguments.

1. Used with a single argument this will contain the text of the header or footer that will be placed in the centre of the page. For example: `\setupfooter-texts[pagenumber]` will write the page number at the centre of the footer.

2. Used with two arguments, the content of the first argument will be placed on the left side of the header or footer, and that of the second argument on the right side. For example `\setupheadertexts[Preface][pagenumber]` will typeset a page header in which the word « preface » is written on the left side and the page number is printed on the right side.

3. If we use three arguments, the first will indicate *the area* in which the other two are to be printed. By *area* I am referring to the *areas* of the page mentioned in section 2.2, in other words: edge, margin, header... The other two arguments contain the text to be placed in the left edge, margin and right edge, margin.

Using it with four or five arguments is equivalent to using it with two or three arguments, in cases where a distinction is made between even and odd pages, which occurs, as we know, when « `alternative=doublesided` » with `\setup-pagenumbering` has been set. In this case, two possible arguments are added to reflect the content of the left and right sides of the even pages.

An important characteristic of these two commands is that when they are used with two arguments, the previous central header or footer (if it existed) is not rewritten, which allows us to write a different text in each area as long as we first write the central text (calling the command with a single argument) and then write the texts for either side (calling it again, now with two arguments). So, for example, if we write the following commands

```
\setupheadertexts[and]
\setupheadertexts[Tweedledum][Tweedledee]
```

The first command will write « and » in the centre of the header and the second will write « Tweedledum » on the left and « Tweedledee » on the right, leaving the centre area untouched, since it has not been ordered to be rewritten. The resulting header will now show up as

Tweedledum and Tweedledee

> The explanation I have just given of the operation of these commands is my conclusion after many tests. The explanation of these commands provided in ConTEXt *excursion* is

based on the version with five arguments; and the one in the 2013 reference manual is based on the version with three arguments. I think mine is clearer. On the other hand, I have not seen an explanation of why the second command call does not overwrite the previous call, but this is how it works if we first write the central item in the header or footer and then the ones either side. But if we write the items either side first in the header/footer, the subsequent call to the command to write the central item will delete the previous headers or footers. Why? I have no idea. I think these small details introduce unnecessary complication and should be clearly explained in the official documentation.

Moreover, we can indicate any combination of text and commands as the actual content of the header or footer. But also the following values:

- **`date, currentdate`**: will write (either of them) the current date.
- **`pagenumber`**: will write the page number.
- **`part, chapter, section...`**: will write the title corresponding to part, chapter, section... or whatever structural division there is.
- **`partnumber, chapternumber, sectionnumber...`**: will write the number of the part, chapter, section... or whatever structural division there is.

Attention: These symbolic names (`date, currentdate, pagenumber, chapter, chapternumber`, etc.) are only interpreted as such if the symbolic name itself is the only content of the argument; but if we add some other text or formatting command, these words will be interpreted literally, and so, for example, if we write `\setupheadertexts[chapternumber]` we will get the number of the current chapter; but if we write `\setupheadertexts[Chapter chapternumber]` we will end up with: « Chapter chapternumber ». In these cases, when the content of the command is not just the symbolic word, we must:

- For `date, currentdate` and `pagenumber` use, not the symbolic word but the command with the same name (`\date`, `\currentdate` or `\pagenumber`).
- For `part, partnumber, chapter, chapternumber`, etc. use the `\getmarking[Mark]` command that returns the contents of the *Mark* that is asked for. So, for example, `\getmarking[chapter]` will return the title of the current chapter, while `\getmarking[chapternumber]` will return the number of the current chapter.

To disable headers and footers on a particular page, use the `\noheaderandfooterlines` command that acts exclusively on the page where it is located. If we only want to delete the page number on a particular page, we must use the `\page[blank]` command.

2.6.2 Formatting headers and footers

The specific format in which the text of the header or footer is shown can be indicated in the arguments for `\setupheadertexts` or `\setupfootertexts` by using the corresponding format commands. However, we can also configure this globally with `\setupheader` and `\setupfooter` that allow the following options:

- **`state`**: allows for the following values: `start`, `stop`, `empty`, `high`, `none`, `normal` or `nomarking`.
- **`style, leftstyle, rightstyle`**: configuration of the header and footer text style. `style` affects all pages, `leftstyle` the even pages and `rightstyle` the odd pages.
- **`color, leftcolor, rightcolor`**: header or footer colour. It can affect all pages (`color` option) or only the even pages (`leftcolor`) or odd pages (`rightcolor`)
- **`width, leftwidth, rightwidth`**: width of all headers and footers (`width`) or headers/footers on even pages (`leftwidth`) or odd ones (`rightwidth`).
- **`before`**: command to be executed before writing the header or footer.
- **`after`**: command to be executed after writing the header or footer.
- **`strut`**: if « yes », a vertical separation space is established between the header and the edge. When it is « no », the header or footer runs up against the edges of the upper or lower edge areas.

2.6.3 Defining specific headers and footers and linking them to section commands

ConTEXt's header and footer system allows us to automatically change the text in the header or footer when we change chapters or sections; or when we change pages, if we have set different headers or footers for odd and even pages. But what it does not allow is to differentiate between the first page (of the document, or of a chapter or section) and the rest of the pages. To achieve the latter we must:

1. Define a specific header or footer.
2. Link it to the section it applies to.

The definition of specific headers or footers is done with the `\definetext` command, whose syntax is:

```
\definetext
  [Name] [Type]
  [Content1] [Content2] [Content3]
  [Content4] [Content5]
```

where *Name* is the name assigned to the header or footer we are dealing with; *Type* can be `header` or `footer`, depending on which of the two we are defining, and the remaining five arguments contain the contents we want for the new header or footer, in a similar way to how we have seen `\setupheadertexts` and `\setupfootertexts` function. Once we have done this, we need to link the new header or footer to some particular section with `\setuphead` by using the *header* and `footer` options (that are not explained in Chapter ??).

Thus, the following example will hide the header on the first page of each chapter and a centred page number will appear as the footer:

```
\definetext[ChapterFirstPage] [footer] [pagenumber]
\setuphead
  [chapter]
  [header=high, footer=ChapterFirstPage]
```

2.7 Inserting text elements in page edges and margins

The top and bottom edges and the right and left margins usually do not contain text of any kind. However, ConTEXt allows some text elements to be placed there. In particular, the following commands are available for this purpose:

- `\setuptoptexts`: allows us to place text at the top edge of the page (above the header area).
- `\setupbottomtexts`: allows us to place text at the bottom edge of the page (below the footer area).
- `\margintext`, `\atleftmargin`, `\atrightmargin`, `\ininner`, `\ininneredge`, `\ininnermargin`, `\inleft`, `\inleftedge`, `\inleftmargin`, `\inmargin`, `\inother`, `\inouter`, `\inouteredge`, `\inoutermargin`, `\inright`, `\inrightedge`, `\inrightmargin`: allow us to place text in the side edges and margins of the document.

The first two commands function exactly like \setupheadertexts and \setupfootertexts, and the format of these texts can even be configured in advance with \setuptop and \setupbottom similar to how \setupheader allows us to configure the texts for \setupheadertexts. For all this I refer to what I have already said in section 2.6. The only little detail that needs to be added is that the text set up for \setuptoptexts or \setupbottomtexts will not be visible if no space has been reserved in the page layout for the upper (top) or lower (bottom) edges. For this, see section 2.3.1.

As for the commands aimed at placing text in the margins of the document, they all have a similar syntax:

```
\CommandName[Reference][Configuration]{Text}
```

where *Reference* and *Configuration* are optional arguments; the first is used for possible cross-referencing and the second allows us to set up the marginal text. The last argument, enclosed in curly brackets, contains the text to be placed in the margin.

Of these commands, the more general one is \margintext as it allows text to be placed in any of the margins or side edges of the page. The remaining commands, as their name indicates, place the text in the margin itself (right or left, inner or outer), or the edge (right or left, inner or outer). These commands are closely related to page layout because if, for example, we use \inrightedge but have not reserved any space in the page layout for the right edge, nothing will be seen.

The configuration options for \margintext are as follows:

- **location**: indicates what margin the text will be placed in. It can be left, right or, in double-sided documents, outer or inner. By default it is left in single-sided documents and outer in double-sided ones.
- **width**: width available for printing the text. By default, the full width of the margin will be used.
- **margin**: indicates whether the text will be placed in the margin itself or in the edge.
- **align**: text alignment. The same values are used here as in \setupalign ??.
- **line**: allows us to indicate a number of lines of displacement of the text in the margin. So, line=1 will displace the text by one line below and line=-1 by one line above.

- **style**: command or commands for indicating the style of text to be placed in the margins.
- **color**: the colour of marginal text.
- **command**: name of a command to which the text to be placed in the margin will be passed as an argument. This command will be executed before writing the text. For example, if we want to draw a frame around the text, we could use « [command=\framed]{Text} ».

The remaining commands allow the same options, except for location and margin. In particular, the \atrightmargin and \atleftmargin commands place the text completely attached to the body of the page. We can establish a separation space with the distance option, which I did not mention when talking about \margintext because I saw no effect on that command in my tests.

In addition to the above options, these commands also support other options (strut, anchor, method, category, scope, option, hoffset, voffset, dy, bottomspace, threshold and stack) that I have not mentioned because they are not documented and frankly, I am not very sure what they are for. Ones with names like *distance* we can guess, but the rest? The wiki only mentions the stack option, saying that it is used to emulate the \marginpars command in LaTeX, but this does not seem very clear to me.

The \setupmargindata command allows us to globally configure the texts in each margin. So, for example,

```
\setupmargindata[right][style=slanted]
```

will ensure that all texts in the right margin are written in slanted style.

We can also create our own customised command with

```
\definemargindata[Name][Configuration]
```

III
Particular issues

Chapitre 3
Characters, words, text and horizontal space

Table of Contents: **3.1 Getting characters not normally accessible from the keyboard;** 3.1.1 Diacritics and special letters; 3.1.2 Traditional ligatures; 3.1.3 Greek letters; 3.1.4 Various symbols; 3.1.5 Defining characters ; 3.1.6 Use of predefined symbol sets; **3.2 Special character formats;** 3.2.1 Upper case, lower case and fake small caps; 3.2.2 Superscript or subscript text; 3.2.3 Verbatim text; **3.3 Character and word spacing;** 3.3.1 Automatically setting horizontal space; 3.3.2 Altering the space between characters within a word; 3.3.3 Commands for adding horizontal space between words; **3.4 Compound words;** **3.5 The language of the text;** 3.5.1 Setting and changing the language; 3.5.2 Configuring the language; 3.5.3 Labels associated with particular languages; 3.5.4 Some language-related commands; A Date-related commands; B The `\translate` command; C The `\quote` and `\quotation` commands;

The basic core element of all text documents is the character: characters are grouped into words, which in turn form lines that make up the paragraphs that make up pages.

The current chapter, starting with « *character* » explains some of ConTEXt's utilities relating to characters, words and text.

3.1 Getting characters not normally accessible from the keyboard

In a text file encoded as UTF-8 (see section ??) we can use any character or symbol, both of living languages and of many already extinct. But, as the possibilities of a keyboard are limited, most of the characters and symbols allowed in UTF-8 normally cannot be obtained directly from the keyboard. This is particularly the case with many diacritics, i.e. signs placed above (or below) certain letters, giving them a special value; but also with many other characters like maths symbols, traditional ligatures, etc. We can obtain many of these characters with ConTEXt by using commands.

3.1.1 Diacritics and special letters

Almost all Western languages have diacritics (with the important exception of English for the most part) and in general, keyboards can generate the diacritics corresponding to regional languages. Thus, a Spanish keyboard can generate all the diacritics needed for Spanish (basically accents and diaeresis) as well as some diacritics used in others languages such as Catalan (grave accents and cedillas) or French (cedillas, grave and circumflex accents); but not, for for example, some that are used in Portuguese, such as the tilde on some vowels in words like « navegação ».

TEX was designed in the United States where keyboards generally do not enable us to get diacritics; so Donald Knuth gave it a set of commands that enable us to obtain almost all the known diacritics (at least in languages using the Latin alphabet). If we use a Spanish keyboard, it does not make much sense to use these commands to obtain the diacritics that can be obtained directly from the keyboard. It is still important to know that these commands exist, and what they are, since Spanish (or Italian, or French...) keyboards do not let us generate all possible diacritics.

Name	Character	Abbreviation	Command
Acute accent	ú	`\'u`	`\uacute`
Grave accent	ù	`\`u`	`\ugrave`
Circumflex accent	û	`\^u`	`\ucircumflex`
Dieresis or umlaut	ü	`\"u`	`\udiaeresis, \uumlaut`
Tilde	ũ	`\~u`	`\utilde`
Macron	ū	`\=u`	`\umacron`
Breve	ŭ	`\u u`	`\ubreve`

Tableau 3.1 Accents and other diacritics

In table 3.1 we find the commands and abbreviations that allow us to obtain these diacritics. In all cases it is unimportant whether we use the command or the abbreviation. In the table, I have used the letter «u» as an example, but these commands work with any vowel (most of them[1]) and also with some consonants and some semivowels.

[1] Of the commands found in table 3.1 the tilde does not work with the letter «e», and I don't know why.

- As most of the abbreviated commands are *control symbols* (see section ??), the letter on which the diacritic is to fall can be written immediately after the command, or separated from it. So, for example: to get the Portuguese «ã» we can write the `\=a` or `\=␣a` characters.[1] But in the case of the breve (`\u`), when dealing with a *control word* the blank space is obligatory.

- In the case of the long version of the command, the letter on which the diacritic falls will be the first letter of the command name. So, for example `\emacron` will place a macron above a lower case «e» (ē), `\Emacron` will do the same above an upper case «E» (Ē), while `\Amacron` will do the same above an upper case «A» (Ā).

While the commands in table 3.1 work with vowels and some consonants, there are other commands to generate some diacritics and special letters which only work on one or several letters. They are shown in table 3.2.

Name	Character	Abbreviation	Command
Scandinavian O	ø, Ø	`\o, \O`	
Scandinavian A	å, Å	`\aa, \AA, {\r a}, {\r A}`	`\aring, \Aring`
Polish L	ł, Ł	`\l, \L`	
German Eszett	ß	`\ss, \SS`	
«i» and «j» wihtout a point	ı, ȷ	`\i, \j`	
Hungarian Umlaut	ű, Ű	`\H u, \H U`	
Cedilla	ç, Ç	`\c c, \c C`	`\ccedilla, \Ccedilla`

Tableau 3.2 More diacritics and special letters

I would like to point out that some of the commands in the above table generate the characters from other characters, while other commands only work if the font we are using has expressly provided for the character in question. So where German Eszett (ß) is concerned, the table shows two commands but only one character, because the font I am using here for this text only provides for the upper case version of German Eszett (something quite common).

That's probably why I can't get the Scandinavian A in upper case either although « `{\r A}` » and `\Aring` work correctly.

The Hungarian umlaut also works with the letter «o», and the cedilla with the letters «k», «l», «n», «r», «s» and «t», in lower or upper case, respectively. The commands to be used are `\kcedilla`, `\lcedilla`, `\ncedilla` ... respectively.

[1] Remember that in this document we are representing blank spaces, when it is important that we see them, with the «␣».

3.1.2 Traditional ligatures

A ligature is formed by the union of two or more graphemes that are usually written separately. This « fusion » between two characters often started out as a kind of shorthand in handwritten texts, until finally they achieved a certain typographic independence. Some of them were even included among the characters that are usually defined in a typographic font, such as the ampersand, «&», which began as a contraction of the Latin copula (conjunction) « et », or the German Eszett (ß), which, as its name indicates, began as a combination of an «s» and «z». In some font designs, even today, we can trace the origins of these two characters; or maybe I see them because I know they're there. In particular, with the Pagella font for «&» and with Bookman for «ß».

As an exercise I suggest (after reading Chapter ??, where it explains how to do it) try representing these characters with these fonts at a size large enough (for example, 30 pt) to be able to work out their components.

Other traditional ligatures which did not become so popular, but are still used occasionally today, are the Latin endings « oe » and « ae » which were occasionally written as «œ» or «ae» to indicate that they formed a diphthong in Latin. These ligatures can be achieved in ConTEXt with the commands found in table 3.3

Ligature	Abbreviation	Command
æ, Æ	`\ae, \AE`	`\aeligature, \AEligature`
œ, Œ	`\oe, \OE`	`\oeligature, \OEligature`

Tableau 3.3 Traditional ligatures

A ligature that used to be traditional in Spanish (Castilian) and that is not usually found in fonts today, is «Ð»: a contraction involving «D» and «E». As far as I know there is no command in ConTEXt that lets us use this,[1] but we can create one, as explained in section 3.1.5.

Along with the previous ligatures, which I have called *traditional* because they come from handwriting, after the invention of the printing press certain printed text ligatures developed which I will call « typographical ligatures » considered by ConTEXt to be font utilities and which are managed automatically by the program, although we can influence how these font utilities are handled (including ligatures) with `\definefontfeature` (not explained in this introduction).

[1] In LaTeX, by contrast, we can use the `\DH` command implemented by the « `fontenc` » package.

3.1.3 Greek letters

It is common to use Greek characters in mathematical and physics formulas. This is why ConTEXt included the possibility of generating all of the Greek alphabet, upper and lower case. Here the command is built on the English name for the Greek letter in question. If the first character is written in lower case we will have the lower case Greek letter and if it is written in capital letters we will get the Greek letter in upper case. For example, the command `\mu` will generate the lower case version of this letter (μ) while `\Mu` will generate the upper case version (M). In table 3.4 we can see which command generates each of the letters in the Greek alphabet, lower case and upper case.

English name	Character (lc/uc)	Commands (lc/uc)
Alpha	α, A	`\alpha, \Alpha`
Beta	β, B	`\beta, \Beta`
Gamma	γ, Γ	`\gamma, \Gamma`
Delta	δ, Δ	`\delta, \Delta`
Epsilon	ϵ, ε, E	`\epsilon, \varepsilon, \Epsilon`
Zeta	ζ, Z	`\zeta, \Zeta`
Eta	η, H	`\eta, \Eta`
Theta	θ, ϑ, Θ	`\theta, \vartheta, \Theta`
Iota	ι, I	`\iota, \Iota`
Kappa	κ, $\varkappa$, K	`\kappa, \varkappa, \Kappa`
Lambda	λ, Λ	`\lambda, \Lambda`
Mu	μ, M	`\mu, \Mu`
Nu	ν, N	`\nu, \Nu`
Xi	ξ, Ξ	`\xi, \Xi`
Omicron	o, O	`\omicron, \Omicron`
Pi	π, ϖ, Π	`\pi, \varpi, \Pi`
Rho	ρ, ϱ, P	`\rho, \varrho, \Rho`
Sigma	σ, ς, Σ	`\sigma, \varsigma, \Sigma`
Tau	τ, T	`\tau, \Tau`
Ypsilon	υ, Y	`\upsilon, \Upsilon`
Phi	ϕ, φ, Φ	`\phi, \varphi, \Phi`
Chi	χ, X	`\chi, \Chi`
Psi	ψ, Ψ	`\psi, \Psi`
Omega	ω, Ω	`\omega, \Omega`

Tableau 3.4 Greek alphabet

Note how for lower case versions of some characters (epsilon, kappa, theta, pi, rho, sigma and phi) there are two possible variants.

3.1.4 Various symbols

Together with the characters we have just seen, TEX (and therefore ConTEXt as well) offers commands for generating any number of symbols. There are many such commands. I have provided an extended although incomplete list in appendix ??.

3.1.5 Defining characters

If we need to use any characters not accessible from our keyboard, we can always find a web page with these characters and copy them into our source file. If we are using UTF-8 encoding (as recommended) this will almost always work. But also in the ConTEXt wiki there is a page with heaps of symbols that can be simply copied and pasted into our document. To get them, click on this link.

However, if we need to use one of the characters in question more than once, then copy-paste is not the most efficient way to do so. It would be preferable to define the character so that it is associated with a command that will generate it each time. To do this we use `\definecharacter` whose syntax is:

`\definecharacter` *`Name Character`*

where

- **Name** is the name associated with the new character. It should not be the name of an existing command, as this would overwrite that command.
- **Character** is the character generated each time we run `\`*`Name`*. There are three ways we can indicate this character:
  - By simply writing it or pasting it into our source file (if we have copied it from another electronic document or web page).
  - By indicating the number associated with that character in the font we are currently using. In order to see the characters included in the font, and the numbers associated with them, we can use the `\showfont[`*`Font name`*`]` `command`.
  - Building the new character with one of the composite character building commands that we will see immediately following.

As an example of the first usage, let's return for the moment to the sections dealing with ligatures (3.1.2). There I spoke about a traditional ligature in Spanish that

we can't usually find in fonts today: «Ð». We could associate this character, for example, with the `\decontract` command so that the character will be generated whenever we write `\decontract`. We do this with:

```
\definecharacter decontract Ð
```

> To build a new character that is not in our font, and cannot be obtained from the keyboard, as is the case of the example I have just given, first we must find some text where that character is found, copy it and be able to paste it into our definition. In the actual example I have just given, I originally copied the «Ð» from Wikipedia.

ConTEXt also includes some commands that allow us to create composite characters and that can be used in combination with `\definecharacter`. By composite characters I mean characters that also have diacritics. The commands are as follows:

```
\buildmathaccent Accent Character
\buildtextaccent Accent Character
\buildtextbottomcomma Character
\buildtextbottomdot Character
\buildtextcedilla Character
\buildtextgrave Character
\buildtextmacron Character
\buildtextognek Character
```

For example: as we already know, by default ConTEXt only has commands for writing certain letters with a cedilla (c, k, l, n, r, s y t) that are usually incorporated into fonts. If we wanted to use a «b» we could use the `\buildtextcedilla` command as follows:

```
\definecharacter bcedilla {\buildtextcedilla b}
```

This command will create the new `\bcedilla` command that will generate a «b» with a cedilla: «ḇ». These commands literally « build » the new character that will be generated even though our font doesn't have it. What these commands do is to superimpose one character over another then give a name to that superimposition.

> In my tests I was unable to make `\buildmathaccent` or `\buildtextognek` work. So I will no longer mention them from here on.

`\buildtextaccent` takes two characters as arguments and superimposes one on the other, raising one of them slightly. Although it is called « buildtextaccent », it is not essential that any of the characters taken as arguments is an accent; but the overlap will give better results if it is, because in this case, by superimposing the accent on the character the accent is less likely to overwrite the character. On

the other hand, the overlapping of two characters that have the same baseline under normal conditions is affected by the fact that the command slightly raises one of the characters above the other. This is why we cannot use this command, for example, to get the contraction «Ð» mentioned above, because if we write

```
\definecharacter decontract {\buildtextaccent D E}
```

in our source file, the slight elevation above the «D» baseline that this command produces means that the (« Ɇ ») effect it produces is not very good. But if the height of the characters allows it we could create a combination. For example,

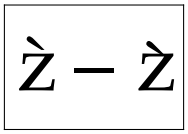
```
\definecharacter unusual {\buildtextaccent \_ "}
```

would define the «z̲"» character that would be associated with the `\unusual` command.

The rest of the build commands takes a single argument – the character that the diacritic generated by each command will be added to. Below I will show an example of each of them, built on the letter «z»:

- `\buildtextbottomcomma` adds a comma beneath the character it takes as an argument («z̦»).
- `\buildtextbottomdot` adds a point beneath the character it takes as an argument («ẓ»).
- `\buildtextcedilla` adds a cedilla beneath the character it takes as an argument («ȥ»).
- `\buildtextgrave` adds a grave accent above the character it takes as an argument («ż»).
- `\buildtextmacron` adds a small bar beneath the character it takes as an argument («z̲»).

At first sight, `\buildtextgrave` seems redundant given that we have `\buildtextaccent`; However, if you check the grave accent generated with the first of these two commands, it looks a little better. The following example shows the result of both commands, at a sufficient font size to appreciate the difference:



3.1.6 Use of predefined symbol sets

« ConTEXt Standalone » includes, along with ConTEXt itself, a number of predefined symbol sets we can use in our documents. These sets are called « `cc` », « `cow` », « `fontawesome` », « `jmn` », « `mvs` » and « `nav` ». Each of these sets also includes some subsets:

- **`cc`** includes « cc ».
- **`cow`** includes « cownormal » and « cowcontour ».
- **`fontawesome`** includes « fontawesome ».
- **`jmn`** includes « navigation 1 », « navigation 2 », « navigation 3 » and « navigation 4 ».
- **`mvs`** includes « astronomic », « zodiac », « europe », « martinvogel 1 », « martinvogel 2 » and « martinvogel 3 ».
- **`nav`** includes « navigation 1 », « navigation 2 » and « navigation 3 ».

> The wiki also mentions a set called **`was`** that includes « wasy general », « wasy music », « wasy astronomy », « wasy astrology », « wasy geometry », « wasy physics » and « wasy apl ». But I couldn't find them in my distribution, and my tests to attempt to get at them failed.

To see the specific symbols contained in each of these sets, the following syntax is used:

```
\usesymbols[Set]
\showsymbolset[Subset]
```

For example: if we want to see the symbols included in « mvs/zodiac », then in the source file we need to write:

```
\usesymbols[mvs]
\showsymbolset[zodiac]
```

and we will get the following result:

```
Aquarius
Aries
Cancer
```

```
Capricorn
Gemini
Leo
Libra
Pisces
Sagittarius
Scorpio
Taurus
Virgo
```

Note that the name of each symbol is indicated as well as the symbol. The \symbol command allows us to use any of the symbols. Its syntax is:

```
\symbol[Subset][SymbolName]
```

where subset is one of the subsets associated with any of the sets we have previously loaded with \usesymbols. For example, if we wanted to use the astrological symbol associated with Aquarius (found in mvs/zodiac) we would need to write

```
\usesymbols[mvs]
\symbol[zodiac][Aquarius]
```

which will give us the « », and this, for all intents and purposes, will be treated as a « character » and is therefore affected by the font size that is active when printed. We can also use \definecharacter to associate the symbol in question with a command. For example

```
\definecharacter Aries {\symbol[zodiac][Aries]}
```

will create a new command called \Aries that will generate the character « ».

We could also use these symbols, for example, in an itemize environment. For example:

```
\usesymbols[mvs]
\definesymbol[1][{\symbol[martinvogel 2][PointingHand]}]
\definesymbol[2][{\symbol[martinvogel 2][CheckedBox]}]
\startitemize[packed]
\item item \item item
 \startitemize[packed]
 \item item \item item
 \stopitemize
\item item
\stopitemize
```

will produce

item
item
item
item
item
item

3.2 Special character formats

Strictly speaking, it is *format* commands that affect the font used, its size, style or variant. These commands are explained in Chapter ??. However, seen more *broadly*, we can also consider the commands that somehow change the characters they take as an argument (thus altering their appearance) to be format commands. We will look at some of these commands in this section. Others, such as underlined or lined text with lines above or below the text (e.g. where we want to provide space to answer a question) will be seen in section ??.

3.2.1 Upper case, lower case and fake small caps

Letters themselves can be upper case or lower case. For ConTEXt, upper case and lower case letters are different characters, so in principle it will typeset the letters just as it finds them written. However, there is a group of commands which allow us to ensure that the text they take as an argument is always written in upper or lower case:

- `\word{text}`: converts the text taken as an argument into lower case.
- `\Word{text}`: converts the first letter of the text taken as an argument into upper case.
- `\Words{text}`: converts the first letter of each of the words taken as an argument into upper case; the rest are in lower case.
- `\WORD{text}` or `\WORDS{text}`: writes the text taken as an argument in upper case.

Very similar to these commands are `\cap` and `\Cap`: they also capitalise the text they take as an argument, but then apply a scaling factor to it equal to that applied by the «x» suffix in font change commands (see section ??) so that, in most fonts, the caps will be the same height as lower case letters, thus giving us a kind of *fake small caps* effect. Compared to genuine small caps (see section ??) these have the following advantages:

1. `\cap` and `\Cap` will work with any font, by contrast with genuine small caps that only work with fonts and styles that expressly include them.

2. True small caps, on the other hand, are a variant of the font which, as such, is incompatible with any other variant such as bold, italic, or slanted. However, `\cap` and `\Cap` are fully compatible with any font variant.

The difference between `\cap` and `\Cap` is that while the former applies the scaling factor to all the letters of the words that make up its argument, `\Cap` does not apply any scaling to the first letter of each word, thus achieving an effect similar to what we get if we use real capitals in a text in small caps. If the text taken as an argument in «caps» consists of several words, the size of the capital letter in the first letter of each word will be maintained.

Thus, in the following example

```
The UN, whose \Cap{president} has his
office at \cap{uN} headquarters...
```

The UN, whose ⋒president has his office at UN headquarters...

we need to note, first of all, the difference in size between the first time we write « UN » (in upper case) and the second time (in small caps, « UN »). In the example, I wrote `\cap{uN}` the second time so we can see that it does not matter if we write the argument that `\cap` takes in upper or lower case: the command converts all letters into upper case and then applies a scaling factor; by contrast with `\Cap` that does not scale the first letter.

These commands can also be *nested*, in which case the scaling factor would be applied once more, resulting in a further reduction, as in the following example where the word « capital » in the first line is scaled yet again:

```
\cap{People who have amassed their
\cap{capital} at the expense of others
are more often than not
{\bf decapitated} in revolutionary
times}.
```

PEOPLE WHO HAVE AMASSED THEIR CAPITAL AT THE EXPENSE OF OTHERS ARE MORE OFTEN THAN NOT **DECAPITATED** IN REVOLUTIONARY TIMES.

The `\nocap` command applied to a text to which `\cap` is applied, cancels out the `\cap` effect in the text that is its argument. For example:

```
\cap{When I was One I had just begun,
when I was Two I was \nocap{nearly}
new (A.A. Milne)}.
```

WHEN I WAS ONE I HAD JUST BEGUN, WHEN I WAS TWO I WAS nearly NEW (A.A. MILNE).

We can configure how `\cap` works with `\setupcapitals` and we can also define different versions of the command, each with its own name and specific configuration. This we can do with `\definecapitals`.

Both commands work in a similar way:

```
\definecapitals[Name][Configuration]
\setupcapitals[Name][Configuration]
```

The « Name » parameter in `\setupcapitals` is optional. If it is not used, the configuration will affect the `\cap` command itself. If it is used, we need to give the name we previously assigned in `\definecapitals` to some actual configuration.

In either of the two commands the configuration allows for three options: « `title` », « `sc` » and « `style` » the first and second allowing « yes » and « no » as values. With « `title` » we indicate whether the capitalisation will also affect titles (which it does by default) and with « `sc` » we indicate whether the command should be genuine small caps (« yes »), or fake small caps (« no »). By default it uses fake small caps which has the advantage that the command works even if you are using a font that has not implemented small caps. The third value « `style` » allows us to indicate a style command to be applied to the text affected by the `\cap` command.

3.2.2 Superscript or subscript text

We already know (see section ??) that in maths mode, the reserved characters « _ » and « ^ » will convert the character or group that immediately follows into a superscript or subscript. To achieve this effect outside of maths mode, ConTEXt includes the following commands:

- `\high{Text}`: writes the text it takes as an argument as a superscript.
- `\low{Text}`: writes the text it takes as an argument as a subscript.
- `\lohi{Subscript}{Superscript}`: writes both arguments, one above the other: on the bottom the first argument, and on top the second, which brings about a curious effect:

```
\lohi{below}{above}
```

above
below

3.2.3 Verbatim text

The Latin expression *verbatim* (from *verbum* = *word* + the suffix *atim*), which could be translated as « literally » or « word for word », is used in text processing programs like ConTEXt to refer to fragments of text that should not be processed at all, but should be dumped, as written, into the final file. ConTEXt uses the command `\type` for this, intended for short texts that do not occupy more than one line and the `typing` environment intended for texts of more than one line. These commands are widely used in computer books to show code fragments, and ConTEXt formats these texts in monospaced letters like a typewriter or a computer terminal would. In both cases the text is sent to the final document without *processing*, which means that they can use reserved characters or special characters that will be transcribed *as is* in the final file. Likewise, if the argument of `\type`, or the content of `\starttyping` includes a command, this will be *written* in the final document, but not executed.

The `\type` command has, besides, the following peculiarity: its argument *can* be contained within curly brackets (as is normal in ConTEXt), but any other character can be used to delimit (surround) the argument.

When ConTEXt reads the `\type` command it assumes that the character which is not a blank space immediately following the name of the command will act as a delimiter of its argument; so it considers that the contents of the argument begin with the next character, and end with the character before the next appearance of the *delimiter*.

Some examples will help us to understand this better:

```
\type 1Tweedledum and Tweedledee1
\type |Tweedledum and Tweedledee|
\type zTweedledum and Tweedledeez
\type (Tweedledum and Tweedledee(
```

Note that in the first example, the first character after the command name is a «1», in the second a «|» and in the third a «z»; so: in each of these cases ConTEXt will consider that the argument of `\type` is everything between that character and the next appearance of the same character. The same is true for the last example, which is also very instructive, because in principle we could assume that if the opening delimiter of the argument is a «(», the closing one should be a «)», but it is not, because «(» and «)» are different characters and `\type`, as I said, searches for a closing character delimiter which is the same as the character used at the start of the argument.

There are only two cases where `\type` allows the opening and closing delimiters to be different characters:

- If the opening delimiter is the «{» character, it thinks the closing delimiter will be «}».
- If the opening delimiter is «<<», it thinks that the closing delimiter will be «>>». This case is also unique in that two consecutive characters are being used as delimiters.

However: the fact that `\type` allows any delimiter does not mean that we should use « weird » delimiters. From the point of view of the *readability* and *comprehensibility* of the file source, it is best to delimit the argument of `\type` with curly brackets where possible, as is normal with ConTEXt; and when this is not possible, because there are curly brackets in the `\type` argument, use a symbol: preferably one that is not a ConTEXt reserved character. For example: `\type *This is a closing curly bracket: '}'*`.

Both `\type` and `\starttyping` can be configured with `\setuptype` and `\setuptyping`. We can also create a customised version of these with `\definetype` and `\definetyping`. Regarding the actual configuration options for these commands, I refer to « `setup-en.pdf` » (in the directory `tex/texmf-context/doc/context/documents/general/qrcs`.

Two very similar commands to `\type` are:

- `\typ`: works similarly to `\type`, but does not disable hyphenation.
- `\tex`: a command intended for writing texts about TEX or ConTEXt: it adds a backspace before the text it takes as an argument. Otherwise, this command differs from `\type` in that is processes some of the reserved characters it finds in the text it takes as an argument. In particular, curly brackets inside `\tex` will be treated in the same way they are usually treated in ConTEXt.

3.3 Character and word spacing

3.3.1 Automatically setting horizontal space

The space between different characters and words (called *horizontal space* in TEX) is normally set automatically by ConTEXt:

- The space between the characters that make up a word is defined by the font itself, which, except in fixed-width fonts, usually uses a greater or lesser amount of white space depending on the characters to be separated, and so, for example, the space between an «A» and a «V» («AV») is usually less than the space between an «A» and an «X» («AX»). However, apart from these

possible variations that depend on the combination of letters concerned and predefined by the font, the space between the characters that make up a word is, in general, a fixed and invariable measure.

- By contrast, the space between words on the same line can be more elastic.
  - In the case of words in a line whose width must be the same as that of the rest of the lines in the paragraph, the variation of the spacing between words is one of the mechanisms that ConTEXt uses to obtain lines of equal width, as explained in more detail in section ??. In these cases, ConTEXt will establish exactly the same horizontal space between all the words in the line (except for the rules below), while ensuring that the space between words in the different lines of the paragraph is as similar as possible.
  - However, in addition to the need to stretch or shrink the spacing between words in order to justify the lines, depending on the active language, ConTEXt takes certain typographical rules into consideration whereby in certain places the typographical tradition associated with that language adds some extra white space, as is the case, for example, in some parts of the English typographical tradition, which adds extra white space after a full stop.

    These extra white spaces work for English and possibly for some other languages (though it is also true that in many instances, publishers in English nowadays choose not to have extra space after a full stop) but not for Spanish where the typographical tradition is different. So we can temporarily enable this function with `\setupspacing[broad]` and disable it with `\setupspacing[packed]`. We could also change the default configuration for Spanish (and for that matter for any other language including English), as explained in section 3.5.2.

3.3.2 Altering the space between characters within a word

Altering the default space for the characters that make up a word is considered very bad practice from a typographical point of view, except in titles and headings. However, ConTEXt provides a command to alter this space between the characters in a word:[1] `\stretched`, whose syntax is as follows:

```
\stretched[Configuration]{Text}
```

[1] It is very typical of the philosophy of ConTEXt to include a command to do something that the ConTEXt documentation itself advises against doing. Although typographical perfection is sought, the aim is also to give the author absolute control over the appearance of his or her document: whether it is better or worse is, in short, his or her responsibility.

where *Configuration* allows any of the following options:

- `factor`: an integer or decimal number representative of the spacing to be obtained. It should not be too high a number. A factor of 0.05 is already visible to the naked eye.
- `width`: indicates the total width that the text submitted to the command must have, in such a way that the command itself will calculate the necessary spacing to distribute the characters in that space.

  > According to my tests, when the width established with the `width` option is less than that required to represent the text with a *factor* equal to 0.25, the *width* option and this factor are ignored. I guess that's because `\stretched` allows us only *to increase* the space between the characters in a word, not reduce it. But I don't understand why the width required to represent the text with a factor of 0.25 is used as a minimum measure for the `width` option, and not the *natural width* of the text (with a factor of 0).

- **`style`**: style command or commands to apply to the text taken as an argument.
- **`color`**: the colour in which the text taken as an argument will be written.

So in the following example we can see graphically how the command would work when applied to the same sentence, but with different widths:

```
\stretched[width=4cm]{\bf test text}
\stretched[width=6cm]{\bf test text}
\stretched[width=8cm]{\bf test text}
\stretched[width=9cm]{\bf test text}
```

t e s t t e x t
t e s t t e x t
t e s t t e x t
t e s t t e x t

> In this example it can be seen that the distribution of the horizontal space between the different characters is not uniform. The «x» and «t» in « text » and the «e» and «b» in « test », always appear much closer together than the other characters. I haven't been able to find out why this happens.

Applied without arguments, the command will use the full width of the line. On the other hand, within the text that is the argument to this command, the command `\\` is redefined and instead of a line break, it inserts horizontal space. For example:

```
\stretched{test\\text}
```

t e s t t e x t

We can customise the default configuration of the command with `\setupstretched`.

There is no \definestretched command that would allow us to set customised configurations associated with a command name, however, in the official list of commands (see section ??) it says that \setupstretched comes from \setupcharacterkerning and there is a \definecharacterkerning command. In my tests, however, I have not managed to set any customised configuration for \stretched by means of the latter, although I must admit that I have not spent much time trying to do so either.

3.3.3 Commands for adding horizontal space between words

We already know that to increase the space between words it is of no use to add two or more consecutive blank spaces, since ConTEXt absorbs all consecutive blank spaces, as explained in section ??. If we wish to increase the space between words, we need to go to one of the commands that allows us to do this:

- \, inserts a very small blank space (called a thin space) in the document. It is used, for example, to separate thousands in a set of numbers (e.g. 1,000,000), or to separate a single inverted comma from double inverted commas. For example: « 1\,473\,451 » will produce « 1 473 451 ».

- \space or « \␣ » (a backslash followed by a blank space which, since it is an invisible character, I have represented as « ␣ ») introduces an additional blank space.

- \enskip, \quad and \qquad insert a blank space in the document of half an *em*, 1 *em* or 2 *ems* respectively. Remember that the *em* is a measure dependent on the size of the font and is equivalent to the width of an «m», which normally coincides with the size in points of the font. So, using a 12 point font, \enskip gives us a space of 6 points, \quad gives us 12 points and \qquad gives us 24 points.

Along with these commands which give us blank space in precise measurements, the \hskip and \hfill commands introduce horizontal space of varying dimensions:

\hskip allows us to indicate exactly how much blank space we want to add. Thus:

```
This is \hskip 1cm 1 centimetre\\
This is \hskip 2cm 2 centimetres\\
This is \hskip 2.5cm 2.5 centimetres\\
```

This is 1 centimetre
This is 2 centimetres
This is 2.5 centimetres

The space indicated may be negative, which will cause one text to be superimposed over another. Thus:

```
This is farce rather than
\hskip -1cm comedy
```

This is farce rathercomedy

\hfill, for its part, introduces as much white space as necessary to occupy the entire line, allowing us to create interesting effects such as right-aligned text, centred text or text on both sides of the line as shown in the following example:

```
\hfill On the right\\
On both\hfill sides
```

On both

On the right

sides

3.4 Compound words

By « compound words » in this section I mean words that are formally understood to be one word, rather than words that are simply conjoined. It is not always an easy distinction to understand: « rainbow » is clearly made up of two words (« rain + bow ») but no English speaker would think of the combined terms in any other way than as a single word. On the other hand, we have words that are sometimes combined with the help of a hyphen or backslash. The two words have distinct meanings and uses but are conjoined (and may in some cases become a single word, but not yet!). So, for example, we can find words like « French–Canadian » or « (inter)communication » (though we may well also find « intercommunication » and discover that the speaking public has finally accepted the two words to be a single word. That is how language evolves).

Compound words present ConTEXt with some problems mainly connected with their potential hyphenation at the end of a line. If the joining element is a hyphen, then from a typographical perspective there is no hyphenation problem at the end of a line at that point, but we would need to avoid a second hyphenation in the second part of the word since that would leave us with two consecutive hyphens which could cause comprehension difficulties.

The « || »command is available to tell ConTEXt that two words make up a compound word. This command, exceptionally, does not begin with a backslash, and allows two different usages:

- We can use two consecutive vertical bars (pipes) and write, for example, « `Spanish||Argentine` ».
- The two vertical bars can have the joining /separating item between two words enclosed between them, as in, for example, « `joining|/|separating` ».

In both cases, ConTEXt will know that it is dealing with a compound word, and will apply the appropriate hyphenation rules for this type of word. The difference between using the two consecutive vertical bars (pipes), or framing the word separator with them, is that in the first case, ConTEXt will use the separator that is predefined as `\setuphyphenmark`, or in other words the hyphen, which is the default (« `--` »). So if we write « `picture||frame` », ConTEXt will generate « Picture–frame ».

With `\setuphyphenmark` we can change the default separator (in the case where we need two pipes). The values allowed for this command are « `--, ---, -, , (, ), =, /` ». Bear in mind, however, that the « `=` » value becomes an em dash (the same as « `---` »).

The normal use of « `||` » is with hyphens, since this is what is normally used between composite words. But occasionally the separator could be a parenthesis, if, for example, we want « (inter)space », or it could be a forward slash, as in « input/output ». In these cases, if we want the normal hyphenation rules for composite words to apply, we could write « `(inter|)|space` » or « `input|/|output` ». As I said earlier, « `|=|` » is considered to be an abbreviation of « `|---|` » and inserts an em dash as a separator (—).

3.5 The language of the text

Characters form words which normally belong to some language. It is important for ConTEXt to know the language we are writing in, because a number of important things depend on this. Mainly:

- Word hyphenation.
- The output format of certain words.
- Certain typesetting matters associated with the typesetting tradition of the language in question.

3.5.1 Setting and changing the language

ConTEXt assumes that the language will be English. Two procedures can change this:

- By using the `\mainlanguage` command, used in the preamble to change the main language of the document.
- By using the `\language` command, aimed at changing the active language at any point in the document.

Both commands expect an argument consisting of any language identifier (or code). To identify the language, we use either the two-letter international language code set out in ISO 639-1, which is the same as that used, for example, on the web, or the English name of the language in question, or sometimes some abbreviation of the name in English.

In table 3.5 we find a complete list of languages supported by ConTEXt, along with the ISO code for each of the languages in question as well as, where appropriate, the code for certain language variants expressly provided for.[1]

So, for example, to set Spanish (Castilian) as the main language of the document we could use any of the three that follow:

```
\mainlanguage[es]
\mainlanguage[spanish]
\mainlanguage[sp]
```

To enable a particular language *inside* the document, we can use either the `\language[Language code]` command, or a specific command to activate that language. So, for example, `\en` activates English, `\fr` activates French, `\es` Spanish, or `\ca` Catalan. Once an actual language has been activated, it remains so until we expressly switch to another language, or the group in which the language was activated is then closed. Languages work, therefore, just like font change commands. Note, however, that the language set by the `\language` command or by one of its abbreviations (`\en`, `\fr`, `\de`, etc.) does not affect the language in which labels are printed (see section 3.5.3).

> Although it may be laborious to mark the language of all the words and expressions we use in our document that do not belong to the main language of the document, it is important to do so if we want to obtain a properly typeset final document, especially

[1] Table 3.5 has a summary of the list obtained with the following commands:
`\usemodule[languages-system]`
`\loadinstalledlanguages`
`\showinstalledlanguages`
Should you be reading this document long after it was written (2020) it is possible that ConTEXt will have incorporated additional languages, so it would be a good idea to use these commands to show an updated list of languages

Language	**ISO code**	**Language** (variants)
Afrikaans	af, afrikaans	
Arabic	ar, arabic	ar-ae, ar-bh, ar-dz, ar-eg, ar-in, ar-ir, ar-jo, ar-kw, ar-lb, ar-ly, ar-ma, ar-om, ar-qa, ar-sa, ar-sd, ar-sy, ar-tn, ar-ye
Catalan	ca, catalan	
Czech	cs, cz, czech	
Croatian	hr, croatian	
Danish	da, danish	
Dutch	nl, nld, dutch	
English	en, eng, english	en-gb, uk, ukenglish, en-us, usenglish
Estonian	et, estonian	
Finnish	fi, finnish	
French	fr, fra, french	
German	de, deu, german	de-at, de-ch, de-de
Greek	gr, greek	
Greek (ancient)	agr, ancientgreek	
Hebrew	he, hebrew	
Hungarian	hu, hungarian	
Italian	it, italian	
Japanese	ja, japanese	
Korean	kr, korean	
Latin	la, latin	
Lithuanian	lt, lithuanian	
Malayalam	ml, malayalam	
Norwegian	nb, bokmal, no, norwegian	nn, nynorsk
Persian	pe, fa, persian	
Polish	pl, polish	
Portuguese	pt, portughese	pt-br
Romanian	ro, romanian	
Russian	ru, russian	
Slovak	sk, slovak	
Slovenian	sl, slovene, slovenian	
Spanish	es, sp, spanish	es-es, es-la
Swedish	sv, swedish	
Thai	th, thai	
Turkish	tr, turkish	tk, turkmen
Ukranian	ua, ukrainian	
Vietnamese	vi, vietnamese	

Tableau 3.5 Language support in ConTEXt

in professional work. We should not mark all the text, but only the part that does not belong to the main language. Sometimes it is possible to automate the marking of the language by using a macro. For example, for this document in which ConTEXt commands are continuously being quoted, the original language of which is English, I have designed a macro which, in addition to writing the command in the appropriate format and colour, marks it as an English word. In my professional work, where I need to quote a lot of French and Italian bibliography, I have incorporated a field in my bibliographic database to pick up the language of the work, so that I can automate the language indication in the quotations and lists of bibliographical references.

If we are using two languages that use different alphabets in the same document (for example, English and Greek, or English and Russian), there is a trick that will prevent us from having to mark the language of expressions built with the alternative alphabet: modify the main language setting (see next section) so that it also loads the default hyphenation patterns for the language that uses a different alphabet. For example, if we want to use English and ancient Greek, the following command would save us from having to mark language of the texts in Greek:

```
\setuplanguage[en][patterns={en, agr}]
```

This only works because English and Greek use a different alphabet, so there can be no conflict in the hyphenation patterns of the two languages, therefore we can load them both simultaneously. But in two languages that use the same alphabet, loading the hyphenation patterns simultaneously will necessarily lead to inappropriate hyphenation.

3.5.2 Configuring the language

ConTEXt associates the functioning of certain utilities with the specific language active at any given time. The default associations can be changed with `\setuplanguage` whose syntax is:

```
\setuplanguage[Language][Configuration]
```

where *Language* is the language code for the language we want to configure, and *Configuration* contains the specific configuration that we want to set (or change) for that language. Specifically, up to 32 different configuration options are allowed, but I will only deal with those that seem suitable for an introductory text such as this:

- **`date`**: allows us to configure the default date format. See further ahead on page 82.

- **`lefthyphenmin, righthyphenmin`**: the minimum number of characters that must be to the left or to the right for hyphenation of a word to be supported. For example `\setuplanguage[en][lefthyphenmin=4]` will not hyphenate any word if there are fewer than 4 characters to the left of the eventual hyphen.

- **`spacing`**: the possible values for this option are « `broad` » or « `packed` ». In the first case (broad), the rules for spacing words in English will be applied, which means that after a full stop and when another character follows, a certain amount of extra blank space will be added. On the other hand, « `spacing=packed` » will prevent these rules from applying. For English, broad is the default.

- **leftquote, rightquote**: indicate the characters (or commands), respectively, that \quote will use to the left and right of the text that is its argument (for this command, see page 84).

- **leftquotation, rightquotation**: indicate the characters (or commands), respectively that \quotation will use to the left and right of the text that is its argument (for this command, see page 84).

3.5.3 Labels associated with particular languages

Many of ConTEXt's commands automatically generate certain texts (or *labels*), as, for example, the \placetable command that writes the label « Table xx » under the table that is inserted, or \placefigure which inserts the label « Figure xx ».

These *labels* are sensitive to the language set with \mainlanguage (but not if set with \language) and we can change them with

```
\setuplabeltext[Language][Key=Label]
```

where *Key* is the term by which ConTEXt knows the label and *Label* is the text we want ConTEXt to generate. So, for example,

```
\setuplabeltext[es][figure=Imagen~]
```

would see that when the main language is Spanish, images inserted with \placefigure are not called « Figure x » but « Imagen x ». Note that after the text on the label itself, a blank space must be left to ensure that the label is not attached to the next character. In the example I have used the reserved character « ~ »; I could also have written « [figure=Imagen{ }] » enclosing the blank space between curly brackets to ensure that ConTEXt will not get rid of it.

What labels can we redefine with \setuplabeltext? The ConTEXt documentation is not as complete as one might hope on this point. The 2013 reference manual (which is the one that explains most about this command) mentions « chapter », « table », « figure », « appendix »... and adds « other comparable text elements ». We can assume that the names will be the English names of the element in question.

One of the advantages of *free libre software* is that the source files are available to the user; so we can look into them. I have done so, and *snooping* through the source files of ConTEXt, I have discovered the file « lang-txt.lua », available in tex/texmf-context/tex/context/base/mkiv which I think is the one that contains the predefined labels and their different translations; so that if at any time ConTEXt generates a

redefined text that we want to change, to see the name of the label that text is associated we can open the file in question and find that we want to change. This way we can see which label name is associated with it.

If we want to insert the text associated with a certain label somewhere in the document, we can do so with the `\labeltext` command. So, for example, if I want to refer to a table, to ensure that I name it in the same way that ConTEXt calls it in the `\placetable` command, I can write: « `Just as shown in the \labeltext{table} on the next page.` » This text, in a document where `\mainlanguage` is English, will produce: « Just as shown in the Tableau on the next page. »

Some of the labels redefinable with `\setuplabeltext`, are empty by default; like, for example, « `chapter` » or « `section` ». This is because by default ConTEXt does not add labels to sectioning commands. If we want to change this default operation, we need only to redefine these labels in the preamble of our document and so, for example, `\setuplabeltext[chapter=Chapter~]` will see that chapters are preceded by the word « Chapter ».

Finally, it is important to point out that although in general, in ConTEXt, the commands that allow several comma-separated options as an argument, the last option can end with a comma and nothing bad happens. In `\setuplabeltext` that would generate an error when compiling.

3.5.4 Some language-related commands

A. Date-related commands

ConTEXt has three date-related commands that produce their output in the active language at the time they are run. These are:

- `\currentdate`: run without arguments in a document in which the main language is English, it returns the system date in the format « Day Month Year ». For example: « 11 September 2020 ». But we can also tell it to use a different format (as would happen in the US and some other parts of the English-speaking world that follow their system of putting the month before the day, hence the infamous date, 9/11), or include the name of the day of the week (`weekday`), or include only some elements of the date (`day, month, year`)

  To indicate a different date format, « `dd` » or « `day` » represent the days, « `mm` » the months (in number format), « `month` » the months in alphabetical format in lower case, and « `MONTH` » in upper case. Regarding the year, « `yy` » will write only the last digits, while « `year` » or « `y` » will write all four. If we want some

separating element between the date components, we must write it expressly. For example

```
\currentdate[weekday, dd, month]
```

when run on 9 September 2020 will write « Wednesday 9 September ».

- `\date`: this command, run without any argument, produces exactly the same output as `\currentdate`, meaning, the actual date in standard format. However, a specific date can be given as an argument. Two arguments are given for this: with the first argument we can indicate the day (« `d` »), month (« `m` ») and year (« `y` ») corresponding to the date we want to represent, while with the second argument (optional) we can indicate the format of the date to be represented. For example, if we want to know what day of the week John Lennon and Paul McCartney met, an event which, according to Wikipedia, took place on 6 July 1957, we could write

  ```
  \date[d=6, m=7, y=1957][weekday]
  ```

  and so we would find out that such an historical event happened on a Saturday.

- `\month` takes a number as an argument, and returns the name of the month corresponding to that number.

B. The `\translate` command

The translate command supports a series of phrases associated with a specific language, so that one or another will be inserted in the final document depending on the language active at any given time. In the following example, the translate command is used to associate four phrases with Spanish and English, which are saved in a memory buffer (regarding the `buffer` environment, see section ??):

```
\startbuffer
  \starttabulate[|*{4}{lw(.25\textwidth)|}]
    \NC \translate[es=Su carta de fecha, en=Your letter dated]
    \NC \translate[es=Su referencia, en=Your reference]
    \NC \translate[es=Nuestra referencia, en=Our reference]
    \NC \translate[es=Fecha, en=Date] \NC\NR
  \stoptabulate
\stopbuffer
```

so that if we insert the *buffer* at a point in the document where Spanish is activated, the Spanish phrases will be played, but if the point in the document where the buffer is inserted has English activated, the English phrases will be inserted. Thus:

```
\language[es]
\getbuffer
```

will generate

Su carta de fecha Su referencia Nuestra referencia Fecha

while

```
\language[en]
\getbuffer
```

will generate

Your letter dated Your reference Our reference Date

C. The \quote and \quotation commands

One of the most common typographical errors in text documents occurs when quote marks (single or double) are opened but not expressly closed. To avoid this happening, ConTEXt provides the \quote and \quotation commands that will quote the text that is their argument; \quote will use single quotation marks and \quotation will use double quotation marks.

These commands are language sensitive in that they use the default character or command set for the language in question to open and close quotes (see section 3.5.2); and so, for example, if we want to use Spanish as the default style for double quotation marks – the guillemets or chevrons (angle brackets)) typical of Spanish, Italian, French, we would write:

`\setuplanguage[es][leftquotation=«, rightquotation=»]`.

These commands do not, however, manage nested quotes; although we can create the utility that does this, taking advantage of the fact that \quote and \quotation are actual applications of what ConTEXt calls *delimitedtext*, and that it is possible to define further applications with \definedelimitedtext. Thus the following example:

```
\definedelimitedtext
  [CommasLevelA]
  [left=«, right=»]

\definedelimitedtext
  [CommasLevelB]
  [left=", right="]
```

```
\definedelimitedtext
  [CommasLevelC]
  [left=`, right=']
```

will create three commands that will allow up to three different levels of quoting. The first level with side quotes, the second with double quotes and the third with single quotes.

Of course, if we are using English as our main language, then the default single and double quotation marks (curly, not straight, as you find in this document!) will be automatically used.

Appendices

Annexe A

Installing, configuring and updating ConTEXt

TEX's main distributions (TeX Live, teTeX, MikTeX, MacTeX, etc.) include a version of ConTEXt. However, this is not the most updated version. In this appendix I will explain two procedures to install two different versions of ConTEXt; the first includes both ConTEXt Mark II and Mark IV and the second includes only ConTEXt Mark IV.

The installation procedure follows the same steps on any operating system; but the details change from one system to another. However, we can simplify things in such a way that in the following lines I will distinguish between two big groups of systems:

- **Unix-type systems:** This includes Unix itself, as well as GNU Linux, Mac OS, FreeBSD, OpenBSD or Solaris. The procedure is basically the same in all these systems; there are some very small differences that I will highlight in the appropriate place.
- **Windows systems**, that includes the different versions of that operating system: Windows 10 (the latest version, I think), Windows 8, Windows 7, Windows Vista, Windows XP, Windows NT, etc.

  Important note on the installation process on Microsoft Windows systems:

  ConTEXt, like all TEX systems, is designed to work from a terminal; the programs and procedures for installation, too. In Windows this is also perfectly possible and should not create any major difficulty. The problem is that, on the one hand Windows users are not always used to doing this, and on the other, since Windows came into being in the *illusion* (false) that everything in a computer system could be done graphically, in general the versions of that operating system do not *advertise* too much about how to use the terminal. And then, it is common for each version of this system to change the name of the program that runs the terminal and how to open it. As far as I know, the Windows terminal emulation program has been given many names: « DOS window », « Command

Prompt », « cmd », etc. The location of this program in the Windows application menu also changes depending on the version of Windows in question.

I stopped using Windows-based systems in 2004, so there is little I can do here to help the reader. He or she will have to figure out, on their own, how to open a terminal in their particular version of the operating system; which shouldn't be too difficult.

1 Installing and configuring « ConTEXt Standalone »

The ConTEXt distribution known as « Standalone », also known as « ConTEXt Suite », is a complete and updated distribution of ConTEXt, which downloads the necessary files from the Internet, does not take up too much disk space, is easy to update, and above all — hence the name *Standalone* — is contained in a single directory which can be located anywhere we want on the hard disk. It would even be possible for a single computer to have several versions of ConTEXt each in its own directory. This distribution includes the fonts, binary files and documentation needed to run ConTEXt Mark II (which implies the TEX PdfLatex and XeTeX engines), and ConTEXt Mark IV (which implies the LuaTeX engine).

For information about TEX *engines*, see section 1.4.1; and on TEX engines in relation to ConTEXt, as well as the versions known as Mark II and Mark IV, section 1.5.1.

The following explains how to install, run, update and restore « ConTEXt Standalone » on our system. The data and procedures provided here are a summary of the much more extensive information included in the ConTEXt wiki, to which I have added some additional detail drawn from a wikibook on ConTEXt hosted on wikibooks. If there is any problem with the installation, or if you want to extend any detail, you should directly consult any of these (though the latter is in French)

1.1 Installation

Installing « ConTEXt Standalone » means having an Internet connection, and implies the following steps:

1. Creating the directory in which ConTEXt will be installed.
2. Downloading the installation *script* into this directory.
3. Running this *script* with the desired options.
4. Making some final adjustments.

Step 1: creating the installation directory

This, in fact, has nothing to do with ConTEXt and we have to assume that every user will know how to do it. In Windows systems the normal way is to do it from the file manager. On Unix-type systems, it can be done from a file manager or from a terminal. It is important, however, to keep in mind that it is not recommended that the installation directory contains any blank space in your path. I personally also tend to shy away from using non-English directory names with things like accented vowels in them.

From now on I will assume that the installation directory is, in Unix-like systems, « `~/context/` » and in Windows, « `C:\Programs\context` ».

Step 2: Download the installation *script* into the installation directory

The installation *script* will differ according to the operating system you are installing on:

- On Unix-like systems it can be downloaded, with a web browser, or, from a terminal with « `wget` » or « `rsync` »:

  ```
  wget http://minimals.contextgarden.net/setup/first-setup.sh
  rsync rsync://minimals.contextgarden.net/setup/first-setup.sh
  ```

- On Windows-type systems, as far as I know, there are no standard tools for downloading from the console. It has to be done with a web browser. The download address can be any of the following:

  ```
  http://minimals.contextgarden.net/setup/context-setup-mswin.zip
  http://minimals.contextgarden.net/setup/context-setup-win64.zip
  ```

  Once downloaded, in Windows you have to unzip the file,

Step 3: Run the installation *script*

The installation *script* must be run from the terminal. In Unix-type systems the name of the *script* is « `first-setup.sh` » and can be run with `bash` or `sh`. In Windows-type systems the *script* is called « `first-setup.bat` » and is run by simply typing its name in the system console or MS-DOS window from the installation directory.

The installation *script* allows for the following options:

- `--context`: this option determines which version of ConTeXt will be installed, whether the most recent development version (« –context=latest ») or the latest stable version (« –context=beta »). The default value is « beta ».

- `--engine`: allows us to indicate whether we want to install Mark IV (« –engine=luatex », the default value) or Mark II.

- `--modules`: also install the ConTeXt extension modules that do not belong to the distribution as such, but that offer interesting additional utilities. To do this we need to indicate « –modules=all ».

  With regard to the installation options, I believe that the information in the wiki is now obsolete. There it says that to install only Mark IV you need to explicitly indicate the "–engine=luatex" option and that the "–context=latest" option installs the latest stable version, not the development version. However, from halfway through 2020 the content of first-setup.sh changed, and taking a look inside it I found that to install the very latest version you need to expressly indicate "–context=latest", and that "–engine=luatex" is enabled by default.

The French Wikibook I mentioned at the beginning adds two other possible options to the options I just mentioned (documented on the ConTEXt wiki): « –fonts=all » and « goodies=all ». ConTeXtgarden doesn't mention them, but including them in the installation command as well doesn't hurt. Therefore I would advise you to run the installation script with the following options (depending on whether we are on a Unix- or Windows-type system):

- Unix: `bash first-setup.sh --context=latest --modules=all --fonts=all --goodies=all`
- Windows: `first-setup.bat --context=latest --modules=all --fonts=all --goodies=all`

This, depending on the speed of our Internet connection, may take some time, but not too much.

Configuring a proxy

The installation script uses rsync to obtain the necessary files. So, if you are behind a proxy server, you need to specify its details to rsync. The easiest way to set this is to set the variable RSYNC_PROXY in the terminal or in your startup *script* (.bashrc or the corresponding file for each shell). Replace the username, password, proxyhost and proxyport with the correct information. This is done, on Unix-type systems, with the « `export` » command, and in Windows-type systems with the « `set` » command. For example:

```
export RSYNC_PROXY=username:password@proxyhost:proxyport
```

Sometimes, when we are behind a firewall, port 873 may be closed for outgoing TCP connections. If port 22 is open for ssh connections, one trick that can be used is to connect to a computer somewhere outside the firewall and tunnel into port 873 (using the nc program).

```
export RSYNC_CONNECT_PROG='ssh tunelhost nc %H 873'
```

where the «tunnelhost» is the machine outside the firewall we have access to. Of course, this machine must have nc and port 873 open for the outgoing TCP connection

After running « `first-setup` » in the installation directory two new directories will appear called, « `bin` » and « `tex` » respectively.

Step 4: Final adjustments (Only on GNU Linux)

In GNU Linux systems there are many directories where fonts can be installed. If we want ConTEXt to use these fonts we must tell it where to find them. To do this we must add the following line to the « `tex/setuptex` » file created after the installation:

```
export OSFONTDIR="~/.fonts:/usr/share/fonts:/usr/share/texmf/fonts/opentype/"
```

with which the environment variable OSFONTDIR is loaded with the three directories in which the fonts installed in the system are normally located

The /usr/share/texmf/fonts/ will only be there if there is some other installation of TEX or other systems based on it in our operating system; in this case it should be included in the OSFONTDIR path so we can use the opentype fonts that such an installation may have included. If you have any commercial fonts that you want ConTEXt to use, you have to make sure that the path to these is one of those included in OSFONTDIR, or otherwise, add the path to this variable. I have seen, for example, that some fonts are installed in /usr/local/fonts instead of /usr/share/fonts.

Finally, it may be a good idea to have ConTEXt generate a database with the necessary files for execution. This will be done by running the following three commands from a terminal:

```
. ~/context/tex/setuptex
context --generate
context --make
```

The first instruction is a point (dot). That's an abbreviation for bash's internal `source` command. We can also, of course, run *source* if it's more convenient for us.

1.2 Running « ConTEXt Standalone »

« ConTEXt Standalone » has been designed to be able to coexist with other installations of TEX systems, which is an advantage because it allows us to have several different versions installed on the same operating system; but in order to exploit this advantage it is essential that the environment variables needed to run ConTEXt are not set permanently, because every time we start a terminal to run « `context` » from it, we'll have to start by loading these environment variables into memory. They are contained in the « `tex/setuptex` » (Unix) or « `tex/setuptex.bat` » (Windows) file. This is done:

- In Unix-type systems, after opening the terminal in which we want to use « `context` », by running either of the following two commands:

  ```
  source ~/context/tex/setuptex
  . ~/context/tex/setuptex
  ```

  (assuming that the directory where the version of « `context` » we want to use is « `~/context` »).

- In Windows-type systems, by running the `tex\setuptex.bat` command from the installation directory in the terminal from which we will use ConTEXt.

If there is no other installation of TEX or any of its derivatives in our system, we can avoid this by automating the execution of this order every time a terminal is opened:

- On Unix-like systems this is done by including it in the file containing the general terminal startup *script* (usually « `.bashrc` »).

  The configuration file of a terminal depends on the *shell* program that the terminal uses by default. If this is `bash` (which is the most used in GNU Linux systems), the file read at the beginning is `.bashrc`. The `sh` and `ksh` *shells* use a file called `.profile`, `zsh` uses `.zshenv`, and `tcsh` or `csh` read the `.cshrc` file. Some specific implementation may change the names of these files and so, for example, `.bashrc` is sometimes called `.bash_profile`.

- In Windows-type systems we can create a shortcut on the desktop that runs cmd.exe and then edit it, putting as a command to run when we double click on it:

  ```
  C:\WINDOWS\System32\cmd.exe /k C:\Programs\context\tex\setuptex.bat
  ```

Another possibility, if we do not wish to run this script each time we want to use ConTeXt, nor want to permanently set the environment variables necessary for it to be run, is to do it from the text editor itself, instead of running ConTeXt from a terminal. How you do this depends on the particular text editor you are using. The ConTeXt wiki provides information on how to set up various common editors: LEd, Notepad++, Scite, TeXnicCenter, TeXworks, vim and some others.

1.3 Updating the version of « ConTEXt Standalone » or returning to an earlier version

Mark IV is still under development, so « ConTEXt Standalone » is often updated. To update our installation just repeat the process: we download a new version of « `first-setup.sh` » and run it.

If, for whatever reason, we want to go back to a previous version of « ConTEXt Standalone », just run « `first-setup` » with the « `--context=date` » option, where *date* is the date corresponding to the version we want to recover. Note that the date has to be introduced in the US months-days-years format.

The complete list of ConTEXt versions and associated dates can be found at this link.

Finally, keep in mind that after reinstalling the system, whether it is to upgrade or to return to a previous version, on GNU Linux systems you will have to run step 4 of the installation again, which I have called « Final Adjustments ».

2 Installing LMTX

If we only plan to use ConTEXt Mark IV, and we want to compile our projects not directly with LuaTeX but with LuaMetaTex, a simplified LuaTeX that uses less system resources and that can work on *less powerful* systems, instead of « ConTEXt Standalone », we need to install LMTX which is the latest version of ConTEXt. The name is an acronym of the name of the TEX engine being used: LuaMetaTeX. This version was launched in 2019, and since approximately May 2020 it is the recommended default ConTEXt distribution as suggested in ConTEXt wiki.

> The current development of LMTX is intense, and the beta version can change several times a week. Some of its developments, moreover, temporarily pose certain incompatibilities with Mark IV, and so, for example, while I am writing these lines, the latest

version of LMTX (August 4, 2020) produces an error with the \Caps command. Therefore I would advise newcomers, for the moment, to work with « ConTEXt Standalone » instead.

2.1 The installation itself

The installation is as simple as:

- **Step 1:** Decide on the directory you want to install LMTX in, and, if necessary, create it. I will assume that the installation is done in a directory called « context » located in our user directory.

- **Step 2:** Download (to the installation directory) the zip file from the ConTEXt wiki that corresponds to your operating system and processor. It can be any of the following:

  - GNU/Linux
    - X86 Processor
      - ▷ 32 bit version.
      - ▷ 64 bit version.
    - ARM Processor
      - ▷ 32 bit version.
      - ▷ 64 bit version.
  - Microsoft Windows
    - 32 bit version
    - 64 bit version
  - Mac OS, versión de 64 bits
  - FreeBSD
    - 32 bit version.
    - 64 bit version.
  - OpenBSD6.6
    - 32 bit version.
    - 64 bit version.
  - OpenBSD6.7
    - 32 bit version.
    - 64 bit version.

  If you don't know whether your system is 32-bit or 64-bit, chances are – unless your computer is very old – it's 64-bit. If you don't know whether your processor is X86 or ARM, it's most likely X86.

- **Step 3:** Unzip, the file downloaded in the previous step into the installation directory. A folder will be created called « `bin` » and two files, one called « `installation.pdf` », that contains more detailed information about the installation, and a second file which is the actual installation program called « `install.sh` » (in Unix-type systems) or « `install.bat` » (in Windows systems).

- **Step 4:** Run the installation program (« `install.sh` » or « `install.bat` »). It needs an Internet connection as the installation program searches the web for the files it needs.

  – On Unix-type systems the installation program, located in the installation directory, is run from a terminal, either with `bash`, or with `sh`. It is not necessary to have administrator privileges, unless the installation directory is outside the user's « `home` » directory.

  – In Windows-type systems, you must open a terminal, move to the installation directory, and from the terminal, run `install.bat`. It is not necessary here either that the installation program is run as a system administrator, but it is recommended that this be done so that symbolic links of the files can be used, thus saving disk space.

- **Step 5** inform the system of the path to LMTX:

  In Windows systems, the installation program generates a file, called « `setpath.bat` » which updates all the configuration files necessary to let Windows know that you have installed LMTX in the system and where you has done so. In GNU Linux systems, FreeBSD or Mac OS no *script* that automates the task is generated, so we must incorporate the address for the ConTEXt binaries in the system's PATH variable, which we would get by running in the terminal, from the installation:

  ```
  export PATH="InstallationDir/tex/texmf-Platform/bin:"$PATH
  ```

  where *`InstallationDir`* is the installation directory (for example, "/home/user/context") and `texmf-`*`Platform`* will vary according to the version of LMTX we have installed. For example, an installation on a 64 bit Linux system, `texmf-`*`Platform`* will be "texmf-linux-64". Therefore we should run the following command from the terminal:

  ```
  export PATH="/home/user/context/texmf-linux-64/bin:"$PATH
  ```

This command will include LMTX in the system path, only as long as the terminal from which it has been run remains open. If we want this to be done automatically every time a terminal is opened, we must include this command in the configuration file of the *shell* program used by default in the system. The name of this file changes according to which *shell* program it is: bash, sh, zsh, ksh, tcsh, csh... On most Linux systems, which use bash, the file is called « `.bashrc` » so we should run the following command from our home directory:

```
echo 'export PATH="/home/user/context/texmf-linux-64/bin:"$PATH >>
.bashrc
```

Important note: By executing this step, we will disable the possibility of using other versions of ConTEXt on our system, such as the one incorporated in TeX Live or « ConTEXt Standalone ». If we want to make both versions compatible, it is preferable to use the procedure described in section 3.

2.2 Installing extension modules in LMTX

ConTEXt LMTX does not incorporate a procedure for installing or upgrading the ConTEXt extension modules. However, in ConTEXt wiki there is a *script* that allows you to install and update all the modules along with the rest of the installation.

To do this we need to copy the aforementioned *script*, paste it into a text file located in the main LMTX installation directory (the one containing `install.sh` or `install.bat`) and run it from a terminal. I have personally verified that this works on a GNU Linux system. I'm not sure if it will work on a Windows system, since I don't have any version of that operating system to check it with.

2.3 Updating LMTX

Updating LMTX is as simple as running the installation program again: it will check the installed files against those on the web server and update as necessary.

If the website from which the files are obtained has changed, we obviously need to also change this address in the installation *script*; although perhaps it is easier to download a new version of the installation files in the same directory and extract from it the new « `install.sh` » or « `install.bat` »; or, even easier, unzip the file with the installation program and reinstall without first needing to removing the old files.

2.4 Creating a file that loads the variables into memory needed for LMTX (only GNU/Linux systems)

« ConTEXt Standalone » contains, as we already know, a (« `tex/setuptex` ») file that loads into memory all the variables needed to run it, but LMTX does not include a similar file. We can, however, easily create it ourselves and store it, for example, as « `setuplmtx` » in the « `tex` » directory. The commands that this file could have would be:

```
export PATH="~/context/LMTX/tex/texmf-linux-64/bin:"\$PATH
echo "Adding ~/context/LMTX/tex/texmf-linux-64/bin to PATH"
export TEXROOT="~/context/LMTX/tex"
echo "Setting ~/context/LMTX/tex as TEXROOT"
export OSFONTDIR="~/.fonts:/usr/share/fonts:/usr/share/texmf/fonts/opentype/"
echo "Loading font directories into memory"
alias lmtx="~/context/LMTX/tex/texmf-linux-64/bin/context"
echo "Creating an alias to run lmtx"
```

With this, besides loading into memory the paths and variables needed to run LMTX we would be enabling the « `lmtx` » command as a synonym of « `context` ».

After creating this file, before being able to use LMTX, where we intend to use it we should run the following in the terminal:

```
source ~/context/LMTX/tex/setuplmtx
```

all this assuming that LMTX is installed in « `/context/LMTX` » and that we have called this file « `setuplmtx` » and stored it in « `/context/LMTX/tex` ».

> The above is what I do, to work with LMTX in the same way I used to work with « ConTEXt Standalone ». However, I do not exclude the possibility that in LMTX it is not necessary, for example, to load into memory the variable OSFONTDIR, since I am struck by the fact that ConTEXt wiki says nothing about this.

3 Using several versions of ConTEXt on the same system (only for Unix-type systems)

The operating system utility called `alias` allows us to associate different names with different versions of ConTEXt. So we can use, for example, the version of ConTEXt included in TeX Live and LMTX; or the *Standalone* version and LMTX.

For example, if we store the versions of LMTX downloaded in January and August 2020 in different directories, we could write the following two instructions in « `.bashrc` » (or equivalent file read by default when opening a terminal):

```
alias lmtx-01="/home/user/context/202001/tex/texmf-linux-64/bin/context"
alias lmtx-08="/home/user/context/202008/tex/texmf-linux-64/bin/context"
```

These instructions will associate the names `lmtx-01` with the version of LMTX installed in the « `context/202001` » directory and `lmtx-08` with the version installed in « `context/202008` ».