Internationalisation in Operating Systems for Handheld Devices

Jere Käpyaho

University of Tampere
Department of Computer and
Information Sciences
Master's thesis
December 2001

University of Tampere
Department of Computer and Information Sciences
Jere Käpyaho: Internationalisation in Operating Systems for Handheld Devices
Master's thesis, 77 pages, 3-page appendix

December 2001

Abstract

Because of their personal nature, handheld devices are expected to conform well to the cultural expectations of their users. In operating system software the support for culture-dependent data representations and interoperability between languages and writing systems is achieved by internationalisation, while native-language user interfaces are realised by subsequent localisation.

The most practical way to implement internationalisation is on the operating system level. The aim of this study is to evaluate the level of internationalisation support in five handheld operating systems and platforms. This is performed by identifying the internationalisation requirements of an operating system and examining their implementation in the selected handheld systems. A comparable internationalisation score for each system is produced.

Keywords: internationalisation, handheld devices, operating systems, character sets, localisation, software development methodology, software metrics

Acknowledgements

I would like to take the opportunity to thank my employer, Nokia Mobile Phones, for providing me with the resources to write this thesis. Warm thanks also go to all my colleagues and supervisors at Nokia for peer reviews and encouragement. Special thanks to Marja Vilander for helping me narrow down the subject.

A big thank you for support and encouragement goes to my lovely wife Virpi. This is the first piece of work by me I actually get to dedicate to someone, and it is only natural that I dedicate this to her. During the writing of this thesis she and our children Siiri and Valtteri reminded me that although handheld devices are a part of life, there is also life without handheld devices.

Table of Contents

1. Introduction

Recent decades in software development have seen, among other new and interesting paradigms, a gradually increasing interest in internationalisation. Software must be adapted to many different cultures and languages – in a word, localised. In order to localise software cost-effectively, a disciplined approach to software development is needed. Internationalisation is such an approach, striving to enable the localisation of software and the production of new language variants [Tuthill and Smallberg, 1997].

An even more recent trend in computing, visible for slightly over a decade now, is the rollout of various handheld computing devices, such as personal digital assistants and other mobile information appliances. The miniaturisation of components has lead to a dramatic decrease in the form factor of computing devices. What was only possible with a minicomputer the size of a refrigerator 20 years ago can now be achieved with a pocketsize device. These diminutive devices are rapidly replacing traditional paper calendars and notepads with a rich computing environment that is increasingly augmented by mobile telephony and Internet connectivity [Burden and Slawsby, 2000; Delio, 2000].

In order to be successful in the global marketplace, handheld devices need to accommodate the language and the customs of their users. Satu Ruuska [1999, 218] has observed that language is the starting point for the internationalisation of personal-purpose products. Most people will not become attached to a handheld computing device if it alienates them by communicating in a foreign language[1].

Most users will also be distracted if the devices handle their appointments and other personal data in a way that is very different from their ordinary everyday use, which is usually tied to the user's specific cultural environment and its customary notations. According to Elisa del Galdo [1996], "design preferences evolve from cultural attitudes and expectations toward the task to be performed". It can be concluded that the adaptation of handheld computing devices to meet the target users' needs is important. For the device makers to succeed, this needs to be done cost-effectively and in a culturally acceptable manner.

The basis of this study is the insight that handheld information appliances and internationalisation are connected in a very important manner, as described above. For a handheld device to succeed in a global market, it needs to

---

[1] Today's high technology work environments make an interesting exception, because they are increasingly international and the working language is often English.

be localised for the cultures of specific market areas. To support all desired market areas with the least effort the device needs to be internationalised. Hence, internationalisation is a prerequisite to localisation.

This study examines the concept of internationalisation and the requirements that it presents to software. It will become evident that applications require extensive support from the operating system for the internationalisation effort to succeed. Handheld computing devices are the most rapidly growing group of information appliances. In the future, significantly more applications will be produced for them than for any other device category, in many different languages. Hence, the support for internationalisation in handheld operating systems is very important.

The aim of this study is to evaluate the level of support for internationalisation in operating systems used in currently available handheld devices. The results of this evaluation can be used as guidelines for device manufacturers and software developers in determining which handheld device platforms to focus on in global development.

Handheld devices also present numerous design constraints for system design and application development. Another aim of this study is to present these constraints and determine which of them have an effect on the level of internationalisation support in a handheld operating system, and what sort of effect can be observed.

As indicated above, the field of internationalisation has only rather recently started to emerge as a branch of software development. The study of internationalisation has so far been largely shadowed by the study of localisation, which is the most visible part of adapting software to different languages and cultures. However, both internationalisation and localisation are essentially parts of the same process.

The study of handheld devices, itself a field with a rather brief history, has usually concentrated on features of the user interface and usability, where the factors involved are mostly independent of any language. The effect of language and culture on handheld usability is often neglected, even though it has great impact on the users' perceptions of the device and its use. This study deals with usability only indirectly, with the aim of enabling better usability through internationalisation and subsequent localisation. Jakob Nielsen and others have conducted research into the design of international user interfaces, which have been collected in [Nielsen, 1990] and [del Galdo and Nielsen, 1996].

This chapter has introduced the work and its aims. Chapter 2 presents definitions of internationalisation and localisation and pinpoints their place in software development methodology. The history of software localisation and

the emergence of internationalisation are chronicled briefly. Chapter 3 presents the case for internationalisation and gives justification for including internationalisation in the software engineering process. It also compares the internationalisation of operating systems versus applications and elaborates on why internationalisation is particularly important for handheld computing devices.

Chapter 4 introduces the concept of *locale*, which is pivotal to internationalisation and localisation. The chapter also gives examples of locale models in use and includes information about current international standardisation work done on locales. Chapter 5 defines the concepts of *character* and *glyph*, and presents *coded character sets*, especially the Unicode worldwide character set.

Chapter 6 outlines the requirements that internationalisation presents for an operating system. It tries to capture the relevant details that need to be addressed in order to develop internationalised software. In Chapter 7 the requirements are formulated into features typically found in operating systems, and metrics for the level of internationalisation in a handheld operating system are developed.

Chapter 8 presents the handheld device operating systems and platforms that are included in this study. Chapter 9 then details the constraints that are typical for handheld computing devices and evaluates their impact on internationalisation.

In Chapter 10 each of the included operating systems is evaluated in terms of operating system support for the internationalisation features identified in Chapter 7. Finally, Chapter 11 concludes the results of this study and presents topics for further research.

2. Internationalisation and localisation

The fields of internationalisation and localisation are relatively new branches of software development. Literature about these topics began to appear in the early 1990's. In their seminal paper, Russo and Bloor [1993] point out that the Computer-Human Interaction (CHI) community had not previously included cultural awareness as an aspect of user awareness. Naturally, localised software has been produced earlier, but the rapid adoption of personal computers nearly all over the globe and the establishment of shrink-wrapped software production in the 1980's created a strong need for localised applications.

2.1 The global marketplace

Traditionally, custom applications for finance and manufacturing have been developed for internal use in the countries where the client businesses operate. In most cases there has been no need for the localisation of applications. In contrast, applications for personal computers have mostly been developed in English-speaking countries, mainly in the United States. They were intended for use in the United States and Canada, but have been used in Europe and to lesser extent on other continents. Until recently, the Asian information technology market has consisted mainly of Japan and Thailand. These countries have developed national software for national use, also with no pressing need for localisation.

The growth of the global marketplace for application software has created situations where the traditional division of markets is no longer the norm. To sustain growth, American companies need to expand their revenue stream to other markets besides domestic and other English-speaking countries, and have indeed done so. According to Luong et al [1995], half or more of the largest American software companies get their revenue from outside of the United States, and the numbers are increasing. European software vendors have realised this need earlier than their American counterparts, but face new challenges in the Asian markets, mainly in China. For all companies involved, these changes bring along shifts in the ways software is developed and marketed. They need to enable cost-effective production of language variants for several different markets and target audiences.

2.2 The customisation approach

The earliest methods of adapting software to different languages have traditionally used an approach that can be labelled as customisation. The extent of customisation has varied a lot, from the simple task of translating the user

manual and shipping it with the original, unchanged software to translating the menus and prompts of the application.

Whatever the extent of customisation, it is still a process that is rather narrow in scope. While the translation of user interface texts is by no means a trivial undertaking, mere translation is not usually enough to adapt the software to a new market. Often the translation work is organised by the local sales representatives and (partly because of that) not carried out by professional or even skilled translators. Throughout the personal computer phenomenon there have been industrious and ingenious end users who have "localised" software to their own culture with varying success. The customisation approach is also very prone to errors, because it is usually patchwork and done by modifying the existing binary representation of the software.

One approach to customisation is the creation of language-specific software versions by the original software developers. O'Donnell [1994, 59] has made the observation that the first version of most software packages, the so-called "Release 1.0", is monolingual. It has the error messages and other user interface texts written in the developers' own native tongue. With the advent of the World Wide Web and the proliferation of the Open Source movement, the trend seems to be shifting from using the developers' native tongue to using English[2].

Whatever the (only) language of monolingual software, the initial custom development plan is to create different code paths in the software for different languages. After identifying the desired language, execution of the software forks and proceeds along one of these paths. This approach is rather error-prone and tedious to maintain. It also has a major effect on the one thing that all software gets blamed of: being late. If the developers decide to support three or four languages in the initial version, adding just another language will cause yet another round of testing. There is also the problem of delivering upgrades to the users, especially if support for the new languages is coupled with new functionality. The update problem has been alleviated by electronic software delivery via the Internet, but upgrading software remains an unnecessary burden for the end user.

2.3 Localisation

Localisation is the technical term for the process that adapts a software package to different target cultures according to their requirements. Contrary to popular misconception, this process of adaptation does not only consist of translation

[2] The version numbers of current open source software trend to be well below 1.0, but the software is still usually monolingual.

work, but also risk and return analysis, development of localisation tools and processes, user interface design and usability testing and checking for conformance to local legislation, to name a few.

The difference between the customisation approach and actual localisation is largely in the structured approach of the latter. Whereas the *ad hoc* customisation approach is carried out should the need arise, localisation is a carefully planned activity. Kano [1995, 11] distinguishes seven levels of localisation that have been used at Microsoft Corporation, a company with a good track record in producing localised versions of its personal computer applications and operating systems. The seven levels are as follows:

1. Translate nothing
2. Translate documentation and packaging only
3. Enable code
4. Translate software menus and dialogs
5. Translate online help, tutorials, and sample and README files
6. Add support for locale-specific hardware
7. Customize features for locale

The localisation model of Kano deserves justified critique. Of the seven levels presented, the first two clearly belong to the customisation category, even though the first level is really only included for completeness. The third level, "enable code", is actually internationalisation, which is discussed in section 2.4. Levels 4 to 7 make up the bulk of the localisation effort, with some parts overlapping with internationalisation. This overlap is something that cannot and should not be avoided; it is often difficult to determine where internationalisation ends and localisation starts.

The concept of *locale* mentioned in step 7 of the Kano model is very important to internationalisation and localisation. A locale captures the details about the cultural expectations of users with a common language and region, and encapsulates them in a form that is easily accessible to application programs. A detailed description of locales and examples of common implementations are found in Chapter 4.

Luong et al [1994, 5] present a more realistic model of the levels of localisation. Their model has the following levels:

- English product only
- English product handling European data
- English product handling Far-Eastern data

- English product handling bi-directional data
- Full or partial translation of English user interface and documentation
- Full localization plus local market features

However, in this model only the last level really has anything to do with localisation. The authors' view is that with internationalised software the localisation step is not technically difficult. This view is analogous to the proponents of object-oriented design such as Rumbaugh et al [1991], who view the object design of the software as the most important part, while the actual coding is mechanical, almost trivial. Even though object-oriented design has rapidly become the dominant software design technique for new software projects, coding has not become a trivial last phase before testing. Similarly, localisation is not an automated by-product even if the software in question has been internationalised.

2.4 Internationalisation

Internationalisation is a way of designing software so that it can be localised with the minimum effort. According to Turnbull [1999], internationalisation "is the process of adapting a system's data structures and algorithms so that localising the system to a new culture is a matter of translating a database and does not require patching the source." Internationalisation cannot be retrofit into software; it is a design methodology that permeates the software development process.

The aim of internationalisation is to produce software with a *single code base* that can be adapted to a different language and culture without modifying the original source code or binaries. This requirement stems from practical software engineering reasons: if the source code is modified for each language version, it will rapidly create a maintenance nightmare, with many different versions of the software, each with slightly different functionality and several code paths, each of which needs to be tested.

When an internationalised software package is localised, it adapts to the selected language settings at runtime, loading resources and code as needed. The localisation effort is significantly augmented by the architecture of the software, which also tolerates the addition of languages after the initial release.

The internal architecture of internationalised software is not, however, analogous to software that runs on multiple operating systems. Typically different operating systems use incompatible binary formats even on the same physical hardware. Furthermore, different versions of the same operating system running on the same hardware require different code paths, which dramatically increases the amount of module testing required. One aim of interna-

tionalisation is to decrease the amount of module testing that becomes necessary just because the software package has to support different languages.

While localisation is usually closely associated with arts and humanities, such as translation studies, social and cultural studies, and communications, internationalisation is clearly an engineering discipline. Most of the software features required by internationalisation are implemented programmatically, often with sophisticated algorithms. It can be argued that the implementation of some internationalisation requirements actually exceeds most application software programming in complexity, with the exception of highly specialised fields such as cryptography and signal processing, to name a few.

The actual time to handle internationalisation is very early in the software project's lifecycle. Ideally, every software project would be internationalised from the very beginning. However, current literature in software engineering and software project management pays little or no attention to internationalisation. This can be attributed to the lack of general awareness of the issues involved. The requirements stated for software by the adaptation to different languages and cultures are very difficult to fulfil later on in the product cycle, because internationalisation affects the design and the architecture of the software, especially the storage of text and the handling of various character encodings. Other parts, such as the rendering of text and the user interface, are highly visible and are in close connection with localisation and usability testing.

This chapter has presented the concepts of localisation and internationalisation, which is a prerequisite to efficient localisation. The progression from customisation to localisation, facilitated by internationalisation, has been outlined. In the next chapter the case for internationalisation is presented in more detail.

3. The case for internationalisation

The need to produce several language variants of software has increased in several years. While there is a certain philosophical and egalitarian appeal in the notion of serving each user in his or her native tongue, usually the most important reasons to design and build multilingual software are commercial. New revenue streams can be generated by creating software that supports as many languages as possible. This is a very compelling business case for internationalisation and localisation, and it is discussed in section 3.1.

Once the decision is made to support many languages, there remains the need to figure out how to do it as cost-effectively as possible. The initial customisation approach described in section 2.2 is not cost-effective, because the repeated effort of modifying software for each language is a waste of engineering resources. Localisation is a far better approach because of its systematic nature, but it must be backed up by internationalisation.

Internationalisation has the greatest potential of saving engineering and localisation resources when it is done properly at the beginning of the product cycle. The production of new language variants of software becomes significantly easier with internationalisation. Like the possibility of new revenue streams in marketing, this is a compelling engineering case, which is discussed in section 3.2.

Operating systems and applications both need to be internationalised. The difference is that operating systems need to provide the support for international applications, while applications need to use the support provided in the operating system in order to be internationalised with as little effort as possible. These differences are explored in section 3.3.

Internationalisation and localisation are important especially in handheld devices because of the users' close personal connection with the devices. A desktop personal computer cannot be carried along at all, and a laptop computer does not fit in a pocket. However, a handheld device travels along with the user and is consulted frequently. If this exchange is conducted in a foreign language, it alienates the user and makes the device less acceptable for everyday use. Section 3.4 details the importance of handheld internationalisation.

3.1 New markets and revenue streams

While there has been no visible stagnation in the software market, there are several trends in information technology that will change the way companies operate. One of these is the globalisation of business strategies and operations. The most significant contributor to this trend is the increase in use of the World

Wide Web for commerce. International businesses can operate on the Internet 24 hours a day, 7 days a week without interruption. This attracts customers from around the globe who are not constrained by the opening hours of the business. Automated ordering and dispatching systems can take care of most of the customers' orders. However, customers expect service in their own native tongue, even if the business operates in a different country than their own.

In the traditional software market, as well as in the handheld device market, there are still market areas with a lot of untapped potential for growth. The most notable of these markets is the Asia-Pacific region, especially China. The growth of the Chinese economy has proceeded at a staggering pace for the last decade. Likewise in Japan, which in the late 20th century has traditionally been the frontrunner in the development of electronic equipment, there are significant market opportunities. Throughout the globe there are similar pockets of relatively untapped revenue, such as Russia and other countries of the former Soviet Union, and Northern Africa. According to a survey conducted by Donald Day [2000], the latter will increase in importance at the expense of Western Europe and other traditional markets.

These new markets can only be penetrated with devices and software that conform to the users' expectations about them. These expectations are strongly linked to native language and cultural conventions. Localisation of the devices and software is essential, while internationalisation makes efficient localisation possible.

3.2 Cost-effective software engineering

In today's high tech work environment, skilled software engineers are a scarce resource. Requirements for programmer competence change rapidly, and intense competition turns project management and recruitment of personnel into a constant battle for resources. Therefore, conventional wisdom says dictates that if it saves software engineering resources and makes the jobs of the overworked engineers easier, it must be a good thing.

The traditional customisation approach to producing software in many languages is a talent sink. Software engineers could be busy doing something more productive than laboriously making slight modifications to existing code just to accommodate another foreign language, sending it out for testing, then doing it again for another language, and so on. Internationalisation provides a better, more cost-effective and meaningful way of using precious software engineering resources in the development of multilingual software. Since international software will gradually become the norm rather than the exception, it is vital to incorporate internationalisation into the software design end engineering processes.

It can be argued that the initial investment in internationalisation is bigger than with non-internationalised products in terms of resources and development time. However, there is an analogy in the realm of object-oriented software development and reusable software can be found: according to Rumbaugh et al [1991, 282] "planning for reuse [in object-oriented design] takes more foresight and represents an investment". Similarly, internationalisation represents an investment in the future, but it is more akin to a design pattern than a design methodology.

Internationalisation facilitates the simultaneous release of a software product in many different languages. For example, during the 1990s Microsoft Corporation has been able to gradually decrease the lead-time, or "delta", between the U.S. English version of Microsoft Windows and localised versions. According to Kano [1995, 14] this is a result of internationalisation.

Even if a software product has been internationalised, it is often necessary to add a language to the selection after the product has already shipped. Internationalisation also facilitates this kind of "post-engineering". If the software has been internationalised, the new language capabilities can easily be incorporated into the next major version of the software. Usually new languages are not added in bug-fix versions of the software, presumably because the internationalisation level of the product is not high enough, and the adding of new languages would cause another test round and introduce new bugs.

3.3 Internationalisation of operating systems and applications

Arguably the best place to implement internationalisation features is at the operating system level. To access these features, an application program interface (API) needs to be defined and made available for software developers, as is customary to do with all operating system functionality intended for use by third party applications. This way all the functionality, including internationalisation support, is immediately useful for all the application programs that are written for the operating system.

There is little point in reimplementing any common features in different applications, but precisely this has been done previously with word processors, spreadsheets, and other office productivity applications. Spell checking and mathematical function libraries have been laboriously implemented from scratch because common functionality has not been available. This code is usually reusable only by the original software vendor.

The next step towards reuse has traditionally been the use of code libraries. They have been provided along with language compilers, causing problems when switching compiler vendors. Nevertheless, an industry of compiler-independent third party libraries has emerged, covering such application features

as data entry, spell checking, database connectivity and also internationalisation.

In internationalisation, many features required are sufficiently hard to implement to warrant their inclusion in the operating system. The operating system vendor is best aware of how to implement solutions such as input methods or text collation (further discussed in Chapter 6) so that the functionality and user experience are uniform across applications. If each application vendor implements internationalisation features by themselves, the results are going to be different, and interoperability between applications will suffer.

3.4 Handheld internationalisation

Handheld devices provide a feasible, truly portable alternative for time management, note taking, and entertainment. Increasingly, these devices also provide capabilities for mobile telephony and Internet access. Whereas portable laptop computers are intended for mobile power users as a replacement for the corporate office while on the road, handheld devices are intended for common everyday tasks such as planning and organising leisure time and appointments, keeping track of birthdays, making grocery lists, tracking car mileage, and many other things. For a handheld device to be a seamless part of everyday life it needs to be accessible in the user's own native language—it needs to be localised.

Most visibly, the localisation of a handheld device means that the user interface features of the device—prompts, menus, status messages and the like—are in the language of the user. As Ruuska [1999] has found, this is a significant part of the user experience. However, it is only a start. Users also expect that all the data that is handled by the device are presented in a format that fits the user's expectations. Examples of such data include dates, monetary amounts, and measurements. These data representations can only be achieved algorithmically in software; hence, internationalisation is an essential part of the implementation of a handheld device.

While handheld users are usually happy with a device that works in their own language, many people tend to travel a lot or interact with people of various nationalities. In many such interactions the handheld device plays an important role. Therefore the ability to switch languages (and conventions) can be considered a useful feature of a handheld.

Even more frequent and important for Internet-enabled handhelds is the need to accommodate data that has been received from another device with different language settings. For example, e-mail messages composed in a foreign language usually use a character encoding that is different than the one used in

the receiver's computing environment, but quite common in the computing environments of that language's users.

In this chapter the business and engineering cases for internationalisation were presented. It has been shown that internationalisation is particularly important for handheld devices because of the users' expectations. In the next chapter an important concept in internationalisation, the locale model, is explored.

4. Locales

When the need for application programs that interact with their users in other languages besides English began to grow steadily, it was soon realised that it was not enough to consider the user's native language as the only feature that requires customisation. Some languages are spoken in many different countries, often with slight variations in pronunciation and spelling. Then again, in some countries more than one language is spoken.

For example, in Canada there are two official languages, English and French. In France the latter is the only official language, while English is spoken also in the United States, Great Britain, and Ireland among others. French is also spoken in many African countries, like Spanish is spoken not only in Spain but in many Latin American countries as well.

Each of these combinations of language and country usually have their own ways of expressing dates, times, and numbers. Another major difference between these countries is their currency and the symbols used to represent it. Therefore it is not possible to use the concept of country or language alone as a classification, for example to determine which language version of some software product is to be sold in any given country.

The term "locale" signifies the relationship between a language and a country or territory. Kano [1995, 2] determines locale as a collection of "the features of the user's environment that are dependent on language, country, and cultural conventions". For the reasons mentioned above we need a way to distinguish French as spoken in France, French as spoken in Canada, English spoken in Great Britain, English spoken in the United States and so on. We use the concept of locale to distinguish these language-country pairs.

This chapter outlines the features of a locale and presents several common locale models. A discussion of the current standardisation work rounds up the chapter.

4.1 Modelling cultural conventions

Locales are a very important concept in internationalisation. Ken Lunde [1999, 17] points out that there are essentially two basic internationalisation models, the *locale model* and the *multilingual model*. Of these two, the locale model is by far easier to implement; however, newer operating systems such as Microsoft Windows 2000 use the multilingual model.

Locales provide a model for encapsulating information about the cultural conventions of a group of users. Locales, in several different forms, are used in UNIX System V and POSIX-compatible operating systems (including Linux and

its many variants), MS-DOS, OS/2, Microsoft Windows, and the Java platform, to name a few traditional systems. While the focus of this study is operating systems for handheld devices, the locale models of handheld operating systems are more or less inherited from traditional desktop systems, as we will see later.

Kano [1995, 2] observes that "in Windows, locales usually provide more information about cultural conventions than about languages". The same applies to many other operating systems as well, since cultural conventions are much easier to encapsulate and describe formally than natural languages. A typical example of such a convention is the calendar system used. While the implementation of a Chinese calendar is not a trivial undertaking, it is still several magnitudes easier than natural language parsing. More examples of these conventions are presented in section 4.2.

Conceptually, a locale is a database of various data related to cultural conventions and language. This database is not necessarily implemented using a database management system. Instead it may be a single flat file or a collection of binary or text files partitioned by locale.

4.2 The scope of a locale

In order for a locale model to be useful, it should be broad enough to encapsulate all the necessary information about the user's cultural preferences. This section attempts to extract all those preferences that should ideally be covered in a locale model.

One possible classification of locale-related information is found in the working draft of the ISO/IEC 15435 standard [ISO, 1999a], which defines an API for internationalisation. The future of this standard is somewhat uncertain, as Arnold F. Winkler, the convenor of the relevant ISO/IEC working group (JTC1/SC22/WG20) has proposed that the project should be withdrawn due to lack of interest in the user community [Winkler, 2000]. However, the working draft provides a reasonably complete outline of things that should be included in a locale. It is closely connected to another ISO/IEC standard in progress, namely ISO/IEC 14652, Specification method for cultural conventions [ISO, 1999b], which is described in section 4.7.

Locale information should include at least the following:

- Information about the language and country related to the locale, such as the native names of the language and the country and their English translations.
- Collation sequence information: the native sort order of the language used in sorting text strings.

- Character information: which characters are used in text? How are the characters classifieds as text, numeric digits, and punctuation? How are the characters mapped into upper and lower case?
- Character encoding information: which character encodings are used? What information is required to convert between encodings?
- Numeric formatting conventions: instructions for the typical representation of numbers (both integer and decimal) and currency values.
- Date and time formatting conventions: instructions for the typical representation of various date and time formats.
- Time zone information, including daylight savings/summer time adjustments.
- Calendar conventions: which is the typical calendar used? Does the calendar use day numbering or week numbering, and how?
- Measurements: does the locale use metric or imperial measurements for length, volume, velocity etc.?
- Paper size: what are the customary paper sizes used in the locale?
- Address formats: what are the fields used in postal addresses, and what is their order?
- Telephone number formats: how are telephone numbers constructed and formatted?

Some other locale-specific conventions, such as colours, graphics, fonts, and sounds, are in the realm of user interface development and are not dealt with here. Formal classification is not possible, but internationalised software supports the changing of these items. The information items outlined above are detailed in Chapter 6.

4.3 POSIX and Standard C locales

One of the first locale models in common use is the one defined by the POSIX standard. POSIX, or Portable Operating System Interface, is a family of standards developed by the IEEE (Institute of Electrical and Electronics Engineers). The POSIX standards consist of a series of specifications for UNIX-like operating systems with common functionality and interfaces. Of these standards, POSIX.1 defines the operating system interface, and is also the ISO/IEC standard 9945-1:1990. When applied to an operating system, the term "POSIX compliant" denotes an implementation of the POSIX.1 standard, which is not restricted to UNIX or UNIX-like systems. For example, Microsoft Windows NT has a subsystem that allows POSIX compliant programs to run.

The POSIX locale model is defined in the POSIX.2 standard [ISO, 1993] and the X/Open Portability Guide, Issue 4 (XPG4). This locale model is also the basis for the locales used in the ANSI/ISO standard of the C programming lan-

guage (ANSI X3.169-1989 and ISO/IEC 9899:1990). Currently this locale model is used in POSIX-compliant operating systems and their C compilers and libraries, such as SunSoft Solaris and the many variants of Linux, but also in the standard C libraries of compilers for non-POSIX-compliant systems.

The POSIX locale model consists of six categories, each of which encapsulates a different area of cultural conventions. The categories and their contents are detailed in Table 1, slightly condensed from [Tuthill and Smallberg, 1997].

Name	**Purpose**
LC_TIME	Determines date and time formats.
LC_MONETARY	Specifies monetary formats.
LC_NUMERIC	Determines numeric separator characters.
LC_CTYPE	Controls character and string handling functions.
LC_COLLATE	Controls sorting of strings.
LC_MESSAGES	Controls message catalogs.

Table 1. The POSIX locale categories.

POSIX locales are identified by a convention established by XPG4. The name of a locale takes the following form:

language_territory.codeset@modifier

The *language* part is one of the language codes defined in the ISO 639:1988 standard [ISO 1988a], while *territory* is one of the codes defined in a corresponding standard, ISO 3166-1:1997 [ISO 1997]. The *codeset* part specifies the desired character encoding, while *@modifier* is intended for vendor-specific extensions. Tuthill and Smallberg [1997, 46] state that there is no standard for naming the *codeset* or *@modifier*. Of these components only *language* is mandatory.

The locale categories are accessed through a C language API which consists of functions and data structures. The functions modify their behaviour according to the system's locale, which is set with an environment variable or with an explicit *setlocale()* function call. Each category can also be set separately with this function. Many of the functions in the locale API replace traditional C library functions, especially in the handling of character strings.

The LC_TIME category contains information about the date and time formats customary to a locale. Applications read the system date and time using library calls or construct a *struct tm* variable, then use the *strftime()* function to obtain a locale-specific textual representation of the date and/or time.

The data in the LC_MONETARY category specifies how currency values are formatted. The *strfmon()* function produces a locale-specific formatted string

quite similarly to *strftime()*. Both of these functions use a series of formatting patterns to modify the result. The LC_NUMERIC category specifies the decimal separator (usually a comma or a period), which affects the *atoi()*, *atof()* and *atol()* library functions. The *localeconv()* function extracts the information in the LC_MONETARY and LC_NUMERIC categories into a *struct lconv* variable.

Traditionally programmers have used the ASCII values of characters to determine if a character is upper or lower case, a digit, or a punctuation character. These methods do not work with other character encodings, so data in the LC_CTYPE category of a POSIX locale modifies the behaviour of the library functions *isalpha()*, *isupper()*, *islower()*, *isdigit()* etc. to return a correct answer for other encodings besides ASCII. Traditional string comparisons using *strcmp()* do not give correct results for other languages and character sets than English and ASCII, so the LC_COLLATE category instructs the *strcoll()* library function to perform a string comparison according to the current locale's customary rules.

The POSIX locale model also facilitates for the encapsulation of a program's user-visible messages into a *message catalog*. Each message is given a unique ID which is compiled along with the message's text into a binary file that is read by the *catgets()* function. A message catalog is created for each locale that the application requires.

The POSIX locale model continues to handle the internationalisation requirements of Standard C programs running in a POSIX compliant system. However, as Schmitt [2000, 82] has observed, because the naming of locales is not standardised in C, it varies from one system to another.

The POSIX locale model is also the basis of locales in the Standard C++ Library, which are discussed in the next section.

4.4 Standard C++ locales

The C++ programming language was standardised by ISO in 1998. The standard, ISO 14882:1998 [ISO 1998], specifies not only the behaviour of the language, but the standard libraries as well. The C++ standard library is extensive, with a large collection of data structures and algorithms, and also contains a locale model for internationalisation.

Just as the Standard C library has support for locales using the POSIX locale model, standard C++ has a locale model (specified in Chapter 22 of the standard). However, unlike the POSIX model, and true to the nature of the C++ language, the design of the C++ locale model is object-oriented. According to Kreft and Langer [1997], the internationalisation semantics are broken out into separate class templates called *facets*. The standard defines a total of seven standard facets, which are roughly equivalent to the categories in the POSIX

locale model, and adds support for conversion between different character encodings.

The standard C++ locale model consists of a class called *locale* and facets that are contained in the locale class. The facets and their uses are described in Table 2. The reader is encouraged to compare the contents of this table with Table 1, which lists the POSIX locale categories.

Purpose	**Facets**

Numeric	`num_get<charT, inputIterator>`
	`num_put<charT, outputIterator>`
	`numpunct<charT>`

Monetary	`money_get<charT, bool, inputIterator>`
	`money_put<charT, bool, outputIterator>`
	`moneypunct<charT>`

Time	`time_get<charT, inputIterator>` `time_put<charT, outputIterator>`
Character classifications	`ctype<charT>`
Collation	`collate<chart>`
Code conversions	`codecvt<internT, externT, stateT>`
Messages	`messages<charT>`

Table 2. Standard C++ facets contained in the *locale* class.

The *xxx_get* facets provide parsing services, while the *xxx_put* facets provide formatting services. The *messages* facet provides access to message catalogs.

Even though the standard C++ locale model covers the same areas as the POSIX locale model, it is semantically quite different. Furthermore, it is very closely connected with the standard C++ *iostream* library. Each *iostream* has an attached locale object, which is initially the global default locale. When another locale is desired, it is *imbued* to the iostream so that it performs according to the locale data thereafter.

The obscure terminology, the use of class templates, and the fact that the locale model uses features of standard C++ not commonly implemented by compiler vendors degrades the usefulness of C++ locales. The close connection between the *iostream* library also encourages the use of coding practices that are not amenable to internationalisation[3]. Nevertheless Nathan Myers, the designer

[3] A notorious example is string concatenation using the << operator.

of the standard C++ locale model, points out in an article [Myers, 1998] that the locale implementation is lightweight and works well with multithreading. Because C++ is a superset of C, and many C++ compilers work double duty as C compilers, it is altogether possible that compiler and library vendors implement this part of the standard using the underlying C locales. However, because the C++ locale model is extensible with new facets, it is more flexible than the C model.

4.5 Windows locales

Microsoft Windows is by far the most widely used operating system platform for personal computers. Since the mid-1980's it has been available in many incarnations, progressing from 16-bit to 32-bit versions, and from a relatively simple GUI shell to a veritable multitasking operating system.

The earliest versions of Windows did not have any specific support for multilingual programs. However, with versions 3.0 and 3.1 Microsoft developed what is known as NLSAPI, or National Language Support API. Originally it consisted of 22 pieces of locale-related information used by a handful of system API calls. The system used a fixed default locale, and the information about other locales was not available for applications.

As Windows evolved into a 32-bit operating system, NLSAPI broadened in scope and became available to application programs in a systematic fashion. NLSAPI, as it is implemented in Windows 95/98, Windows NT/2000, and Windows CE, has information about date, time, numeric, and currency formats, calendars, the sorting of character strings and the classification of characters.

The advent of Windows 2000 broadened the scope of NLSAPI and introduced a full implementation of Multilingual API (MLAPI). This component had been in limited use since Windows 95, but reached its full potential with Windows 2000. MLAPI supports several different languages simultaneously, effectively separating the locales for input and output.

The Windows locale model was introduced with NLSAPI. Locales are identified with a locale id, or LCID. It is a 32-bit identifier indicating a *language*, a *sublanguage*, *sort type* and *sort version* as bitfields containing numeric identifiers for these components, as indicated in Figure 1. The *language* bitfield corresponds to a POSIX/C locale's language, while *sublanguage* actually corresponds to the country. *Sort* determines the string sorting system used for the language in question (such as German default ordering vs. German phonebook ordering), while *sort version* is not used. Locale IDs are generated from the components using the MAKELCID macro in the Win32 Platform SDK. They are used in locale-sensitive API calls.

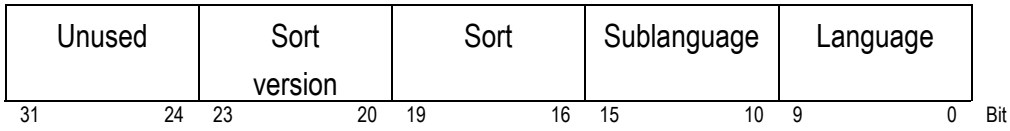

Figure 1. The bitfields of a Windows locale identifier (LCID).

Although the Windows API has some object-oriented features, it is still essentially a C language API. Because of this, NLSAPI is used through structures and function calls. NLSAPI defines several LCTYPE constants representing different locale-related values, such as the locale-specific "picture" of the date format and the type of the calendar used in the locale[4]. The LCTYPE constants are passed to NLSAPI functions along with the LCID and other flags.

LCTYPE constants are named using a typical Windows convention using the "Hungarian notation", where a prefix in a name indicates data type. Constants of the form LOCALE_*I*xxx represent integers, while LOCALE_*S*xxx formats represent strings. For example, LOCALE_IDATE represents the order of day, month, and year in a date string, and LOCALE_SCOUNTRY indicates a string storing the name of the country associated with the current locale.

The NLSAPI functions and their uses are outlined in Table 3. An all-purpose function called *GetLocaleInfo()* returns arbitrary locale information by LCID and LCTYPE. Specialised functions, such as *GetDateFormat()* and *GetCurrencyFormat()* handle locale-specific formatting of data items. However, as Schmitt [2000, 290] observes, these functions do not return the information used for locale-specific formatting, despite their names, but actually apply a format description to the data and return a formatted string.

There are also several enumeration functions such as *EnumCalendarInfo()*, which invoke an application-defined callback function to repeatedly receive data about their subject. They are used for example to list all calendars or all date formats supported in a locale.

[4] The term LCTYPE should not be confused with the LC_CTYPE concept of POSIX/C locales.

Locale category	NLSAPI functions
Locale information	`GetLocaleInfo()`, `IsValidLocale()`, `EnumSystemLocales()`, various language and LCID functions
Date and time formatting, calendars	`GetDateFormat()`, `EnumDateFormats()`, `EnumDateFormatsEx()`*, `GetTimeFormat()`, `EnumTimeFormats()`, `EnumTimeFormatsEx()`*,
Calendars	`GetCalendarInfo()`*, `EnumCalendarInfo()`, `EnumCalendarInfoEx()`*
Number and currency formats	`GetNumberFormat()`, `GetCurrencyFormat()`
Character classification	`IsCharXxx()`
Text processing	`CompareString()`, `FoldString()`, `lstrxxx()`, various string and character manipulation functions

* Available in Windows 2000

Table 3. Overview of the NLSAPI functions in Microsoft Windows.

Windows NT and Windows 2000 are based on the Unicode character set, but Windows 95, Windows 98, and Windows Me are not. For this reason there are two different sets of NLSAPI functions: the other set, with names ending in W, use Unicode "wide" characters in text parameters, while the other set of functions with A-names use the 8-bit ANSI character set. The Unicode versions are available in Windows NT and Windows 2000, and the ANSI versions are their counterparts in Windows 95, Windows 98, and Windows Me.

The separation of an application's resources from the program code is supported in Windows through resource files, which can be compiled and linked with the executable or used as a dynamic link library (DLL). Typically, the resources are compiled into separate DLL files for each language, but David Wendt [2000] describes a technique for using a single DLL for many languages. The resources are identified using the same LCIDs as with other features of the locale model. Resources include dialogs, bitmaps, icons, cursors, and text strings.

NLSAPI and the Windows locale model are a rich combination of culture-specific conventions and functions for applying them. With the advent of Windows 2000 and MLAPI, the locale model has been strengthened. Although the API is not object-oriented, it continues to serve Windows application

programmers well. It is also the basis for the internationalisation support for Windows CE, Microsoft's operating system for handheld devices.

4.6 Java locales

Since its introduction in 1995, the Java programming language has taken the software development community by storm. Interest in Java has been high from the very beginning, but the language has lived up to the hyperbole that has often surrounded it. Java has evolved into a general-purpose application programming language and an excellent academic tool.

In addition to the programming language, the Java concept also includes the Java platform, which defines an operating system and its APIs. However, pure Java operating systems have not entered the mass market. The Java platform has fragmented somewhat in recent years, and there are now variants of Java intended for enterprise computing, personal computing and handheld devices.

There are currently three major versions of Java: the original version 1.0, the much improved version 1.1 and the evolutionary Java 2. In Java 1.0, internationalisation support was not extensive, but starting with Java 1.1, it has risen to the level of Microsoft Windows and in some respects even surpassed it. The internationalisation API in Java is mostly licensed from Taligent, now a wholly owned subsidiary of IBM. This API contains a rich locale model and associated tools.

The Java programming language is based on the Unicode character set, which makes the implementation of the internationalisation API easy compared to systems that rely on narrower character sets. The API is based on utility classes that are collected in packages *java.util* and *java.text*. In addition, nearly all the graphical user interface components in the standard Java libraries are "locale-sensitive" in that they can adapt to the system's default locale or another explicitly set locale.

A central element of the Java internationalisation API is the class *java.util.Locale*. However, as David Flanagan has observed [1999, 524], this class does not implement any internationalisation features itself, but is used to identify a locale and get a description of it suitable for end users.

Locales are identified using the language and country codes standardised by ISO [ISO 1988a; ISO 1997], and an optional, vendor-specific variant code that can indicate a special version of the locale or an operating system. For example, since version 1.1.7 of the Java runtime environment there have been separate variants for the member countries of the European Monetary Union (EMU), which uses the euro as the locale's currency instead of the national currencies to be replaced by the euro in the year 2002.

The format of the Java locale identifier is *language_country_variant,* where *language* is the ISO standard two-letter language code, *country* is the ISO standard two-letter country code (upper case), and *variant* is the vendor-specific part. This is somewhat similar to the convention used in POSIX locales, as detailed in section 4.3, although Java has set a strong *de facto* standard. Table 4 lists some typical Java locale identifiers and their meanings.

Locale identifier	**Meaning**
`fi_FI`	Finnish (Finland)
`fi_FI_EURO`	Finnish (Finland) using the euro as the currency
`ja_JP_UNIX`	Japanese (Japan) for UNIX

Table 4. Examples of Java locale identifiers.

True to the nature of the language, the implementation of Java's locale model is thoroughly object-oriented. The actual implementations of locale-specific features are factored out to classes and interfaces in the *java.text* package and other packages, as outlined in Table 5. For example, character classification is a feature that is closely related to Unicode. Therefore it is implemented in the core class *java.lang.Character*. The classes that handle the formatting of locale-specific data are contained in *java.text*.

Java also implements the separation of an application's resources from the application code. This is achieved using "resource bundles", which are typically collections of text strings. As Dale Green points out in [Campione et al., 1998], these resource bundles can also include arbitrary Java objects which represent audio clips, images, and colours.

Locale-specific features	**Class or interface**
Locale information	`java.util.Locale`
Date and time formatting	`java.text.DateFormat`, `java.text.SimpleDateFormat`, `java.text.DateFormatSymbols`
Number and currency formatting	`java.text.NumberFormat`, `java.text.DecimalFormat`
Character classification	`java.lang.Character`
Calendars	`java.util.Calendar`, `java.util.GregorianCalendar`, `java.util.TimeZone`
Text processing	`java.text.Collator`, `java.text.RuleBasedCollator`,

	`java.text.BreakIterator`
Message formatting	`java.text.MessageFormat`, `java.text.ChoiceFormat`
Application resources	`java.util.ResourceBundle`, `java.util.ListResourceBundle`, `java.util.PropertyResourceBundle`

Table 5. The classes and interfaces of the Java internationalisation API.

Java is intended as a portable language with special importance on network capabilities. The designers of Java realised early on that support for internationalisation is an essential part of Java's suitability for worldwide software development. However, like with other platforms, the use of the internationalisation features are still more an exception than the norm. The awareness of these features is building slowly, but unlike Microsoft Windows, Java has been around for a relatively short time, and the learning curve can be daunting even without regard for internationalisation.

The Java locale model is made up of the features that currently have library classes to handle them. There are no restrictions on adding new locale-specific features to Java; indeed, IBM has demonstrated this by creating an open-source software package called International Components for Unicode for Java, or ICU4J [IBM, 2001]. It contains a more comprehensive implementation of calendar support, time zones, and Unicode utilities. The only requirement of a new internationalisation feature is that it is identifiable by a Java locale.

4.7 Towards a locale standard

Many of the locale models described above are in active use, but they are not official or international standards. They are *de facto* standards, established by software and operating system vendors, with the notable exception of C++ locales. The general notion seems to be that standardisation work especially in the field of software development always lacks behind current practice, because standards are designed by committees. Especially international committees need a lot of time for arranging meetings and preparing several drafts of standards. Standardisation committees also often get accused of being "out of touch" with current practices, and the preparation of international standards sometimes involves unfortunate compromises because of conflicting national interests.

Because of these reasons, it can be argued that standardisation work for internationalisation is as futile as other software standardisation efforts. How-

ever, standards are never harmful, and especially in software development standards are direly needed. Therefore it is notable that there are several international standards in preparation that are related to internationalisation.

The POSIX.2 shell and utilities standard (ISO/IEC 9945-2) defines a locale model, but it is only a small part of a large standard. The advent of the Unicode character encoding standard and its "official" international counterpart, ISO/IEC 10646, has caused the POSIX model to lag behind. ISO/IEC 14652, or "Information technology – Specification method for cultural conventions", is a standard in preparation that tries to remedy this. It is backwards compatible with POSIX, but it contains several enhancements listed in Annex A of the draft standard. Character information has been updated to use Unicode and ISO/IEC 10646, while there are entirely new categories for paper sizes, names, addresses and telephone numbers. Also the monetary category has support for multiple currencies such as the euro, and the time category supports alternate calendars and time zones.

Closely connected with the ISO/IEC 14652 is the draft standard ISO/IEC 15897 (in the last stages of development as of spring 2001), which defines a procedure for the registration of cultural elements used in the standard locale model. The aim of the standard is to establish a central repository for locale information in the ISO/IEC 14652 format. The initial registration authority is the Danish UNIX Systems User Group (DKUUG), which operates a registry at http://www.dkuug.dk/cultreg/. National standards bodies and other comparable organisations can submit entries to the registry.

There are several other international standards related to the cultural convention specification and the registry. For example, the ordering of character strings is one area where exact specifications are needed. ISO/IEC 14651 (in preparation) defines an international standard for ordering text strings, which is comparable to the Unicode Collation Algorithm [Davis and Whistler, 2001]. Furthermore, Küster [2000] describes the European ordering rules (EOR) developed by the European Committee on Standardization (CEN).

This chapter has introduced the concept of locale, which is crucial to internationalisation, and presented several contemporary locale models and current standardisation efforts. The next chapter describes the issues related to coded character sets used in text representation.

5. Character encodings

The representation of text is a fundamental property for many computing applications. Due to the large number of languages spoken in the world and the many different scripts used to represent them, there are several mutually incompatible computer representations of text in use.

This chapter presents the basic ideas and problems in the encoding of characters in text, and introduces the Unicode standard, which has had an enormous impact on internationalisation and localisation efforts.

5.1 Scripts and languages

There are dozens of different writing systems or *scripts* in use throughout the world. This work has been prepared using the Latin script, which is used in most of the Western world, with occasional examples in other scripts.

Scripts can be divided into *phonetic* and *ideographic* scripts. Phonetic scripts use *alphabets* to express sounds or *syllabaries* to express syllables. Ideographic scripts use *ideographs* to express morphemes, the smallest meaningful language units. Roughly put, alphabets and syllabaries consist of letters and syllabic characters, while ideographs are symbols that correspond to words or concepts.

Many languages share a script: English, German, French, Spanish and many others all use the Latin script, while the Cyrillic script is used by Russian, Ukrainian, and Bulgarian, among others. The alphabets used in languages sharing the same script are often different, although they overlap considerably.

The direction of writing in the Latin, Cyrillic, and Greek scripts is exclusively left-to-right. In the Arabic and Hebrew scripts writing proceeds primarily from right to left, but the text may contain elements, such as numbers and Latin text, that are written from left to right. For this reason Arabic and Hebrew scripts are called bi-directional or "BiDi" scripts. Figure 3 shows an example of an Israeli web page rendered by Microsoft Internet Explorer, with several instances of bi-directional text.

Traditional Japanese is written from right to left and from the top down, so that "lines" of text are actually columns. Modern Japanese and Chinese are written left-to-right like the Latin script.

Some scripts are compositional, meaning they are composed of basic and additional character forms. For example, the Thai script consists of independent vowels and consonants, but has additional combining vowels, tone marks, and diacritics. Another example of a compositional script is *hangul* used in Korean, where elements called *jamo* are arranged inside a square cell in a strict order.

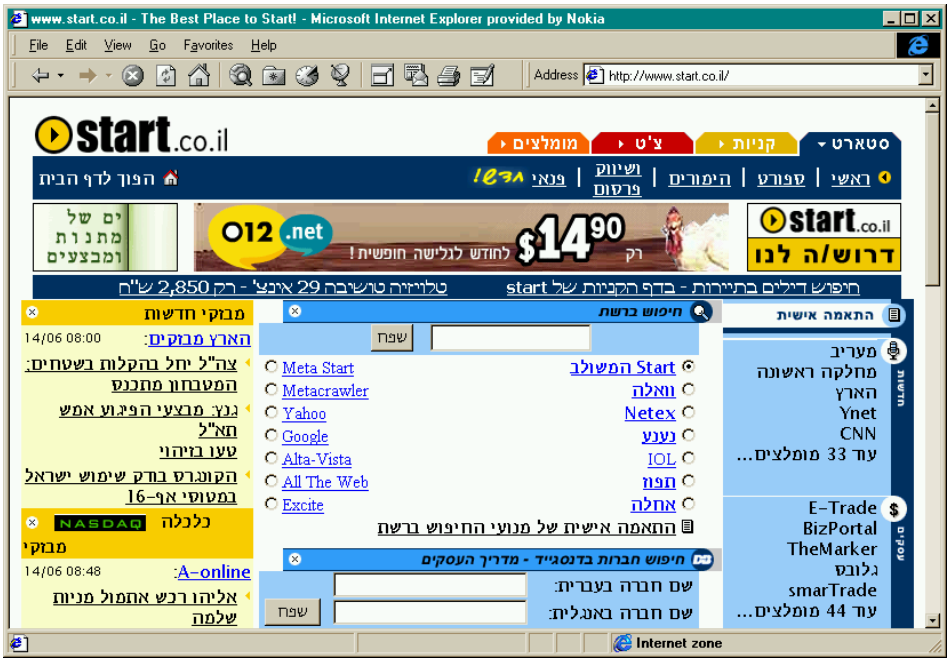

Figure 3. An Israeli web page provides an example of bi-directional text.

5.2 Characters, glyphs, and fonts

In the context of text processing, *characters* are the constituent parts of any piece of text. The Unicode standard version 3.0 defines a character as "the smallest component of written language that has semantic value" [The Unicode Consortium, 2000]. Characters refer to an abstract meaning or shape, such as "the Latin capital letter A" or "the euro sign". These meanings are independent of the actual representations, which vary considerably in style.

The actual representations of characters are usually called *glyphs*. The previous example of a character, "the Latin capital letter A" can be represented as an italic, boldface, underlined, or some other form, size, and colour. The underlying semantics of the character are not changed with the representation. Hence, there is a one-to-many mapping from characters to glyphs. Likewise, a glyph can map to several characters, especially in the case of *ligatures*, which present two or more characters with the same glyph. A *font* is a collection of glyphs that share a similar visual representation, and typography is the art of designing aesthetically pleasing fonts.

5.3 Coded character sets

For the purposes of representing texts, we need information about which characters are required in the representation. Together these characters form a *character repertoire*. For computer processing it is necessary to assign a numeric value for each of the characters in the repertoire. The result of this assignment

is a *coded character set*. (The term *character encoding* is used synonymously.) The numeric values assigned to characters are nonnegative integers, and the technical name for the value is *code point*.

Dozens, if not hundreds of coded character sets have been established and used since the advent of computers. The best known is undoubtedly ASCII (American Standard Code for Information Interchange), which can be used to represent English text using a 7-bit code for each character. As the need grew to represent other languages besides English, the limitations of ASCII became apparent.

Several vendor-specific variants of ASCII were developed for languages using the Latin script and an 8-bit encoding. In Japan, China, and Thailand different 8-bit character sets became to use through national standardisation efforts. ISO became active in character set standardisation in the 1980s, developing a series of 8-bit character set standards in the ISO 8859 series. The languages covered by these standards are listed in [ISO 1999c]. The ISO 2022 series provides a set of standards for encoding Chinese, Japanese, and Korean characters. For these languages simple 8-bit character sets are not enough, so multiple byte code sets are used. The character code points in such code sets are either one, two, or three bytes, which makes them somewhat difficult and error-prone to process.

Operating system vendors, such as IBM and Microsoft, have traditionally defined their own coded character sets, adding to the general confusion about character encodings and conversions between them. Currently coded character sets are registered with the Internet Assigned Numbers Authority (IANA), who issues a numeric identifier and a standard MIME name for each registered character set.

The biggest problem with coded character sets has always been their relative incompatibility with each other. While the ISO 8859 character sets have ASCII as their subset, the remaining code points are assigned to different characters in different parts of ISO 8859. Because of this multiple languages cannot easily be represented in the same document. The problem is even worse in different encodings for Asian languages, which do not necessarily have any common subset.

5.4 Unicode

The Unicode character encoding standard [The Unicode Consortium, 2000] was designed to overcome the limitations and problems of having various different coded character sets. According to Tony Graham [2000], the project began in 1988, and the Unicode Consortium was formed in 1991 by IT industry members to address the need for a unified coded character set with a sufficiently large

character repertoire for worldwide use. Some time earlier ISO had started the preparation of a similar character set standard, ISO 10646. The Unicode Consortium and ISO decided to merge the efforts in 1991, and currently Unicode and ISO 10646 are fully compatible.

The aim of Unicode is to define a character repertoire that covers all major scripts and languages used in the world and a coded character set for representing text in all of them. This is made possible through the use of 16-bit code points instead of only 7 or 8 bits. The latest version of the Unicode standard, version 3.0, contains 49,194 characters out of the 65,536 possible using 16 bits. The code point space can be further expanded with the use of surrogate pairs, making possible the addition of over 900,000 additional characters[5]. The characters in the repertoire are selected to cover all the character sets in widespread use today.

In addition to the worldwide character repertoire, the Unicode standard contains implementation guidelines for text processing tasks, such as sorting and searching for text strings and mapping characters to other characters. The Unicode Consortium also issues annexes to the standard and technical reports that address issues such as bidirectionality and collation, providing reference implementations for software developers.

5.5 Unicode and internationalisation

If the internal character encoding of an operating system is a vendor-specific coded character set or a single standardised character set, there will be problems in creating localised versions of the operating system and the applications written for it. As Tony Graham [2000, xxix] observes, each localised version of the system and its applications need to have a different, regional character set as the native encoding. This causes problems with filenames, document interchange, and electronic communications, among others.

The use of a single, worldwide coded character set makes localisation efforts much easier, because Unicode contains all the characters that are required in regional versions. The operating systems and applications still need to be localised, but the use of Unicode greatly reduces the possibility of errors caused by incompatible character encodings.

Probably the most important internationalisation design decision for an operating system is to support Unicode. The system still needs to support traditional single byte and multiple byte character sets, so Unicode support alone does not solve all internationalisation and localisation problems. However, it

---

[5] Unicode 3.1 will add 44,946 new characters, most of them through the surrogate mechanism.

makes them much more manageable. The most widely used Unicode-enabled operating system is currently Windows NT 4.0, with Windows 2000 following suit. The Java platform was designed to support Unicode internally from version 1.0.

This chapter has presented the notions of characters, glyphs, and fonts together with character repertoires and coded character sets. The Unicode worldwide character encoding standard was presented, and its importance for internationalisation and localisation was analysed. The next chapter describes the requirements for an internationalised operating system.

6. Internationalisation requirements for operating systems

The design and implementation of international software presents many challenges for software designers and software engineers. Internationalisation brings along many concerns that have not traditionally been part of the software engineering process. However, like regular software features, these concerns can be expressed as requirement specifications. This chapter presents the most common requirements for internationalisation support in an operating system and explores them in detail. It also presents the division of internationalisation requirements into algorithmic and content-based.

Requirement specifications need to be written in the earliest stages of software development [Pressman, 2001]. They often precede any design or technical specifications, because the requirements dictate the minimum functionality of software. Requirements are also often drafted in close cooperation with marketing, because they are essential in the enabling of features that are desired in a software product.

Even though it were desirable to create an operating system or application that handled the input and output of all languages in the world and supported every conceivable locale and character encoding in existence, it would not be practical in most cases. Thus requirements for internationalisation are usually based on regional, market-driven needs. Typically there are not enough resources to develop software that is internationalised beyond the immediate market needs. However, adding international features to software late in its lifecycle can lead to repetitive development work and increased need for testing.

6.1 Algorithmic and content-based requirements

Although a locale model described in Chapter 4 is very important and also highly practical in terms of implementation complexity, it is not an all-encompassing solution for internationalisation. Not all cultural conventions are easily described as discrete units such as the component parts of a locale definition. Many require narrative descriptions to guide the implementation, as in the case of linguistic elements and components of the visual user interface.

Furthermore, even essentially discrete elements such as conversions between character encodings are not easily described using static data. While some character encoding conversions can be achieved using simple mapping, others require the application of algorithms. The same applies to calendars, which often employ astronomical calculations, even though they also operate on historical and cultural data.

For these reasons it is feasible to divide the requirements for internationalisation into two categories, *algorithmic* requirements and *content-based* requirements. These two differ in the nature of the implementation. The categories are outlined below with specific examples, and elaborated on later in this chapter. In this thesis the focus is on the algorithmic requirements, because support for them is more difficult to implement without operating system capabilities than support for content-based requirements. Furthermore, the aim of this work is to measure the level of internationalisation support in handheld operating systems as opposed to handheld applications.

6.1.1 Algorithmic requirements

Algorithmic internationalisation requirements are those that can be fulfilled with a computation requiring an algorithm, either sophisticated or simple. The desired culture-specific representation of data is achieved by invoking the algorithm.

An example of an algorithmic internationalisation requirement is the textual formatting of a calendar date. This involves an element of raw data, namely the actual point in time, which is usually stored as the amount of milliseconds elapsed from some particular epoch. However, the textual representation of the date varies greatly from one locale to another, usually containing the day, the month, and the year in numeric form.

The characters used to separate the different components of the date, the inclusion or omission of leading zeros in numbers, and the choice between two-digit and four-digit years are all factors that need to be encapsulated. Figure 2 shows Java code for transforming elapsed time (in milliseconds since midnight, January 1, 1970 UTC) into a representation that is culturally appropriate for Finland.

Typically the sort of transformation implemented by the Java code shown is achieved by using a specification template interpreted by an algorithm. It contains placeholders for several replaceable items that are substituted with their actual values at runtime. Most importantly, the algorithm itself is unmodified in each case, but its behaviour is controlled by the specification template.

```
import java.util.Date;
import java.util.Locale;
import java.util.Calendar;
import java.text.DateFormat;

public class DateFormatSample {
    public static void main(String[] args) {
        long current = System.currentTimeMillis();
        System.out.println("Current time: " + current + " ms");

        Locale currentLocale = new Locale("fi", "FI");
        Date now = new Date(current);
        Calendar cal = Calendar.getInstance(currentLocale);
        System.out.println("Date components:"
            + " y = " + cal.get(Calendar.YEAR)
            + " m = " + (cal.get(Calendar.MONTH) + 1)
            + " d = " + cal.get(Calendar.DAY_OF_MONTH));
        DateFormat shortDate = DateFormat.getDateInstance(
            DateFormat.SHORT, currentLocale);
        System.out.println("Finnish short date: " + shortDate.format(now));
        DateFormat longDate = DateFormat.getDateInstance(
            DateFormat.LONG, currentLocale);
        System.out.println("Finnish long date : " + longDate.format(now));
    }
}

Output of program:

Current time: 992510563406 ms
Date components: y = 2001 m = 6 d = 14
Finnish short date: 14.6.2001
Finnish long date : 14. kesäkuuta 2001
```

Figure 2. Java code to transform the elapsed time into a culturally appropriate date representation.

6.1.2 Content-based requirements

Content-based internationalisation requirements are those that relate to the system's user interface and its contents. These differ from algorithmic requirements mainly in that they are not implemented using an algorithm as such, but achieved through text, images, colour, and tones.

The architecture of internationalised software needs to facilitate the changing of user interface elements according to their cultural suitability. Text, for example, is the most obvious form of user interface content that needs to be changed when the software is adapted to different cultures. The visible texts in the user interface needs to be translated into other languages. The text need not even be visible; if speech synthesis is used, the input to the synthesiser also has to be translated.

With regard to internationalisation, images, colours, and tones are more subtle forms of user interface content than text. All of these must be adapted to the target culture, and del Galdo and Nielsen [1999] have collected many anecdotes about cultural misconceptions related to the inappropriate use of all these

kinds of elements. These inappropriate uses are also potentially disastrous for the image of the software product in the users' minds.

The adaptation of the actual user interface content is the responsibility of the localisation effort. However, if the capability for adaptation has not been built in the software, the localisation effort will be unnecessarily hindered or even made impossible. International software must provide a way of changing all culture-specific user interface content. This can be implemented by adopting company-wide conventions in the construction of user interface resources. Support from the operating system makes the effort considerably easier.

An interesting special case that lies between algorithmic and content-based requirements is the support for the "mirroring" or "flipping" of the user interface. This means supporting languages where the primary direction of writing is right-to-left, such as Arabic and Hebrew, instead of left-to-right as in Western languages. This capability is discussed in more detail below.

6.2 Requirement breakdown

This section discusses the internationalisation requirements for an operating system or platform in detail. The requirements have been categorised according to desired functionality and user experience.

6.2.1 Cultural convention support

Adherence to cultural conventions in marking down everyday items is the most obvious requirement for internationalisation. These items include dates, times, numbers, currency amounts and measurements, and also addresses, phone numbers, names and titles of people, and salutations. The calendar used in a culture and its different features are also strong cultural conventions.

An internationalised system must support various formats for representing dates and times. Although ISO has standardised the date format in [ISO 1988b], this format has not surpassed the national conventions. Times are expressed using either a 12-hour or 24-hour clock, which in the former case leads to the use of regionally varying symbols for A.M. and P.M. There are also variations in whether day and month numbers below ten are written with or without leading zeros. Finally, the names of weekdays and months need to be translated to the target language. Table 6 collects several numeric representations of the date 5th September 2000 and the time 9:24 A.M.

The representation of numbers also varies greatly around the world. Some countries use a decimal point to separate whole part and the decimal part of a decimal number, while some use a comma instead. In some countries the thousands in numbers are grouped using commas, while in others a (non-breaking) space character is used.

Locale	Long date	Short date	Time
Finnish (Finland)	5. syyskuuta 2000	5.9.2000	9:24
Hungarian (Hungary)	2000. szeptember 5.	2000.09.05.	9:24
English (U.S.)	September 5, 2000	9/5/00	9:24 AM
English (U.K.)	5 September 2000	05/09/00	9:24 AM
Polish (Poland)	5 września 2000	00-09-05	9:24
Russian (Russia)	5 сентября 2000 г.	05.09.2000	9:24
Japanese (Japan)	2000年9月5日	00/09/05	9:24

Table 6. Examples of different national date and time formats.

Currency values are often represented with a corresponding currency symbol. Some examples of these symbols are shown in Table 7. In the absence of a dedicated currency symbol a longer textual element, such as "mk" or "kr", is used. In banking and finance the ISO standard three-letter currency code is often used. The currency symbol either precedes or follows the currency amount, depending on national conventions. Sometimes there is a blank separating the currency symbol from the amount, sometimes not. By convention the ISO code always precedes the amount, separated by a blank.

£	$	¥	฿	€
GBP	USD	JPY	THB	EUR

Table 7. Examples of currency symbols and their corresponding ISO currency codes.

The representation of negative currency amounts has regional varieties as well. In some countries, the negative currency amount is put inside parentheses, while in others it is represented in red. The position of the minus sign also depends on local conventions.

Table 8 shows examples of numbers and monetary amounts in different locales. Note the varying separator characters and the positions of the currency symbols.

Calendars and their different uses are also strong cultural conventions. The Gregorian calendar is used in Western countries, but there are several other calendars in widespread use, such as the Islamic calendar, the Hebrew calendar, and the Chinese lunar calendar. The Japanese calendar is almost identical to the Gregorian calendar, but the years are numbered from the start of the cur-

rent emperor's era. All these calendars have many common features, such as the seven-day week, but the required calculations and the presentation of the calendar are quite different from each other.

Locale	Number format	Currency format
Finnish (Finland)	1 234,56	1 234,56 mk
Hungarian (Hungary)	1 234,56	Ft1 234,56
English (U.S.)	1,234.56	$1,234.56
English (U.K.)	1,234.56	£1,234.56
French (France)	1 234,56	1 234.56 F
French (Switzerland)	1'234.56	SFr. 1'234.56
Italian (Italy)	1.234,56	L. 1.235
Thai (Thailand)	1.234,56	฿1,234.56

Table 8. Examples of national number and currency formats.

Even within the realm of the Gregorian calendar there are usage differences. Some countries start the week on Sunday, others on Monday. In some countries week numbers are used, especially in business, while in others there is practically no concept of week numbering. There are also several variants of the Islamic calendar. Especially time management software needs to know about the default calendar used in the operating system and its regional features.

Country-specific conventions must also be applied to postal addresses and phone numbers. Middle initials and ZIP codes are used in the United States, while house numbers are appended to street names in most European countries. Postal addresses in Russia are written with country first and addressee last. Digits in domestic and international telephone numbers are often grouped differently from each other, with different separator characters between groups. Titles in names are mandatory in some countries, while in others the family name is always written before the first name. Examples and special cases are numerous.

Finally, even some of those countries that officially use the metric system still apply the old imperial system, with feet and inches used instead of meters and centimetres. Software that processes measurements must be prepared to apply different formatting or conversions to them according to the target culture. The underlying operating system must provide ready-made functions for this.

Different countries even use different sizes of printing paper: most European countries use the DIN standard A series sheets (A3, A4 and others), while

U.S. Letter and U.S. Legal sizes are the norm in the United States, and Japan uses JIS standard size sheets. The printing subsystem and the printer drivers of the operating system need to provide the possibility for users to select and for application programmers to detect and change the paper size used.

6.2.2 Multiple script support

As described in section 5.1, different languages use various different scripts. The support for these scripts often poses a problem for systems that are not internationalised.

Most operating systems contain a text rendering subsystem responsible for the drawing of text on the screen or producing an image on the printed page. This subsystem must support different scripts and different directionalities of text. One way of achieving this is using Unicode in the storage of text and implementing relevant Unicode-related algorithms, such as the Unicode Bi-directional Algorithm [Davis, 2000].

To fully support different scripts, the fonts used in the system must also contain all the required glyphs for rendering any relevant characters. Recently the use of Unicode fonts has increased, but at the same time the size of the fonts has become a serious problem especially in handheld devices. Both outline and bitmapped fonts for Unicode require an inordinate amount of storage space. This problem will be analysed further in Chapter 9.

Another problem with supporting different scripts is the difference between the logical order of text and its visual rendering. There may not be a one-to-one correspondence between characters in memory and characters on screen. The text rendering subsystem should not assume that the memory representation and the screen image are identical.

6.2.3 Support for multiple input methods

In most computing devices user interaction happens primarily using a keyboard. Auxiliary devices such as computer mice rarely require locale-specific adjustments, but keyboards need to accommodate different languages and conventions. This is achieved through support for multiple, language-specific keyboard layouts. The printed indications on the keycaps are changed, and the keys are mapped to produce different characters depending on the configuration of the system.

Although keyboards are becoming more prevalent in handheld devices, many handheld devices still have no keyboard in the traditional sense, and many never will. Some are instead controlled through a keypad with far less keys than are available in a keyboard, and others use a touch-sensitive screen

and a stylus or "pen" to apply slight pressure to the screen surface. The keys of a keypad can be mapped to produce different characters, just like the keys of a keyboard. The use of a stylus, however, often brings along another means of input, namely handwriting.

Handwriting recognition has been available in handheld devices since the ill-fated Apple Newton in the early 1990s. Handwriting technology has progressed enormously since the Newton, but most systems still require users to learn a specialised "handwriting alphabet" so that the device actually can recognise input. Handwriting input is often tedious and error-prone, but it also needs adaptation for different languages. The letterforms that are recognised should not be limited to only ASCII characters, like they often are.

Some scripts and languages cannot be supported with traditional keyboard input methods or even handwriting recognition. For example, the Chinese language has tens of thousands of ideographs. It would be nearly impossible to build a keyboard that contained them all, although Lunde [1999, 217] states that keyboards with thousands of keys do exist. It is also very difficult to create a handwriting recognition system that recognises all or even most of the ideographs. The solution is to implement an input method editor (IME), also known as a front-end processor (FEP) that uses a database of ideographs and allows the user to narrow down the selection of an ideograph until the desired one is found and the selection is confirmed.

Lunde classifies input methods of ideographic scripts into direct and indirect. Direct methods require the operator to know the numeric codes for different characters in some selected character encoding, while indirect methods usually use transliteration into Latin characters. Direct methods are rather inefficient and of limited use, but several solutions have been developed for indirect methods. For example, Microsoft Internet Explorer 4.0 and later, as well as Microsoft Office 2000, have input method editors for so-called CJKV languages (Chinese, Japanese, Korean, Vietnamese).

Compositional scripts require yet different input methods. For example, the Thai language has strict rules regarding the placement and allowed sequences of characters. Therefore Thai requires a special input method different from the ideographic transliteration methods described above. A specification of a Thai input method called WTT 2.0 is found in [Koanantakool, 1993]. Korean character sets typically use a number of precomposed hangul characters, but these are often not enough, so the system must support the composition of *hangul* characters from *jamo*.

The operating system obviously needs to support input methods for other than Latin scripts. In some cases this can be achieved with a suitable keyboard

layout, but for ideographic and compositional scripts it is more practical to include an input method editor for entering transliterated text. This usually requires an additional conversion dictionary that allows transliterated input strings to be converted into CJKV ideographs.

6.2.4 Graphical user interface elements

Elements in a graphical user interface usually present text to the user: text fields, buttons, list boxes and combo boxes have text strings that need to be localised. The components need to adapt to the text that is contained in them. Because text can expand in translation (usual estimates are 30-300%), the components may need to be resized dynamically in the horizontal direction.

Ideographic and compositional scripts may cause also vertical expansion. For example, Chinese characters are usually taller than Latin letters because very small character sizes are not legible. The Thai script expands vertically because tone marks may be inserted on top of the base characters.

When using bi-directional scripts the orientation of the user interface may need to change. In the user interface of an Arabic and Hebrew application text flows mainly from right to left, and this also causes the need to "mirror" or "flip" the user interface. This means changing the orientation and visual appearance of the user interface components. In the U.S. and European versions of Microsoft Windows, for example, the scrollbars are typically on the right side of the window being scrolled. In Arabic and Hebrew versions of Windows they are on the left, while checkmarks and radio buttons are on the right side of their associated text instead of left, cascading menus open from right to left, and so on.

6.2.5 User interface content

Text, images, sounds, and colours used in applications are generally referred to as user interface content. The adaptation of this content makes up the bulk of the localisation process. However, the underlying operating system needs to provide means of segregating the user interface content from the application code so that it can be easily replaced with content that is adapted to the target locales. If the user interface content is embedded in the program source code, the process effectively degenerates to customisation (detailed in section 2.2).

The usual method of segregating user interface content from program code is the use of *resource files*. "Resource" is the common name for all changeable user interface content. The term probably originates in the Microsoft Windows environment, where text, dialogs, menus, bitmaps, and cursors are referred to as resources. This practice has since spread to many other environ-

ments as well. POSIX systems use message catalogs, and Java uses resource bundles.

The resources are collected into a separate file (or several files) that are then optionally compiled into binary form with a resource compiler and linked into the final application. The process varies from one operating system to another, but the underlying idea remains. The creation of new language variants of application is made significantly easier because the original program code does not need to be modified, provided that the code has been internationalised to use program resources in all situations.

6.2.6 Text processing

The processing of text is fundamental to most applications. One of the first applications of the personal computer was "word processing", and nowadays also many handheld devices provide applications that allow the user to manipulate textual content. Mobile phones have for several years had the capability of composing and sending short messages (SMS), and entries in the mobile phone's phonebook are short text items.

Naïve conceptions of text processing may regard it as trivial, and the handling of English text encoded in ASCII usually is. However, from an internationalisation point of view text is more than homogenous character strings. Different scripts and different languages bring along many complications.

One of the most problematic issues in text processing is the character encoding used to store the text. (Character encodings and character sets were discussed in detail in Chapter 5.) The most practical method of handling different character sets is to use Unicode as the operating system's native character encoding and support conversions to and from other encodings as required.

The Latin script supports the notion of upper and lower case characters, which are used in many languages as a means of distinguishing words from one another. For example, names are usually capitalised, but so is every noun in the German language. Then again, ideographic scripts do not have the concept of character case. Apart from letters and ideographs, among other distinctive character classes are digits and punctuation. The operating system needs to provide means for making these distinctions for application programs that require this information.

The detection of word and sentence boundaries differs between scripts. In languages using the Latin script words are separated by blank spaces, and sentences are separated by a full stop or other punctuation. Ideographic scripts do not use word delimiters, so boundaries between words or concepts need to be established by other means, such as using dictionary lookup. These boundaries

also determine appropriate points of breaking a line of text. The operating system must provide a locale-specific way of detecting word, sentence, or line boundaries.

The ordering of text strings is dependent on script and language. The generic term *collation* is used in reference to the sorting of text items. The Unicode standard provides a default ordering of characters, but it must usually be tailored to use locale-specific rules, which may change between locales even if the language used is the same. The operating system needs to provide functions for locale-specific text comparisons. These are often part of the locale model detailed in Chapter 4.

This chapter has presented the internationalisation requirements of an operating system in detail. The next chapter describes typical implementations of these requirements. A metric for measuring the internationalisation support in an operating system is also developed.

7. Operating system support for internationalisation

When internationalisation requirements are implemented at the operating system level, they provide the most benefit to application development, as argued in section 3.3. This chapter maps the internationalisation requirements presented in Chapter 6 into features typically found in operating systems. Metrics for determining the internationalisation level of an operating system are also developed. These metrics are used in Chapter 10 to evaluate the extent of internationalisation in the handheld operating systems selected for this study.

7.1 The locale model

As we have seen in Chapter 4, cultural conventions are often supported with a locale model. The implementation of the locale model consists of a locale database and an API for accessing locale-specific data. The API must include at least the following functionality:

- A function to identify the system's current locale setting.
- Functions to query the locale database for information.
- Functions to prepare locale-specific representations of data.

Optional functionality can include functions to reset the system's locale settings and construct new locales.

The implementation of the locale database can range from a simple collection of text files to a relational database with query facilites. Often the locale information is compiled into a binary format for storage efficiency.

7.2 Support for Unicode and other character encodings

Since the widespread adoption of the Internet, computing environments have become increasingly international. Textual data that is transferred between computing devices can be encoded using many different character encodings. To successfully communicate with other computer systems in the network, operating systems and application programs need to support various coded character sets and conversions between them. This process is often referred to as *transcoding*.

The development of Unicode has not removed the need to support "legacy" character sets, but the increasing use of Unicode in major operating systems and applications is promising in this respect. Another useful property of Unicode is its capability of acting as a "pivot point" for transcoding. This technique is described by Tony Graham [2000, 103].

Support for different character encodings is typically implemented as an API to an underlying character conversion engine. In most cases this engine can be dynamically augmented with new conversion facilities as needed, although the system should initially support as many encodings as possible. The API may provide functionality for transcoding the contents of a buffer in memory, or the conversion may be integrated into the I/O facilities of the system.

7.3 Support for application resources

The notion of application resources was described in section 6.2.5. For applications to take advantage of user interface content and other resources that are detached from the actual program logic, the underlying operating system needs to support the following:

- Detachment of resources from application in development phase.
- Access to resource elements at runtime.
- Dynamic processing of resource elements, such as updating user interface controls.

Operating system vendors usually provide application developers with development kits which contain utilities to process resource files. The guidelines for preparing application resources are included in the relevant literature.

The ability to access application resources at runtime is essential because they contain the localised user interface elements of an application. The application logic need not change, but the content of the user interface does change according to the target culture. The application program loads the required resources as needed.

As resources are loaded at runtime, the application must be able to initialise or reset the user interface controls with the new content. Often this is done as the application program starts, but the dynamic loading of code at runtime requires that the program be able to access the resources at any time and that changes are immediately visible to the end user.

7.4 Text rendering and fonts

Practically all modern operating systems, especially those for handheld systems, have a graphical user interface. This greatly enhances the possibility of displaying glyphs for different characters of different scripts correctly. Text in graphical user interfaces is based on either bitmap or outline fonts.

Glyphs in bitmap fonts consist of rigid rectangular cells that determine a set of lit and unlit pixels on the screen. Bitmap fonts are restricted to one size only;

if a different size is required, another bitmap font with a new cell size needs to be created, or the font needs to be scaled, often with poor visual results.

Outline fonts, on the other hand, contain instructions for rendering the glyphs at practically any point size. Font technologies such as Adobe Type 1 and Microsoft TrueType have made it possible to implement high-quality, scalable typefaces for computing devices.

Since bitmap fonts contain a complete pixel matrix for each glyph, they usually require much more memory than outline fonts. This problem can be alleviated with compression algorithms. Ultimately the operating system's graphical output code draws even outline fonts as single pixels. If the screen resolution is low, this may cause jagged edges to appear in the glyphs on the screen. The removal of these edges, called anti-aliasing, can be augmented using rendering hints included in outline fonts.

With both font types the most important issue for internationalisation is the code point mapping of the glyphs in a font. The font specification must contain information about the code point assigned to each glyph so that the desired glyph can be found. In operating systems that use Unicode as their native encoding, glyphs are usually mapped to the corresponding Unicode code points. In non-Unicode operating systems, the glyph mapping conforms to the default encoding of the system, such as ISO 8859-1 or WGL4.

Because Unicode defines such a large amount of characters, it is often not feasible to build fonts that glyphs for all of them. The size of the font quickly grows too big to be practical, especially in handheld devices. For these reasons operating system vendors need to carefully consider which glyphs are actually needed and then build fonts that have well-defined subsets of glyphs. The font rendering subsystem must also support ligatures that are used to replace common combinations of letters for typographic and aesthetic reasons. In the Arabic script ligatures are essential, because Arabic letters have several contextual forms.

7.5 Text processing support

The most feasible way for an operating system to support multiple scripts is to implement the Unicode standard. As described in Chapter 5, Unicode is more than a large repertoire of characters. Related specifications of algorithms issued by the Unicode Consortium, such as the Unicode Bidirectional Algorithm and the Unicode Collation Algorithm, are practical models for implementers.

The minimal unit of text processing is the character. The Unicode standard provides a character model that treats characters as 16-bit entities with an assigned code point in the Unicode character set. Each character is associated with many properties that give information about the character. The operating

system needs to make this information available to applications through an API. This functionality may be implemented as part of the locale model.

Text is formed of characters – letters, digits, punctuation and separators. Text can be partitioned into different meaningful units depending on the language and the script used. The operating system must provide means of detecting boundaries between these units in text. With simpler scripts this can be achieved by detecting punctuation characters in the text, but with ideographic characters there are no discernible word breaks to be detected. Because of this, rules for line breaking are similarly complicated. The usual solution is to use a dictionary and look up units of text as they are processed to determine suitable line breaking points.

For many text processing applications it is important that items of text are sorted correctly in terms of the user's language and culture. The sorting routines can be provided in the operating system, provided that they take into account the characters used in a language and the locale-specific conventions. Unicode provides a reference algorithm for collation, but it needs to be tailored for each locale. Sorting is more complicated with ideographic scripts, because languages such as Chinese and Japanese do not have alphabetical ordering like languages that use the Latin script. There are several established orderings for ideographic scripts [Lunde 1999, 440] but they often require large look-up tables.

7.6 Input method support

The problems in implementing support for traditional input methods such as keyboard entry are already well understood. They usually arise when trying to decide which keys should be available on the keyboard. Because the same physical keyboard layout is often used in all product versions, there are a limited number of keys available.

Tuthill and Smallberg [1997, 105] emphasise that an internationalised application cannot assume any particular mapping between keystrokes and input characters. Characters that belong to any supported codeset but are not directly available on the keyboard are often created by composition together with modifier keys such as AltGr or Compose.

Theoretically handwriting recognition poses a larger problem, because there appears to be an infinite number of strokes to use. In practice, however, the number of distinguishable strokes is much smaller, and because handwriting algorithms often rely on special abbreviated or approximated strokes, the number of usable strokes is even smaller.

In order to support input methods for different scripts, the operating system must have an input method architecture. One feasible design approach

uses the client/server model. The input method server supplies client programs with information about the desired language and script. Client programs may then use any or all of the following methods, as classified by Lunde [1999]:

- Direct keyboard input for Latin and other non-ideographic scripts.
- Transliterated Latin input for ideographic scripts.
- Structural input for ideographic scripts: by radical, by number of strokes, and other methods.

Most input methods for ideographic scripts require a conversion dictionary, which the input method server uses to locate characters corresponding to the input strings supplied by the input client. Few standards exist for the format or content of these dictionaries.

Microsoft has developed an architecture called Global IME (Input Method Editor), which supports Chinese and Japanese. It is not connected to the standard Input Method Manager (IMM) of Microsoft Windows, and therefore only works with a limited set of applications. Similarly Java 2 defines an Input Method Framework with a client API and a service provider interface (SPI) for creating new input method editors.

7.7 Proposed metrics for internationalisation

According to Roger S. Pressman [2001, 81], the IEEE Standard Glossary of Software Engineering Terms defines a metric as "a quantitative measure of the degree to which a system, component, or process possesses a given attribute". This definition of a metric is used in this study to evaluate the internationalisation support in a handheld operating system. The aim is to develop a set of metrics to measure the amount of features related to the development of international software.

Table 9 lists the internationalisation features itemised at the beginning of this chapter and assigns to each a set of values that describe the degree of its implementation in a handheld operating system.

Internationalisation feature	Values	Max. pts
Locale model		
Locale identification	Yes/No	1
Locale information database	Yes/No	1
Locale database queries	Yes/No	1
Cultural conventions		
Date and time representations	API/Available/None	2
Number and currency representations	API/Available/None	2

Internationalisation feature	Values	Max. pts
Unicode support		
Character properties	API/Available/None	2
Unicode Collation Algorithm	Integrated/Available/None	2
Unicode Bidirectional Algorithm	Integrated/Available/None	2
Standard Compression Scheme	Integrated/Available/None	2
Character encoding support		
Multiple character set support	Extensible/Fixed/None	2
Transcoding between character sets	Yes/No	1
Autodetection of character sets	Yes/No	1
Application resources		
Detachment from application code	Yes/No	1
Runtime access through API	Yes/No	1
Supported scripts		
Latin	Yes/No	1
Greek	Yes/No	1
Cyrillic	Yes/No	1
Ideographic (Chinese)	Yes/No	1
Compositional (Thai, Korean)	Yes/No	1
Bi-directional (Arabic, Hebrew)	Yes/No	1
Fonts and text rendering		
Bitmap fonts	Yes/No	1
Outline fonts	Yes/No	1
Ligatures	Yes/No	1
Text processing		
Text boundary detection	Yes/No	1
Locale-specific collation of text strings	Yes/No	1
Input method support		
Input method architecture	Extensible/API/None	2
Direct input	Yes/No	1
Transliterated input	Yes/No	1
Structural input	Yes/No	1
User interface layout		
Dynamic resizing of UI components	Yes/No	1
Selectable orientation of UI components	Yes/No	1
Total		**39**

Table 9. Internationalisation features and their associated values.

Each set of values has an associated score in points, determined by table 10.

Values	**Points**
Yes/No	Yes = 1, No = 0
API/Available/None	API = 2, Available = 1, None = 0
Integrated/Available/None	Integrated = 2, Available = 1, None = 0
Extensible/Fixed/None	Extensible = 2, Fixed = 1, None = 0
API/None	API = 1, None = 0
Extensible/API/None	Extensible = 2, API = 1, None = 0

Table 10. Scores associated with the values of internationalisation metrics.

The maximum total score of all the metrics is 39 points. The value "API" means that there is support on the API level for a particular feature. The associated data may be "Available" directly, or some functionality may be "Integrated", meaning that its use is transparent to applications. If a feature is "Extensible" (as opposed to "Fixed"), it indicates that the basic functionality may be augmented by application developers or platform licensees.

This chapter has identified operating system features related to internationalisation. A metric for evaluating the implementation level of said features was also developed, to be used in Chapter 10.

8. Handheld operating systems

During the 1990s several operating systems or platforms for handheld devices have been developed. This chapter presents four such systems in detail. They were chosen because they are seen as being the market leaders or having the greatest potential for becoming such.

The systems chosen for this study are:

- Microsoft Windows CE
- Symbian OS
- PalmOS
- Java 2 Platform, Micro Edition (in two variants)

There are currently several additional handheld operating systems, but they are proprietary, being phased out, or both.

The selected systems are all open platforms, meaning that practically anyone can develop devices that utilise them, although licensing fees usually apply. Developing applications for these platforms is also unrestricted. In most cases the manufacturers of the systems themselves provide the tools for application development.

A common feature of handheld operating systems is design for low memory requirements and small screens. Some systems also have telephony characteristics, but none of the selected ones make allowances for real-time computing, with the exception of Windows CE 3.0.

Since the focus of this work is on the internationalisation features of the operating systems, the following descriptions are rather brief and do not describe all the features of the selected operating systems and platforms in detail. Additional information can be found in the appropriate manufacturers' product literature, which is available on the World Wide Web. Table 11 lists the Internet addresses of the manufacturers' WWW sites.

Operating system / platform	**WWW site address**
Microsoft Windows CE	http://www.microsoft.com/windows/embedded/ce/default.asp
Symbian OS	http://www.symbian.com
PalmOS	http://www.palmos.com
Java 2 Platform, Micro Edition	http://java.sun.com/j2me/

Table 11. Selected handheld OS manufacturer Web site addresses.

8.1 Microsoft Windows CE

Microsoft Windows CE is an operating system by Microsoft for personal digital assistants (PDA) and handheld personal computers (HPC). Jeff Baker [1997] chronicles the development of Windows CE. It was born from the combination of two different handheld device projects at Microsoft. Initially a lightweight operating system kernel subset of the full Win32, codenamed Pegasus, was developed. In 1996 it adopted the user interface of Windows 95, and first prototype devices were developed by hardware manufacturers such as Hewlett-Packard, Compaq, and Casio.

Since 1996, Windows CE has progressed through major versions 1.01, 2.00, and 2.12 to version 3.0, which was released in the year 2000. Recently Microsoft has also refocused Windows CE to better serve market needs. Windows CE devices have been divided into three mobile device categories, as detailed in Table 12.

Category name	**Purpose**
Pocket PC	Devices with quarter-VGA screen, no keyboard, possibly pen input.
Handheld PC	Half or full VGA screen, possibly keyboard and/or pen input.
Mobile phones	PDA and mobile telephone combined as a "smart phone".

Table 12. Categories of Windows CE mobile devices according to Microsoft.

Several manufacturers have Windows CE devices available. Examples of these include the HP Jornada series in both the Pocket PC and Handheld PC categories, Compaq iPAQ (PocketPC), and the NEC MobilePro series (Handheld PC). In the mobile phone category, Microsoft is developing a Windows CE variant codenamed Stinger. So far Sendo, Mitsubishi and Samsung have announced Stinger-based devices.

Previously Windows CE applications were developed using a specific version of Microsoft Visual C++. Recently Microsoft has introduced standalone development tools that are optimised for handheld programming. Microsoft Windows CE Platform Builder and Microsoft eMbedded [sic] Visual Tools 3.0 require Windows 2000 or Windows NT.

8.2 Symbian OS

In the mid-1990s Psion, the British maker of handheld computing devices, designed a new 32-bit operating system that superceded the company’s earlier

16-bit SIBO system. The new object-oriented operating system was given the name EPOC, with a clear connotation to the English word “epoch” [Pountain, 1997]. Martin Tasker [2000] chronicles the early history of Psion and EPOC.

Psion spun off its software division in 1996 as Psion Software with the intention of licensing EPOC to other hardware vendors. In 1998 it was transformed into a joint venture between Psion, Nokia, and Ericsson and renamed as Symbian. Later also Motorola and Matsushita joined the alliance. These companies own Symbian, which licenses the operating system to them on a royalty basis.

The first major release of EPOC by Symbian was EPOC Release 5 in 1999. It is the operating system of Psion’s current range of handheld devices (Series 5mx, netBook/Series 7, Revo) and the Ericsson MC218, which is manufactured by Psion and corresponds to the Series 5mx. An interim version of Release 5 was also used in the Ericsson R380.

The development after Release 5 has continued in a manner that is very similar to the evolution of Windows CE. EPOC has been renamed to Symbian OS, and the platform has been divided into Generic Technology and so-called Device Family Reference Designs or DFRDs, which are outlined in Table 13. They have been developed by Symbian in co-operation with its owner-cum-licensees.

Reference design	**Purpose**
Crystal	Communicator-type devices equipped with a keyboard and half-VGA screen.
Quartz	Tablet-type, pen-operated devices with quarter-VGA screen, no keyboard.
Pearl	Smartphones with PDA functionality beyond basic telephony

Table 13. Symbian Device Family Reference Designs (DFRDs).

Version 6.0 of Symbian OS is at the heart of the Nokia 9210 Communicator (shown in Figure 4), which is based on the Crystal DFRD. The 9210 is the first Nokia device to use the Symbian OS, which replaces the GEOS operating system used in the 9000 and 9100 series of Communicator.

Figure 4. The Nokia 9210 Communicator uses the Symbian OS 6.0.

Symbian has produced software development kits for the Crystal and Quartz DFRDs running on Symbian OS 6.0. Device makers may provide their own products with SDKs based on these, like Nokia has done for the 9210 Communicator. The platform is open for third-party developers to create and deploy applications, but development requires Microsoft Visual C++. Metrowerks agreed in 2001 to produce a Symbian-enabled version of their CodeWarrior development environment.

8.3 PalmOS

The story of Palm handhelds and PalmOS began with a handwriting recognition software package called Graffiti. As indicated by Rhodes and McKeehan [1999, 4], the makers of Graffiti, a company called Palm Computing, decided to create their own handheld computing device, the Pilot 1000, in 1996. The Pilot 1000 and its successor, the Pilot 5000, quickly became instant successes. The company was acquired by U.S. Robotics and later by 3Com. In the year 2000, 3Com spun Palm off as a separate division.

During the years 1998-2001 Palm Computing has introduced several new handheld devices and became almost the synonym for the pocket-sized PDA. Palm devices have an appealingly simple user interface and ingenious pen input method, but no built-in keyboard. They are designed to work as companion devices to regular desktop PCs and rely on personal computers for backup and the entry of large amounts of data.

The operating system of Palm devices, PalmOS, has become a licensed product. Its licensees include Handspring, Sony, and Kyocera. Recent models of Palm and compatible devices include a colour display and an interface for accessories, which range from a fold-in keyboard to a mobile phone.

Palm devices are functionally simple devices that concentrate on the user interface and experience. The available memory on them is usually severely limited, so there is a strong trend of minimalism in Palm applications. Palm Computing provides a software development kit, but it requires a separate C compiler or an integrated development environment, such as Metrowerks CodeWarrior.

8.4 Java 2 Platform, Micro Edition

Java was created in the early 1990s and released in 1995 by Sun Microsystems. Initially called Oak and designed by Sun researcher James Gosling, it was the programming language for a handheld device called *7 (“StarSeven”). The language was intended as a cross-platform, object-oriented programming language, but it quickly evolved into a platform in its own right. However, the Java platform is more virtual than physical, since endeavours such as JavaOS and microprocessors that use Java byte code as their native instruction set have largely failed to materialise.

Java evolved through versions Java 1.0 and 1.1 to a greatly enhanced version called Java 2. Recently it has been split into variants that more closely match the intended deployment targets.

There are three editions of Java 2 with different foci and contents: Standard Edition (J2SE), Enterprise Edition (J2EE), and Micro Edition (J2ME). This work only deals with J2ME, since it is the version of Java 2 used in handheld device Java implementations. Java 1.1 has also spawned a scaled-down version of Java intended for handheld devices, called PersonalJava. However, future Java implementations on handhelds will more likely be based on J2ME rather than PersonalJava, due to the smaller memory footprint of J2ME.

So far Java has been available for handheld devices as an add-on rather than an essential part of the devices and their software. For example, there has been a Java 1.1 implementation available for EPOC Release 5 since 1999, but it is directed for third-party application developers. As of this writing, very few devices are equipped with J2ME. Among the first is the NTT DoCoMo 503i mobile phone, manufactured by Fujitsu and Matsushita (Panasonic). J2ME is an important part of the phone’s firmware, enabling interactive services that use Java code. These services are collectively known is “i-Appli”. Other new mobile phones using J2ME are the Motorola Accompli 008 and Siemens SL45i. All of these phones were announced in the first half of 2001.

J2ME defines two different *configurations*: one for devices with 128-512 kilobytes of memory available for the Java environment and applications, and one for more than 512 kilobytes available. The former is called Connected Limited Device Configuration (CLDC) [Sun 2000a] and is designed for memory and

processor-constrained devices, while the latter is known as Connected Device Configuration (CDC) and intended for devices with more robust resources. *Profiles* are built on top of configurations, and they define which classes are included in a particular API set for a device. Device manufacturers are involved in defining the profiles through the Java Community Process established by Sun.

This chapter has presented the handheld operating systems and platforms that this study focuses on. The next chapter examines the constraints of handheld devices and their effect on the implementation of the internationalisation requirements presented in Chapter 7.

9. Constraints of handheld devices

Handheld devices are significantly different from conventional personal computers, both in hardware and in software. This brings along many challenges in hardware and software design, the solving of which requires relentless innovation.

This study focuses on the software design aspects of handheld devices. After a brief look on the form factor and related features of handheld devices, this chapter presents some of the most important design constraints of handhelds as they apply to software internationalisation.

9.1 Handheld form factor

Conventional desktop computers have a relatively uniform design: a central unit, detachable keyboard and mouse, and a large monitor. Laptop computers bring the design closer to handheld devices with their focus on portability. All the components of a laptop computer are integrated in the same clamshell design.

Handheld device designs can be roughly divided into two categories: those equipped with a keyboard, and those without. There is a significant difference in form factor between these two types, because the keys of the keyboard require a lot of space to be usable. Partly because of this, handheld devices equipped with a keyboard also provide a larger screen, and bring the device closer in form and function to a laptop computer. This type of device is usually called a "communicator" in reference to the Nokia 9110 Communicator and other similar devices, such as the Psion Series 5mx.

The keyboardless handheld devices are small, usually designed to be held entirely in one hand. This gives a rather obvious origin for the term "handheld", but these smaller devices are often also referred to as palm-sized, in reference to the range of devices offered by Palm Computing. These devices have a significantly smaller display than communicators.

9.2 Storage limitations

In recent years the cost of memory and mass storage has decreased significantly. The typical amount of RAM (Random Access Memory) in a personal computer is from 64 to 256 megabytes, and the sizes of hard disks range from 10 to 30 gigabytes. However, the size and cost of these storage components are still prohibitive for handhelds, which are often mass-market devices, low retail price being an important competitive advantage.

While the amount of memory found in handheld devices is steadily increasing, typical handheld devices still come with only 2-8 megabytes of

RAM, although many high-end devices boast as much as 16-64 megabytes. Hard disks are usually only found in desktop and laptop computers, but the lack of magnetic storage is somewhat compensated with removable media, such as CompactFlash and MMC memory cards, which provide 16-64 megabytes of additional storage.

Another important storage characteristic for handheld devices is the amount of Read-Only Memory (ROM). Parts of the operating system are typically stored on ROM, so that it is immediately available when the device is turned on. Because handhelds have no hard disks, as many applications as possible are stored in flash memory, which can be easily updated when fixes and additions to the software are required. Handheld devices are usually equipped with 2-16 megabytes of ROM.

The amount of RAM or ROM is very limited when compared to desktop and laptop computers. Therefore software for handheld devices needs to be memory-efficient.

9.3 Processing power

The central processing unit (CPU) of a desktop computer is usually manufactured by Intel or AMD for PC compatibles, or by Motorola for Apple computers. Especially Intel and AMD are engaged in an ongoing battle to provide PC manufacturers with CPUs with increasingly higher clock speeds, typically in the range from 600 to 1200 MHz. Software for personal computers has traditionally not taken enough advantage of the increased processing speeds.

Motorola, ARM, Hitachi, NEC, and Intel are the most prominent manufacturers of CPUs for handheld devices. Clock speeds for handheld CPUs typically range from 16 to 200 MHz, but comparing the clock speeds of handheld and desktop CPUs is a mismatched comparison, since the processor architectures and the applications are significantly different. Handheld processors do not typically perform extensive amounts of number-crunching and other intensive tasks, although processor-intensive features for multimedia content processing are an emerging trend. The I/O capabilities of handhelds are not as versatile as in desktop computers, and updating the small screen displays does not consume as many processor cycles as in PCs where the display hardware rivals the main processor in speed and complexity.

Despite research efforts there have been no commercially successful implementations of microprocessors that use Java bytecode as their native instruction set. These processors are already technically feasible, and they would speed up the execution of Java software significantly. The investment in existing C and assembly language software is still much too large to be replaced and rewritten,

at least partly, in Java. However, manufacturers such as ARM and Sun Microsystems offer accelerators to speed up the execution of Java programs.

Power consumption is a significant factor in the hardware design of handhelds. The devices must run for several days on the main batteries, and the information in volatile memory needs to be maintained for some time even when the main batteries are removed. The electronics design of a handheld device must minimise the consumption of power when the device is idle as well as when it is active.

9.4 Display technology

The screen resolution of a handheld device is typically very low compared to the monitor of a desktop PC. For example, the screen size on the Palm handhelds is 160 x 160 pixels, while Psion Series 5mx has a 640 x 240 pixel screen, as does the Nokia 9210 Communicator. Many devices based on the Pocket PC version of Microsoft Windows CE have 240 x 320 screens, as will future smartphones based on the Symbian Quartz design.

Until recently, screen displays of handheld devices have been monochrome or capable of at most 16 shades of grey. In early 2000 Palm introduced its first colour handheld device, the Palm IIIc. At least Compaq and Hewlett-Packard now have handheld models with colour screens, which is also a feature of the Nokia 9210 Communicator. The colour resolution of these screens ranges from 8-bit to 12-bit, or from 256 to 4096 simultaneous colours.

The increased screen resolution has brought handheld devices closer to their desktop counterparts. More information can be fit on the screen without resorting to abbreviations and often cryptic icons, and larger images and other content can be viewed and manipulated. The introduction of colour has been an important step towards a friendlier user interface, giving users more options to customise the appearance of the handheld applications. Spencer et al. [2001, 81] note that the use of colour also seems to improve the perceived resolution of the screen.

Despite increased resolution and the use of colour, handheld screens are still too small for most productivity and content creation applications. The low resolution also degrades the readability of the text on the display, especially in non-alphabetic languages, which often require a generous amount of both vertical and horizontal space for the text to be legible.

9.5 Input methods

The presence or absence of a keyboard largely dictates the applications that are feasible for a particular type of handheld device. Without a keyboard the user is not going to enter large amounts of data into the device, which is exactly the

design philosophy of the Palm handhelds. Massive data entry happens on the desktop computer. Small additional amounts of data are recorded with the handheld, and the data is occasionally resynchronised with the desktop. It is possible to enter large amounts of data using a stylus and handwriting recognition, but it requires patience and determination.

In order to fit a working keyboard into a handheld device, the manufacturers have had to shrink the keys down to the level where users with average to large fingertips have difficulty in hitting the right keys. Touch typing is usually difficult even on laptop computers, but the keys on handheld keyboards are typically only one fourth of the laptop keys. The number of special keys also needs to be limited; for example, the Nokia 9210 Communicator has only one Shift and Control key instead of two of each, as is typical in laptops, and the separate cursor movement keys for each direction have been replaced with a single gamepad-type rocker key.

The requirements of text input in languages using the Latin script are usually well understood. As detailed in section 6.2.3, handwriting recognition (HWR) is one viable option for complex scripts, and interest in it is high especially in China [Spencer et al., 2001]. However, HWR is still slow and somewhat awkward, and requires the user to adapt to the software and not the other way around.

Another method of text input is the use of a virtual keyboard, which is used with a stylus. The software draws an image of a keyboard on top of the user screen, and the user taps the image of a key to select it. This method is quite efficient especially with special characters and punctuation, but it is somewhat unwieldy for entering large amounts of text.

9.6 Impact of constraints on internationalisation

While features required by internationalised software are highly desirable from a usability point of view, on handheld devices the implementation poses problems because of the often rather limited resources outlined above. The biggest challenge of handheld device manufacturers and software suppliers is to implement as many desirable internationalisation features as possible while wasting as little memory, processing power, and screen real estate as possible.

This section analyses the constraints of handheld devices in terms of internationalisation. The set of internationalisation requirements outlined in Chapter 6 provide a framework for the discussion.

9.6.1 Locale model

As detailed in Chapter 3, the locale model of an operating system consists of the locale API (often known as National Language Support or NLS) and the associ-

ated locale database containing national and regional conventions, data related to text processing, and other relevant information. The memory footprint of this data varies according to the locale model implementation.

In desktop and mainframe operating systems, the lack of storage space is usually not a concern. For example, in the English language version of Microsoft Windows NT 4.0, which has a limited selection of Western European locales, the locale database occupies approximately two megabytes of hard disk space. (The memory required by the code that implements the locale API is unknown.)

In the Symbian platform, version 6.0, the locale data for nine reference locales occupies a little under 400 kilobytes, but the size of the data for different locales varies from 33 to 105 kilobytes. By comparison, the international edition of the Java 2 Runtime Environment (JRE) version 1.3 contains a single Java archive with resource data for over 100 locales, with a total size of 2.6 megabytes.

In a handheld device it is obviously not desirable to keep the data for all the different locales present in the device, because they take up a large amount of storage space. The user probably needs only one or at most two locales two select from. Parts of the hardware design, such as the keyboard, also need to be customised for different locales.

9.6.2 Character encodings

Internet connectivity for electronic mail and the World Wide Web is becoming increasingly important in handheld devices. The Internet Mail Consortium report MAIL-I18N [IMC, 1998] lists as problems in Internet mail the inability for users to compose and view messages using the correct character set, and the lack of language and control information in messages. IMC recommends that all mail-creating and mail-displaying programs should handle UTF-8 and a wide variety of common character sets.

This leads to the conclusion that mobile handheld devices providing Internet e-mail capabilities to users should support multiple character encodings in both sending and receiving mail. By extension the same applies to HTML. Other applications besides e-mail will also need transcoding capabilities, so the underlying operating system should support different encodings.

Supporting different character encodings is not a trivial undertaking, and is often hampered by a lack of understanding of the issues involved. The conversions between coded character sets can often be achieved using mapping tables, but the size of these tables varies with the character sets involved. It is not uncommon to find that some encodings for ideographic scripts require map-

ping tables of several hundred kilobytes in size. Therefore support for a large number of character sets may require many megabytes of storage, which is a prohibitive amount for handhelds.

With careful consideration it should be possible to determine which character encodings are essential in a given locale. However, it is feasible to try and support as many encodings as possible, because users have increasingly international interactions which involve text encoded unpredictably and differently from the "native" encoding of the user and the device. With the use of Unicode all incoming data can be forwarded, if not viewed or manipulated.

9.6.3 Application resources

When the user interface texts and other localisable content are separated from the program logic, they can be easily replaced with other content at runtime. The localised content needs to be labelled correctly for each locale, and the software must be able to obtain the current locale from the operating system to load the correct content. These mechanisms usually involve only a few API function calls.

An example of resource management can be found in Microsoft Windows CE. Since tens of thousands of developers are at least somewhat familiar with the Windows development model of separating localisable content into resource files, they can immediately carry on with the same practice in Windows CE, since the same model is used in handheld development [Hall, 2000]. Similarly, according to Deitsch and Czarnecki [2001], Java 2 uses resource bundles in all editions, although this is not exactly accurate in J2ME (see section 10.4).

The difficulties in handheld development do not stem so much from the mechanisms used to identify and load localisable content, but the format of the content itself and the extent to which the programmers take advantage of these mechanisms. An example follows: a single text string such as "Account:" should never be reused across different parts of an application, because there may be several different translations of the English word in different contexts. If the application programmer attempts to save memory by defining a single string and then reusing it in several places, it causes localisation problems that are next to impossible to solve and usually involve re-engineering. The temptation to reuse strings is great, especially with limited available memory, but the savings should be carefully balanced with potential localisation problems.

Another example is string concatenation, which is often used in constructing error messages that involve variable components such as file names or line numbers. The traditional method is to build the string image at runtime from

its constituent parts, namely static text and dynamic file name. Because of linguistic differences, this will not usually work across languages. Fortunately, practically all programming languages or platforms now support variable components in messages, which also avoids the creation of many temporary strings[6].

9.6.4 Text rendering and fonts

If a handheld device offers the selection of several fonts to the user, the technology used to render these fonts has a significant impact on the output quality and memory usage of the device. As stated in section 7.4, bitmap fonts usually consume much more memory than outline fonts, because the former are made of pixel matrices. In contrast, outline fonts consist of rendering instructions and font metrics, which are algorithmically processed.

Even at the relatively low processor speeds of handheld devices, the processing of outline fonts is feasible. The output quality obtained is much better than that of bitmap fonts, and the savings in memory may compensate for the performance hit.

Outline fonts are typographically much more sophisticated, especially when the text being rendered is written in a complex script, such as Arabic, Hebrew, Thai or Hindi. The correct rendering of nearly all of these scripts requires the use of ligatures, which is difficult to achieve using bitmap fonts. The readability of ideographic scripts is also improved by the increased accuracy.

The problem with all fonts regardless of technology is still the size of the font information. Because handheld devices often support Unicode and several scripts, it remains a subject of debate whether the fonts in a device should contain glyphs for other scripts than the device's native one. Fonts with glyphs for all characters defined in Unicode, such as Arial Unicode MS found in Microsoft Office 2000, are in the size range of 20 megabytes, which is obviously too large for a handheld device. The problem is usually solved by equipping the font(s) with only the glyphs in some specific range(s) of Unicode characters. If a character with no glyph is encountered, a predefined "unknown character" glyph is displayed, but the original content is retained.

---

[6] In handheld operating systems stack space is often limited, calling for coding practices that minimise the use of temporary variables.

9.6.5 Text processing

From an internationalisation point of view the single most important decision in the design of an operating system, be it for a handheld or desktop computing device, is to support Unicode. The Unicode standard [The Unicode Consortium, 2000] is the most practical, if not only, solution to a universal problem of handling text in many scripts and languages. However, the use of Unicode brings along a couple of problems that are exacerbated in handheld computers.

Firstly, the amount of storage required for text is effectively doubled when compared to traditional character encoding methods. What used to be 8-bit entities are now 16-bit, and the expansion of Unicode brings along even 32-bit entities. To alleviate this problem, the Unicode Consortium has come up with the Standard Compression Scheme for Unicode or SCSU [Wolf et al., 2000]. It is based on the characteristics of Unicode-encoded text, and is efficient also for short strings, unlike Huffman encoding and LZW. Graham Asher [2001, 9] argues that Unicode text does not really need to be compressed, but states that to reassure developers and licensees, SCSU has been implemented on a very low level of the I/O stream component of the Symbian platform.

Secondly, in order to meaningfully process Unicode-encoded text, the underlying system needs to know several things about the characters that constitute the text. It is vital for parsers and text formatters to know whether a character is a number, a letter or a punctuation character. Given the more than 40,000 characters in Unicode, with more to come, this information set is rather large. The designers of Unicode have collected character property data into the Unicode Character Database, a delimited text file of 763 kilobytes in version 3.1. By ingenious data manipulation and the use of the trie data structure this amount can be reduced to an impressive 18 kilobytes [Asher 2001, 6].

String collation is a highly locale-sensitive text processing task. The Unicode Collation Algorithm or UCA [Davis and Whistler, 2001] provides an unambiguous ordering for all Unicode characters, but it needs to be further tailored to a particular locale. The algorithm works by reducing strings to sort keys and comparing them on three levels. An easy method of implementing collation is to implement UCA and to add the locale-specific elements.

Because most of the required collation data is common to all locales, most of the locale-specific implementations do not cause significant overhead. Ideographic scripts are yet again more problematic because of the large number of characters involved and the absence of traditional alphabetical ordering. As with other locale data, collation data for all locales are not feasible to include in the same version.

9.6.6 Input methods

The main input methods – keyboard entry, handwriting recognition, and virtual keyboard – all present internationalisation challenges. The keyboard layouts, whether physical or virtual, need to be localised for different language versions. In the case of physical keyboards there are also hardware manufacturing considerations, since different keycaps need to be printed for most locales.

With keyboard input there is also the choice of transliterated or structural input, as stated in section 7.6. These methods are mainly used in CJKV, and they usually require a sizable conversion dictionary. There are literally dozens of software implementations of CJKV input methods for desktop and handheld computers. The memory limitations of handhelds force the input method designer to limit the size of the conversion dictionary.

The appearance of the input method is constrained by the display of the handheld device. There are four basic styles of input methods, originating from the X Window System Input Method specification (XIM):

- on-the-spot (composed text is rendered inside the text window)
- over-the-spot (composed text is rendered over the insertion point in a layer over the document window)
- off-the-spot (text is composed in an area attached to the text window)
- root-window (text is composed in an entirely separate window)

Any of these styles can be used in a handheld device, but the selection of the style is influenced by the size of the screen display and the general user interface design of the device.

This chapter has presented the constraints found in handheld devices and analysed their impact on the implementation of internationalisation features. In the next chapter the support for internationalisation is analysed in terms of the metrics developed in section 7.7.

10. Internationalisation metrics in selected operating systems

The handheld operating systems discussed in this study, presented in Chapter 8, all have some degree of support for internationalisation. The aim of this chapter is to evaluate the exact amount of that support using the metrics defined in Chapter 7.

The internationalisation features of each operating system are described and presented in a consistent tabular format, and a score for each system is calculated based on the values defined in section 7.7. The information concerning internationalisation applies to the latest shipping versions of the operating systems. For general details about the systems, refer to Chapter 8.

10.1 Microsoft Windows CE 3.0

Microsoft Windows CE is one of the members of Microsoft's Win32 platform family. It inherits much of its internationalisation features from its desktop siblings, namely Windows 95/98/Me and Windows NT/2000. The kernels of each of these operating systems are significantly different, but Microsoft has managed to largely preserve the core Win32 API across them, along with the NLS API.

A significant feature of Windows CE is its full support for Unicode as the system's native character encoding. Windows CE also supports other character encodings and conversions between them and Unicode through the *MultiByteToWideChar* and *WideCharToMultiByte* API functions. The number of character sets (*code pages* in Microsoft terminology) is not extensive, however; Hall [1999, 35] notes that a call to *EnumSystemCodePages* returns only three code pages. This set of code pages may also be different among the Windows CE language family. According to Microsoft [2000], there are five different language groups: U.S. English, Western European, Eastern European, Far Eastern, and bi-directional and other complex script languages.

The locale model of Windows CE is identical to the desktop Windows versions, described in section 4.5. Locales are identified using an LCID, but the default user locale is always the same as the system locale. The locale database stores data about cultural conventions, used by several NLS API functions for formatting dates, times, and currencies and so on. The data can also be queried using the *GetLocaleInfo* API function.

Windows CE supports the separate compilation of application resources for easy localisation. Resources are compiled to a processor-independent binary intermediate file and are bound to the final executable file or to a separate resource-only dynamic link library (DLL). Resources are accessed at runtime

using the *LoadResource* API function. Dialog controls are specified in resource files using dialog units that are translated into physical pixels at runtime.

The Graphics Device Interface (GDI) of Windows CE has supported TrueType fonts since version 2.0 of the OS. For input, Windows CE devices use different methods depending on the form factor of the device. PocketPC devices have no keyboard, unlike Handheld PC devices, so they use handwriting recognition or a virtual pop-up keyboard to input characters. The functionality of the Windows CE handwriting recognition engine is exposed to applications through the Hwx API.

Internationalisation support in Windows CE is quite extensive. Windows CE 3.0 is still a relatively new release, so there is no definite information yet about internationalisation developments in future versions. The platform already supports complex scripts and other traditionally "hard" internationalisation features.

10.2 Symbian OS 6.0

At the heart of the Symbian platform is the EPOC operating system, which is specifically designed for handheld devices with limited memory and processing power. In its earliest forms EPOC was severely limited in terms of internationalisation, but the development work initiated some time before version 5.0 and described by Tasker et al. [2000] led to a new version 6.0 with full support for Unicode.

Unicode has been the native character encoding of Symbian OS since version 6.0. The Unicode implementation is described in detail by Graham Asher [2001]. The memory usage hit caused by the move to Unicode has been largely cancelled by the increased amount of memory in the devices using the Symbian platform and the use of data compression at a low level. The platform also supports transcoding between Unicode and other character sets using an extensible character converter API called CharConv.

The Symbian locale model reflects the heritage of versions of EPOC up to 5.0. The market for Psion handhelds has traditionally consisted largely of the Western European markets, and initially there was no concept of locale. The international software variants were distinguished by a numeric language identifier, with symbolic names such as *ELangGerman* or *ELangSpanish*. Before version 6.0, software versions for other than Western European languages using the Latin script have been the responsibility of third-party companies crafting patches to the operating system. Version 6.0 supports the Greek and Cyrillic scripts as well as Chinese and Japanese.

As the market potential of the Symbian platform has increased, locales and associated cultural conventions have become a part of the platform. The

concept of locale is heavily biased on language, with variants for different countries retrofit to the original language identifier scheme. Data stored in a system global *TLocale* class can be queried through public member functions of the class.

Application resources can be detached from program logic using separate resource script files, which are compiled to a binary form using a resource compiler. The appropriate resources are loaded at runtime using member functions of resource-handling classes such as *RResourceFile* and *TResourceReader*. Dialog controls are arranged automatically in a one-dimensional layout, and automatically resized when possible, with no need for specification with pixels or dialog units.

Non-Unicode versions of EPOC supported only bitmap fonts. The Unicode-enabled versions use a system that allows the use of multiple outline font formats by plugging in a new font rasterizer. The most important font format is TrueType.

As with Windows CE, the input methods for Symbian platform devices vary according to form factor. The Crystal Device Family Reference Design (DFRD) uses a keyboard for input, while the Quartz DFRD uses a virtual keyboard or handwriting recognition. The platform provides a framework for licensees and third-party developers to add front-end processors (FEPs), which may use all the features of the GUI to provide different styles of input methods to end users.

Currently the Symbian platform does not support complex scripts such as Thai, Devanagari, and bi-directional scripts. The locale model also needs to separate language and region more clearly. Symbian's Unicode support is extensive, and version 6.0 contains several other important internationalisation features, such as the character set conversion framework.

10.3 PalmOS 3.5

Palm is the current market leader in the handheld device marketplace, but the user base of Palm devices consists largely of English speakers. As Palm devices have spread further than North America, the limited support in PalmOS for other languages than English has caused severe problems in localising the devices.

Palm provides only a few localised versions of the software, namely the EFIGS variants (for the initial letters of English, French, Italian, German, and Spanish). Other language variants, such as Japanese and Arabic, have been the responsibility of industrious third-party development companies, and implemented as patches to the operating system [Ó Broin, 2000].

The concept of locale in PalmOS is rather rudimentary. In version 3.0 the set of locales is restricted to those that use the Latin character encoding. PalmOS 3.1 added support for Shift-JIS encoding and the Japanese locale, and only in version 3.5 it was made possible to select resources according to the system locale of the device. It is therefore fair to say that PalmOS has not been internationalised, partly because Palm did not anticipate the success of the device outside North America.

PalmOS has some information about cultural conventions available through public API functions, but there is additional private information available only through non-standard methods that will fail when the internal locale information changes.

There is no support for Unicode in PalmOS. The native encoding of the device is an 8-bit single-byte or multi-byte character set based on the system locale, Latin-1 for Western locales and Shift-JIS for Japanese. PalmOS has a small selection of built-in bitmap fonts with the possibility to use additional fonts as application resources.

Palm devices use either a virtual pop-up keyboard or handwriting recognition for input. The Palm's handwriting method is called "Graffiti", and it supports essentially ASCII characters with special additional strokes for other Latin characters. The virtual keyboard contains several international symbols. In future PalmOS versions the Graffiti system will be enhanced to support Japanese characters.

Resources used by Palm applications are compiled into a separate application-specific resource database. In essence, each application is a resource database, with one resource containing application code, another the application's icon, and others storing the forms, menus, and strings [Rhodes and McKeehan 1999, 91].

Palm is under pressure to develop PalmOS to support more international locales, but is going to have hard time making sure that the new features do not break the very large installed base of third party applications for the Palm devices. PalmOS 4.0 is reported to have extended support for locales and transcoding between Unicode and other character sets, but will still be left far behind other handheld systems in terms of internationalisation.

10.4 Java 2 Platform, Micro Edition

Java 2 Micro Edition (J2ME) consists of two configurations targeted on different kinds of handheld devices, as detailed in section 8.4. So far, industry groups have, in cooperation with Sun, defined one profile for each configuration. The Mobile Information Device Profile or MIDP is built on the Connected, Limited Device Configuration (CLDC) and intended for mobile phones and two-way

pagers. The Foundation profile is built on the Connected Device Configuration (CDC), and is targeted at consumer electronics and embedded devices.

The use of Java in mobile devices is geared towards "smart" mobile handsets, the focus is shifting away from the CDC towards CLDC and MIDP. However, neither of these two configurations is currently usable as a standalone operating system. Therefore this study includes both the CDC and the CLDC as application platforms. Some features of the configurations may be implemented using existing native support from the underlying operating system.

Both J2ME configurations inherit the use of Unicode as the platform's native character encoding from the more extensive Java 2 editions, Standard Edition (J2SE) and Enterprise Edition (J2EE). However, other support for internationalisation is very limited in both configurations. The only requirements are support for the U.S. English locale (identifier en_US) and the ISO 8859-1 character encoding. It is the reponsibility of the profiles to implement additional internationalisation features.

JavaSoft has deprecated many parts of the Java API, especially the internationalisation parts, between versions 1.0 and 1.1. Application developers have been clearly instructed not to use deprecated parts of the API, because they will not be supported indefinitely. In J2ME configurations JavaSoft no longer supports these deprecated APIs, so applications that use them may require a rewrite.

10.4.1 CLDC and MIDP

Compared to J2SE, internationalisation support in the CLDC is severely limited. For example, the *java.util.Locale* class is not available, and according to Eric Giguere [2001] "the CLDC provides no support for any formatting of strings, numbers, currencies, or other locale-specific operations". The exact support for internationalisation ultimately depends on the profile definition, but the MIDP definition does not add any internationalisation features. Even resource bundles need to be simulated, because the *java.util.ResourceBundle* class is not available. However, MIDP supports transcoding between Unicode and other character encodings at the discretion of the MIDP implementer.

The CLDC does not define a graphical user interface; it is left to the profile definitions, currently the MIDP in practice. The Abstract Window Toolkit is not supported because it was originally designed for desktop computers [Sun 2000b, 49]. The user interface of a MIDP device is constructed out of a limited set of primitives. The MIDP specification does not state whether these primitives support any of the internationalisation features of J2SE, such as bi-directional text and ligatures.

10.4.2 CDC and the Foundation profile

The scope of the CDC is wider both in general and in internationalisation. It includes a significant portion of the internationalisation API found in J2SE, so it is easier for application programmers to port existing internationalised Java applications to CDC. The locale model is identical to J2SE, meaning that the *java.util.Locale* class is available, as are most of the classes in the *java.text* package of J2SE that are used to format data according to locale-specific cultural conventions. The CDC configuration does not define all relevant internationalisation features, such as the collation of text strings using *java.text.Collator* and related classes, but the Foundation profile adds them.

The state of internationalisation in J2SE and J2EE is excellent, but the J2ME configurations and profiles are a step backwards in this respect. Especially CLDC devices are in danger of being left with no significant support for internationalisation, unless device manufacturers actively provide internationalisation features beyond the MIDP specification or decide to revise the profile. This sort of revision is in progress under the working title of MIDP-NG [Uotila, 2001], but the improvements to internationalisation support are not known at the time of this writing.

10.5 Roundup of internationalisation features

The preceding narrative descriptions give an overview of the internationalisation support in each handheld platform. The table in Appendix 1 collects the essence of these descriptions into an easily comparable format. Table 14 contains the final scores for the platforms.

Handheld platform	**Internationalisation score**
Microsoft Windows CE 3.0	**31**
Symbian OS 6.0	**30**
PalmOS 3.5	**10**
J2ME CDC + Foundation	**24**
J2ME CLDC + MIDP	**11**

Table 14. Internationalisation scores for the studied handheld platforms.

These results indicate that in terms of these metrics, the internationalisation status of Windows CE 3.0 and Symbian OS 6.0 are almost the same. How-

ever, the metric is somewhat biased towards the completeness of the Unicode implementation. Therefore it is important to note that the support for complex scripts and different input methods are better in Windows CE than in Symbian OS.

PalmOS 3.5 and J2ME/CLDC/MIDP are the clear losers in this comparison. PalmOS has not been designed for internationalisation, and MIDP has been purposefully stripped of most internationalisation functionality in the interest of low memory consumption.

J2ME/CDC/Foundation strikes a balance between an acceptable memory footprint and internationalisation support. Because it is not a standalone operating system, the memory requirements may still be excessive for handheld use.

11. Summary

The users of computing devices and applications normally expect the user interfaces and functionality of them to conform to the users' cultural expectations. This is especially important with handheld devices because of their personal nature.

The amount of support for internationalisation in an operating system or platform can be measured by identifying the internationalisation requirements of an operating system and examining how well it meets them. Handheld operating systems present additional constraints due to their unique hardware and software design.

In this study I have selected and presented the most viable or popular handheld operating systems in use during the years 2000-2001 and evaluated their internationalisation support with a set of metrics derived from the internationalisation requirements. I have also examined the impact of the handheld design constraints on the implementation of internationalisation.

11.1 Results of the study

The metrics compiled for the handheld operating systems are based on the identification of the most common internationalisation features expected of an operating system. Each feature was given an associated point value, as detailed in section 7.7.

The point values were assigned separately for each handheld operating system. The total score for each system is established by adding up the points awarded for each internationalisation feature in the table.

The final results in Table 14 indicate a near draw between Windows CE and Symbian OS, but the metric is somewhat biased towards the Unicode technical implementation of the platform. For practical localisation and product creation Windows CE reaches a broader audience because it has built-in support for several complex scripts, unlike Symbian OS.

The narrow scope of PalmOS with regard to internationalisation shows well in its final score. PalmOS 3.5 has no concept of Unicode whatsoever, and falls behind also in the support for different character encodings.

The two configurations of Java 2 Micro Edition fared rather differently. The CDC has significantly more internationalisation features inherited from "desktop Java" than its CLDC counterpart. CDC is not far behind Windows CE and Symbian OS, whereas CLDC scored almost as badly as PalmOS, despite its built-in support for Unicode. It is important to note, however, that the actual implementations of CDC and CLDC may choose to add significant internation-

alisation support, for example in the form of character encodings or input methods.

11.2 Further research topics

The subject of this thesis is a moving target in that the mobile computing industry produces new appliances at a rapid pace. For example, during the writing of this thesis Palm has already shipped PalmOS 4.0 with improved internationalisation support. Several new Java-enabled mobile phones have been announced, and Microsoft has completed a beta version of PocketPC 2002. Symbian has announced version 6.1 of their OS, but it will not have significant improvements in internationalisation.

Another emerging trend that has been overlooked in this research work is the use of Embedded Linux in handheld devices. Linux has gained enormous popularity in the desktop in recent years, and is rapidly gaining momentum in the handheld industry as well. Several handheld devices using Linux have indeed been announced. Their internationalisation support could fall in the realm of the POSIX model, but because of the strict memory requirements of handheld devices the level of internationalisation might be affected in a manner similar to Java 2 Micro Edition and MIDP. The use of Linux in handhelds would definitely be an interesting topic for future research.

References

[Asher, 2001] Graham Asher, Unicode in your pocket. In: *Proceedings of the 18th International Unicode Conference* (2001), A6.

[Baker, 1997] Jeff Baker, *Windows CE Application Programming*. Macmillan Technical Publishing, 1997.

[Burden and Slawsby, 2000] Kevin Burden and Alex Slawsby, *The Battle at Hand: The Smart Handheld Devices Market Forecast and Analysis, 2000-2004*. IDC, 2000.

[Campione et al., 1998] Mary Campione, Kathy Walrath, Alison Huml, Tutorial Team, *The Java Tutorial Continued – The Rest of the JDK*. Addison-Wesley, 1998.

[Davis, 2000] Mark Davis, *Unicode Standard Annex #9: The Bidirectional Algorithm*. The Unicode Consortium, 2000. http://www.unicode.org/unicode/reports/tr9/ (last accessed 2001-03-19).

[Davis and Whistler, 2001] Mark Davis and Ken Whistler, *Unicode Technical Standard #10: Unicode Collation Algorithm, version 8.0, 2001-03-23*. The Unicode Consortium. http://www.unicode.org/unicode/reports/tr10/tr10-8.html (last accessed 2001-05-23).

[Day, 2000] Donald Day, Gauging the extent of internationalization activities. In: *Designing for Global Markets 2: Proceedings of the Second International Workshop On Internationalisation of Products and Systems*, 125-136. Backhouse Press, 2000.

[Deitch and Czarnecki, 2001] Andy Deitch and David Czarnecki, *Java Internationalization*. O'Reilly & Associates, 2001.

[del Galdo and Nielsen, 1996] Elisa M. del Galdo and Jakob Nielsen (ed.), *International User Interfaces*. John Wiley & Sons, 1996.

[Delio, 2000] Michelle Delio, Handheld battle heating up. *Wired News*, 27 December 2000. http://www.wired.com/news/gizmos/0,1452,40856,00.html (last accessed 2001-01-23)

[Flanagan, 1999] David Flanagan, *Java In a Nutshell, 3rd Edition*. O'Reilly & Associates, 1999.

[Giguere, 2001] Eric Giguere, Writing world-aware J2ME applications. In: *J2ME Tech Tips, January 29, 2001*. Java Developer Connection (JavaSoft), 2001. http://developer.java.sun.com/developer/J2METechTips/2001/tt0129.html#tip1 (last accessed 2001-05-29).

[Graham, 2000] Tony Graham, *Unicode: A Primer*. M&T Books, 2000.

[Hall, 2000] Bill Hall, Internationalizing Windows CE. *Multilingual* **30 (**March 2000**),** 33-37.

[IBM, 2001] IBM Corporation, International Components for Unicode. Available at http://oss.software.ibm.com/icu/ (last accessed 2001-12-05).

[IMC, 1998] Internet Mail Consortium, *Using International Characters in Internet Mail*. Internet Mail Consortium Report MAIL-I18N (IMCR-010). Available at http://www.imc.org/mail-i18n.html (last accessed 2001-05-15).

[ISO, 1988a] *ISO 639:1988 Code for the representation of names of languages*. International Organization for Standardization, 1988.

[ISO, 1988b] *ISO 8601:1988 Data elements and interchange formats – Information interchange -- Representation of dates and times.* International Organization for Standardization, 1988.

[ISO, 1993] *ISO/IEC 9945-2:1993 Information technology -- Portable Operating System Interface (POSIX) -- Part 2: Shell and Utilities*. International Organization for Standardization, 1993.

[ISO, 1997] *ISO 3166-1:1997 Codes for the representation of names of countries and their subdivisions -- Part 1: Country codes*. International Organization for Standardization, 1997.

[ISO, 1998] *ISO/IEC 14882:1998 Programming languages -- C++*. International Organization for Standardization, 1998.

[ISO, 1999a] *ISO/IEC WD3 15435 Information Technology – Internationalization APIs*. International Organization for Standardization, 1999.

[ISO, 1999b] *ISO/IEC PDTR 14652 Information Technology – Specification method for cultural conventions*. International Organization for Standardization, 1999.

[ISO, 1999c] *ISO/IEC 8859-15:1999 Information Technology – 8-bit single-byte coded graphic character sets – Part 15: Latin alphabet No. 9*. International Organization for Standardization, 1999.

[Kano, 1995] Nadine Kano, *Developing International Software for Windows 95 and Windows NT*. Microsoft Press, 1995. Available at http://msdn.microsoft.com/library/default.asp?URL=/library/books/devintl/s24ae.htm (last accessed 2001-01-04)

[Koanantakool, 1993] Thaweesak Koanantakool, The Keyboard Layouts and Input Method of the Thai Language. In: *Proceedings of the Symposium on Natural Language Processing in Thailand 1993*. Available at http://www.nectec.or.th/it-standards/keyboard_layout/thai-key.htm (last accessed 2001-03-19).

[Kreft and Langer, 1997] Klaus Kreft and Angelika Langer, Internationalization using Standard C++. *C/C++ User Journal*, Volume 15, Issue 9 (September

1997). Available at http://home.camelot.de/langer/Articles/Internationalization/I18N.htm (last accessed 2001-07-25).

[Küster, 2000] Marc Wilhelm Küster, Developing European ordering rules. *Multilingual* **33** (Volume 11 Issue 5), 33-36.

[Lunde, 1999] Ken Lunde, *CJKV Information Processing*. O'Reilly & Associates, 1999.

[Luong et al, 1995] Tuoc V. Luong, James S.H. Lok, David J. Taylor, Kevin Driscoll, *Internationalization: Developing Software for Global Markets*. John Wiley & Sons, 1995.

[Microsoft, 2000] Microsoft Corporation, *Developing International Applications for Microsoft Windows CE 3.0-based devices*. Microsoft Developer Network, 2000. Available at http://msdn.microsoft.com/library/techart/local.htm (last accessed 2001-02-01).

[Myers, 1998] Nathan Myers, C++ locales. *Dr. Dobb's Journal* **288** (August 1998), 42-45.

[Nielsen, 1990] Jakob Nielsen (ed.), *Designing User Interfaces for International Use*. Elsevier, 1990.

[Ó Broin, 2000) Ultan Ó Broin, Localizing Palm operating system devices. *Multilingual* **30** (March 2000), 38-41.

[O'Donnell, 1994] Sandra Martin O'Donnell, *Programming for the World – A Guide to Internationalization*. PTR Prentice Hall, 1994.

[Pountain, 1997] Dick Pountain, A New Epoch for Hand-Helds. *BYTE*, October 1997, 45-46.

[Pressman, 2001] Roger S. Pressman, *Software Engineering – A Practitioner's Approach, 5th Edition*. McGraw-Hill, 2001.

[Rhodes and McKeehan, 1999] Neil Rhodes and Julie McKeehan, *Palm Programming – The Developer's Guide*. O'Reilly & Associates, 1999.

[Rumbaugh et al, 1991] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[Russo and Boor, 1993] Patricia Russo and Stephen Boor, How fluent is your interface? In: *Proceedings of INTERCHI '93*, 342-347.

[Ruuska, 1999] Satu Ruuska, Mobile communication devices for international use – exploring cultural diversity through contextual inquiry. In: *Proceedings of the first International Workshop on Internationalization of Products and Systems* (1999), 217-226.

[Schmitt, 2000] David A. Schmitt, *International Programming for Microsoft Windows*. Microsoft Press, 2000.

[Spencer et al., 2001] Matt Spencer, Keith Bailey, Bill Morgan, Sally Dent, *Mobile Device Input Methods – Qualitative Behaviour and Attribute Study*. Nokia Mobile Phones internal document.

[Sun, 2000a] Sun Microsystems, *Connected, Limited Device Configuration Specification Version 1.0*. Sun Microsystems, 2000.

[Sun, 2000b] Sun Microsystems, *Mobile Information Device Profile (JSR-37) - JCP Specification – Java 2 Platform, Micro Edition, 1.0a*. Sun Microsystems, 2000.

[Tasker et al., 2000] Martin Tasker, Jonathan Allin, Jonathan Dixon, John Forrest, Mark Heath, Tim Richardson, Mark Shackman, *Professional Symbian Programming*. Wrox Press, 2000.

[Turnbull, 1999] Steve Turnbull, Alphabet soup: the internationalization of Linux, part 1. *Linux Journal* **59** (March 1999), 30-38. Available at http://turnbull.sk.tsukuba.ac.jp/Tools/I18N/LJ-I18N.html (last accessed 2000-12-18).

[Tuthill and Smallberg, 1997] Bill Tuthill and David Smallberg, *Creating Worldwide Software, Second Edition*. Sun Microsystems Press, 1997.

[The Unicode Consortium, 2000] The Unicode Consortium, *The Unicode Standard Version 3.0*. Addison-Wesley, 2000. Available at http://www.unicode.org/unicode/standard/standard.html (last accessed 2000-12-18).

[Uotila, 2001] Aleksi Uotila, e-mail communication, 2001-05-29.

[Wendt, 2000] David Wendt, Multilanguage programming. *Dr. Dobb's Journal* **318** (November 2000), 68-76.

[Winkler, 2000] Arnold F. Winkler, *Personal thoughts about the future of SC22/WG20 – Internationalisation for consideration by the SC22 plenary in Nara*. http://www.pori.tut.fi/~hj/for-guests/public/sc22-plenary-2000/def/n3164.htm (last accessed 2001-02-22).

[Wolf et al., 2000] Misha Wolf, Ken Whistler, Charles Wiksteed, Mark Davis, Asmus Freytag, *Unicode Technical Standard #6: A Standard Compression Scheme for Unicode, version 3.2, 2000-08-31*. The Unicode Consortium. http://www.unicode.org/unicode/reports/tr6.html (last accessed 2001-05-16).

Appendix 1. Internationalisation metrics evaluation sheet

Internationalisation feature	Windows CE 3.0	Symbian OS 6.0	PalmOS 3.5	J2ME/CDC, Foundation	J2ME/CLDC, MIDP
Locale model					
Locale identification	Yes	Yes	No	Yes	Yes
Locale information database	Yes	Yes	No	Yes	No
Locale database queries	Yes	Yes	No	Yes	No
Cultural conventions					
Date and time representations	API	API	API	API	None
Number and currency representations	API	API	API	API	None
Unicode support					
Character properties	API	API	None	API	None
Unicode Collation Algorithm	Available	Integrated	None	Available	None
Unicode Bidirectional Algorithm	Integrated	None	None	None	None
Standard Compression Scheme	None	Integrated	None	None	None
Character encoding support					
Multiple character set support	Fixed	Extensible	None	Fixed[2]	Fixed[2]
Transcoding between character sets	Yes	Yes	No	Yes	Yes
Autodetection of character sets	No	No	No	No	No

Internationalisation feature	Windows CE 3.0	Symbian OS 6.0	PalmOS 3.5	J2ME/CDC, Foundation	J2ME/CLDC, MIDP
Application resources					
Detachment from application code	Yes	Yes	Yes	Yes	No
Runtime access through API	Yes	Yes	Yes	Yes	No
Supported scripts					
Latin	Yes	Yes	Yes	Yes	Yes
Greek	Yes	Yes	No	Yes	Yes
Cyrillic	Yes	Yes	No	Yes	Yes
Ideographic (Chinese)	Yes	Yes	No	Yes	Yes
Compositional (Thai, Korean)	Yes	No	No	No	No
Bi-directional (Arabic, Hebrew)	Yes	No	No	No	No
Fonts and text rendering					
Bitmap fonts	Yes	Yes	Yes	Yes[1]	Yes[1]
Outline fonts	Yes	Yes	No	Yes	Yes
Ligatures	No	Yes	No	No	No
Text processing					
Text boundary detection	No	No	No	Yes	No
Locale-specific collation of text strings	Yes	Yes	No	Yes	No
Input method support					
Input method architecture	API	Extensible	API	None	None
Direct input	Yes	Yes	Yes	Yes	Yes

Internationalisation feature	**Windows CE 3.0**	**Symbian OS 6.0**	**PalmOS 3.5**	**J2ME/CDC, Foundation**	**J2ME/CLDC, MIDP**
Transliterated input	Yes	No	No	No	No
Structural input	Yes	No	No	No	No
User interface layout					
Dynamic resizing of UI components	Yes	Yes	No	Yes	Yes
Selectable orientation of UI components	Yes	No	No	No	No
Total	**31**	**30**	**10**	**24**	**11**