cefischer

OS/2 Lab Notes, Issue 1

cefischer

OS/2 Lab Notes, Issue 1

Software Engineering Magazine

Bibliographic Information Published by the Deutsche Nationalbibliothek

The Deutsche Nationalbibliothek lists this publication in the *Deutsche Nationalbibliografie*; detailed bibliographic data are available in the Internet at `http://dnb.dnb.de`.

Internet: `http://verlag.cefischer.de`
E-mail: `post@cefischer.de`

Accompanying website of this publication: `http://os2labnotes.cefischer.de`
Technical support: `support@cefischer.de`

1. Printing, March 2016
Printed and bound in Germany.

ISBN 978-3-944037-50-9

Contents

Tables

Page

Figures

Listings

1

Preparing a Minimalistic Software Development Environment for OS/2

1.1 Introduction

The most precious resource programmers spend in software development projects is the total amount of times their hearts beat until they get their job done. What is important to programmers, therefore, is to lose as little time as possible to implement their programs.

Saving time in programming is a twofold issue as with any other craft. Tools are required to make something. Mastering these tools is required to use them. The quality of the tools directly influences the quality of the products made with them as the method to use them does. In the same way one can use a good tool with the wrong technique and achieve bad results, the correct use of bad tools yields not better an outcome. In order to save time, therefore, a tool need meet these requirements:

1. It must be simple enough so that it can be mastered quickly and applied with ease.
2. It must be effective enough so that it can be used efficiently and yields robust products of the desired quality.

The first point addresses the learning curve that characterizes the amount of effort invested by the user of the tool to master it as a function of time. In a nutshell, this curve should be flat so that little effort need be spent in a short amount of time. This directly yields the prime criterion for making good tools, namely that its complexity should be as low as possible. Clearly, the more complex the tool, the longer it takes to learn how to use it *and* to get used to it.

The second point calls for the effectiveness of a tool which directly is connected to the circumstance that the grade of a tool's complexity should not be higher than that of the product to make. Simple tools lend themselves more to effective use than complex ones and suffer less from malfunction. Moreover, their simplicity makes them more robust by themselves, not only in handling them, and easy mechanisms are likely to produce other easy mechanisms which bear with them robustness alike. All these factors save time: Less time is required for training, less for successful use. Of course, a tool must not be too simple since this might result in a lack of functionality needed to make products with certain standard requirements.

Besides the desired character good tools should have, their accessibility and affordability are paramount. Tools should be permanently accessible, that is they should be market-resistant: whatever happens to their creator and however market circumstances change, the tools need be available under all circumstances. In programming, this is refers

to the language, the translator and test tools, and the target platform itself. Usually, all investments in learning, studying, and understanding as well as in the making of the products created—and the products themselves—are lost once any of the aforementioned components disappear into oblivion[1]. Accessibility of tools also means that one can pay the bill to get them. It would be best if tools were free but often this is not possible. For example, a high-speed oscilloscope simply cannot be made for free. Fortunately, the situation is less severe in software engineering. Here, very good tools often are available freely. Anyway, this should not lead to the misinterpretation that only free tools are good tools. The tools we will create in the course of our OS/2 projects can be used and replicated free of charge. Both affordability and accessibility are thus granted.

This article shows, by example of OS/2 as historic yet still widely used platform, what can be done to preserve skills, effort, and existing products created for a specific system; and how to set up a minimalistic development environment to explore this system while creating simple but powerful tools for further development of both new products and a new platform that evolves from studying a matured environment.

1.2 Traditional Development Paths and Projections

Since the advent of computers for personal computing a high competition between operating systems, programming languages and its associated tools, as well as design and implementation tenets exists. When OS/2 first came out it had to compete with DOS and later with Windows, as DOS had to compete with CP/M once. While the development strategies under DOS and CP/M were not thus different, OS/2 pointed developers in a totally different direction most programmers simply did not adopt. The system kept this tradition when its graphical runtime environment, Presentation Manager (PM), was introduced which, although very similar to Windows, confronted programmers accustomed to Windows with few but important differences. Over all, the change from typical DOS to event driven programming marked a considerable move away from widely adopted methodology. At the advent of OS/2 2.0, another radical change of application development concepts was presented by IBM, namely from event- driven procedural to language neutral object-oriented software construction with *WorkplaceShell* (WPS)[2] and its underlying technology, *SOMobjects* (SOM). All these changes in OS/2 related software engineering did not take place in isolation. Windows advanced at the same time and took over great parts of the market while constant changes in hardware technology drove software companies to strive for platform independent program development so that applications became interchangeable between different operating systems as much

[1] Or their documentation does. Most software is not sufficiently documented or, as in many cases, not at all and many good programs which might be accessible still are virtually useless since their documentation is lost. This also holds true for programs available in source form. Depending on the size of software components, sources alone might not be sufficient to understand the program good enough for maintenance or refinement. Program source texts are implementations, not design material.

[2] A graphical application environment based on a runtime-adaptable tree of object classes. WPS follows object-oriented concepts of use and programming and is unparalleled in conception, implementation, and function, still.

as possible. Platform independence is considered paramount today, still. It can and could be achieved by the following means:

1. Introduction of compatibility layers into new platform products to maintain backward compatibility to existing ones. This allows the further use of adopted development standards and products on the new platform to full or at least some extent. An example is the `bind` utility of OS/2 1.x which allows to create so called *family applications*, programs that run both under OS/2 and DOS without modification.

2. Using programming languages which are supported by a wide range of tools available on many platforms. An example of such a language is C.

3. Employing virtual machines and emulators. Emulators are platform dependent programs that mimic another platform and thus allow programs written for the latter to run on the first. By using emulators, a complete platform can be preserved, including all software tools and development methodologies as well as the hardware architecture on which the software depends. For example, an 8085 MCU emulator written for OS/2 would allow the flawless execution of 8085 firmware on the system.

4. Applying combined approaches which consist of an emulator which offers an own system architecture. An example of such a solution is Java.

The aforementioned points also show that the concept of platform independence goes hand in hand with a separation of hardware and software architectures. The term architecture is used in the common sense here, that is the set of resources and facilities programmers see when they use a system. Platform independence itself and the concepts to achieve it by means of appropriate development methodologies have been treated as essential in software engineering traditionally so the idea has been carried over into contemporary product development strategies, also. Therefore, any platform dependent concepts mostly are deprecated, especially:

1. Development of "native software", for a specific, often hardware dependent (and thus less good portable) platform that is, or threading platform specific optimizations into normally platform-neutral software.

2. Programming in machine languages viz. their symbolic representation in form of assembly languages which are naturally dependent on specific processor architectures.

Anyway, these resentments which lead to the tenet of encapsulating software from its underlying hardware at almost all cost no longer hold true at the time of this writing. As programmable logic device technology has progressed rapidly since the late 1990ies, a broad spectrum of devices are available to very affordable prices, especially advanced PLAs, CPLDs, and, to some extent, FPGAs. In addition to that, tiny logic devices allow for the construction of discrete circuits in form of highly dense board-level solutions. All that paves the way to construct an open hardware platform which is reproducible with ease and affordable to everyone given that:

1. The platform is open, freely usable, and thus available to everyone unlimited in time so that any investment in mastering the platform never is lost. A broad spectrum of applications and a wide repository of development methodology can thus evolve.

2. The platform can be implemented by using industry standard parts in order to eliminate dependencies on specific manufacturers.

3. The platform is simple by design so that it is likewise simple to reproduce, yielding a wide variety of different implementations, or devices respectively, which are highly interoperable.

4. The platform is simple to program and simplicity is key in programming tool and procedure creation alike, so to ease development further and make it more time-efficient.

5. The platform's architecture is standardized, including its extensibility features, and comprehensively documented so that application development methodology becomes stable.

Therefore, an architecture can be chosen at leisure, implemented following the above-mentioned guidelines, and development take place in a strictly platform dependent way—taking all platform features into account, including most specific optimizations—without worrying about the inclusion of compatibility and hardware abstraction layers or tools to make software run on variety of different architectures.

1.2.1 Revision of Platform Independent Software Development

The aforementioned concept bases on the assumption that it is no longer necessary to support a variety of proprietary hardware architectures by highly adaptable software and system compatibility layers since the architecture itself is completely open as well as freely and easy reproducible at any one time and thus always available. The software then becomes highly platform dependent—but this platform keeps stable. The underlying hardware can evolve at the same time—but *not* by changing the platform's architecture: only its *implementation* might change when it becomes worthwhile as implementation technology advances. This concept is the base for the following discussions and the engineering projects on which this magazine reports.

Figure 1.1 shows a sketch of the proposed platform in regards to key development tools and dependencies as they evolve from compatibility issues. As can be seen, we will define a high-level design exposition and implementation language (called μODE) with which we can create both systems and applications software at a macro and microlevel. The segmented protected 80x86 architecture represents the foundation of the projected final architecture which is a superset of its ancestor with enhancements drawn from common microcontrollers. Development thus can start on a well known and supported processor platform and yields an open extensible host processor architecture (called *Ginger*).

Compatibility need not necessarily be carried to the binary level. By exchanging the assembler, machine language programs can be retranslated. As far as programmers are concerned, they need not apply any changes to existing programs nor need learn anything new should the binary representation of their applications change. Anyway, it becomes difficult to interchange translated modules between systems if binary compatibility is broken. The original 8086 variable length machine language instruction set is good to yield compact programs so it is reasonable to build on it and shield programmers from any changes on the hardware side. This can perfectly be achieved in a standardized way by microprogramming which takes place on a level unreachable by software developers.

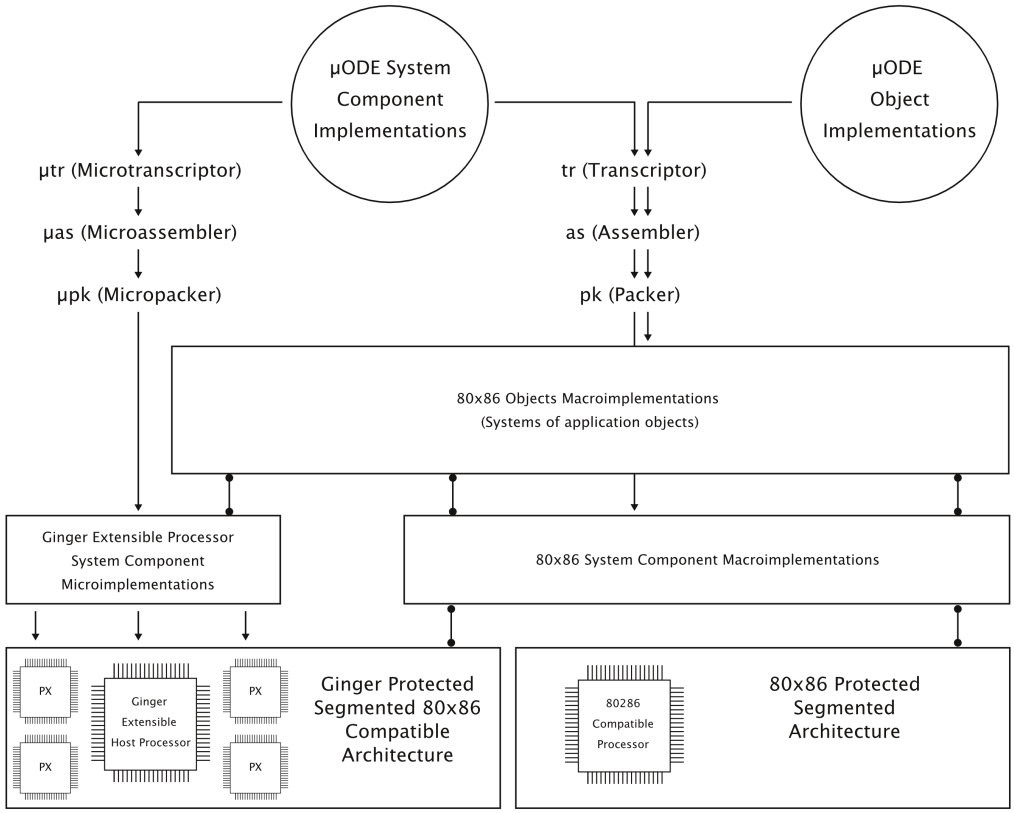

Figure 1.1: Platform dependencies and compatibility. *µODE* is the system and object exposition language suitable for both micro and macroprogramming. The architecture of the new platform as seen by the programmer at the object level remains compatible with the 80x86 protected segmented architecture, thus protecting investments of time and effort spent in application development. Keeping the architecture stable yet extensible at the microlevel employing the same programming paradigms as on the macrolevel keeps learning curves low and allows for the development of efficient, since platform dependent, software. The systems software components inherit from many concepts of OS/2, the higher-level components from *Presentation Manager* and *WorkplaceShell*, and can be macroprogrammed to be executed like common machine language programs or microprogrammed to be put "on a chip".

In the devised concept, microprogramming is the means to afford technological changes at the basic hardware level and achieve more optimal implementations of macroprograms at the mircolevel as hardware implementation technologies advance. This ensures a stable architecture and stable binary representations of programs over time. In addition, extensions to the architecture are not implemented by changing the basic instruction set but by adding new instruction sets implemented through co-processing units (Processor Extensions or PXs) which can be added to the base system as necessary. The only issue with this approach is that enhancing the architecture by processor extensions must be taken into consideration early during base platform definition. This is achieved by build-

ing mechanisms into the host processor which either interpret any external instruction by executing emulation microprograms, ignore these instructions, or asking other extensions to execute them. All this, however, is of no importance to programmers. The most important design goal is to keep the architecture stable and, along with it, programming tools and procedures[3].

1.2.2 The Role of OS/2 in Open Platform Engineering

The role OS/2 plays as a matured environment in the development of an open platform as the one proposed is twofold:

1. It serves as example of an excellent multitasking environment which greatly supports the development of multiprogrammed, highly modular applications. Studying it thoroughly thus yields important knowledge which can be used to advantage in the construction of new systems software components.

2. It introduced object-oriented concepts of use as well as object-based programming procedures and runtime environments which resulted in working implementations useful to draw from during the development of strictly object-based application runtime and development environments.

The first point can be illustrated best by studying OS/2 1.x, while OS/2 2.x merits close inspection in regards to point two and the conception of system compatibility features. Studying the operating system is done in a practical way, not only by conducting experiments with sample applications exploiting various features of the system so to derive knowledge from a matured platform; also, development tools will be created which can be used on the new platform. This ensures that software created on and for OS/2 in the course of this publication can directly be used later without any waist of time and loss of skills acquired. We thus need focus on the creation of tools first.

Leaving designing and writing a program out of sight, the conventional or traditional development path consists of compiling program source text files to object modules and linking those modules together, usually with further pre-compiled modules, to produce a version of the program which is directly loadable and then executable on the platform at which the program is targeted.

Executing a compiled version of a program can be done by several means. Either the program was compiled to machine code in which case it is directly executable by the hardware; or to some intermediate language or code which need be processed by an emulator, or virtual machine respectively. Also, the step of typical compiling can be dropped and the source form of the program executed directly by a special emulator which does not translate the program text at all but executes on behalf of it. Hence this type of program execution also is called interpretation or interactive compiling and the special compiler performing it an interpreter. We will assume that program source texts are translated or compiled into machine code for direct execution on the hardware.

[3] An interesting corollary of this concept is that the ongoing development of operating systems becomes obsolete an issue. Once systems software is defined for a stable architecture there is no need to change this software layer at all. Other means, like object classes and their inherent features such as extensibility at runtime through the introduction of replacement modules defining feature overrides, then serve for enhancing the systems software layer.

Conceiving a program as a fabric made of executable traces of control or threads, usual programs cannot be run directly on the hardware. Instead, user or special program logic is combined with other default or standard threads thus enabling programs of any kind to run in a specific runtime environment which makes available the resources of the target platform in a predefined manner. If the runtime environment is an operating system, then the user program must comply with a certain format so that the system can read it into memory and prepare it for execution. This process is called loading. Making a loadable fabric of a program using the compiled program code itself and the system's standard threads, if needed, and bringing this fabric into a distributable form is called linking. Often, linking and loading are highly mutually dependent processes. We will assume that compiled program source texts need be linked in order to be loaded by the system.

The foregoing definitions may sound familiar to professional readers and obviously superfluous. Anyway, they were presented here to give readers getting in touch with the matter for the first time a place from where to start; and to have a initial position we can contrast with alternative approaches later.

OS/2 was designed to be programmable using high-level languages from the very start[4]. The API was structured thus to follow typical compiler conventions: Parameters are pushed on the stack by the caller, from left to right; results are passed through buffers so a pointer to that buffer is passed as a parameter; the error code is returned in register AX and the stack cleaned up by the callee. Therefore, programming OS/2 with C or Pascal is a straightforward task. Furthermore, we will see that OS/2 programs are structured to lend themselves perfectly to what C compilers produce. Fortunately, at the time OS/2 was designed, programming the PC in assembly language still was much more standard; and to make OS/2 attractive for DOS programmers to change platforms, the system does not exclude assembly language programming by making it cumbersome. In fact, OS/2 programs can be written in assembly as comfortably as in high-level languages like C. The calling conventions even help in structuring the program sources when it comes to interfacing with the system which allows to automate the process of invoking the system for service well. In the course of this article and the discussions following it, we take these aspects to advantage and will focus on:

1. The authoring of OS/2 programs written solely in assembly language.
2. The automatization of common bookkeeping tasks including calls to operating system services.

We will achieve this by the construction of a high-level language using the capabilities of a macro macro assembler thus yielding self-documenting program texts easy to digest on one side and clear and lucid assembly listings on the other.

It will be shown that, following these objectives, programming on OS/2 with a macro assembler alone is sufficient from both a design an implementation viewpoint. In addition, standardization of the devised high-level language used as input for the assembler can be used to advantage in:

1. Treating the program texts input to the assembler as hardware independent program sources and thus decoupling them from the assembly process,

[4] Like the 8086 and its successors. The 80x86 instruction set and addressing modes lend themselves well to the conventions of high-level language compilers.

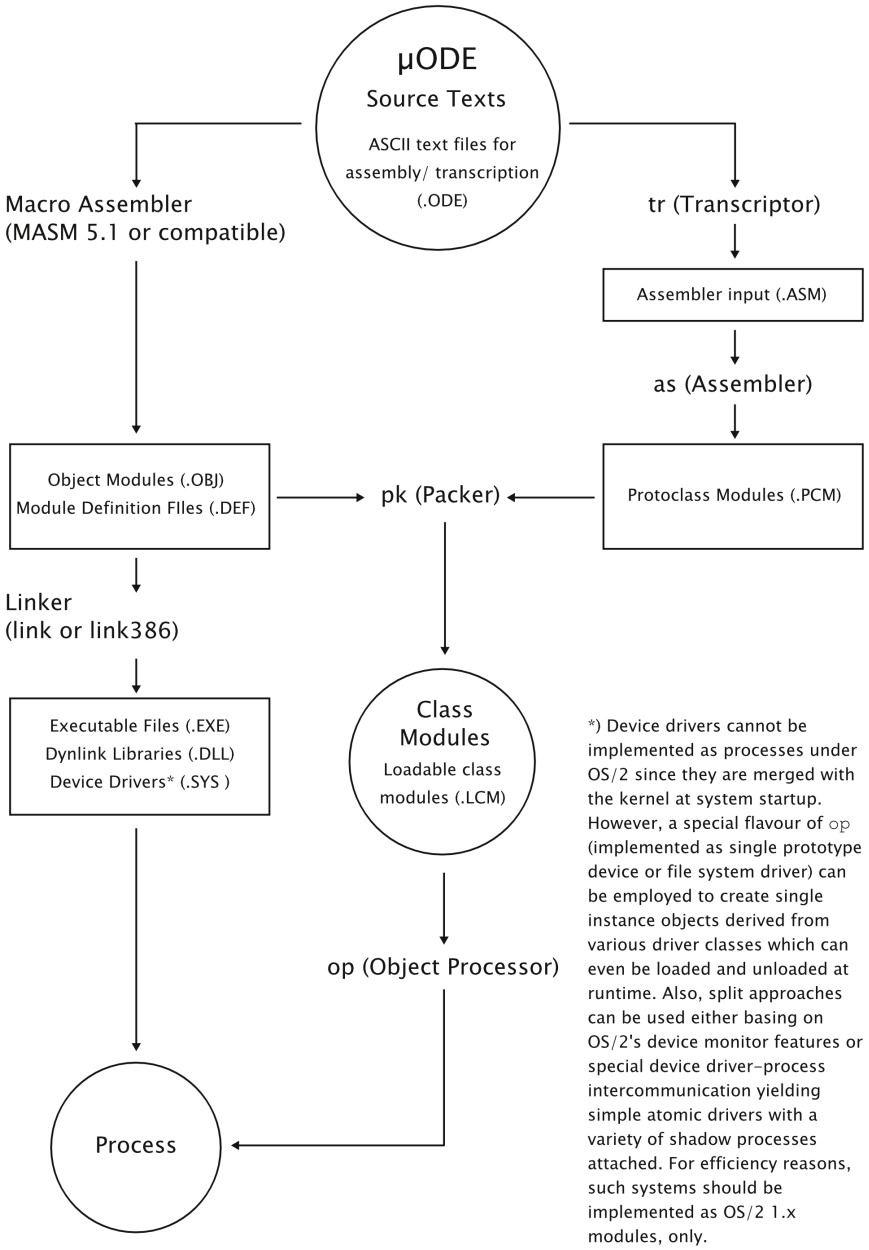

Figure 1.2: Development tools and paths for the OS/2 software projects. Starting with simple tools and following common development cycles, new basic tools are created which allow for independence of third-party software and following new trails in software engineering.

2. Treating the assembly listing output by the assembler as hardware dependent intermediate language expression of the original sources,

3. Substituting the macro assembler for a translator solely transcribing the program texts to their assembly form and an assembler translating the assembly form into actual machine language instructions.

The division of the macro assembler used during early design an prototyping phases of the development system into two separate programs reveals several interesting aspects:

1. The programs used to translate the source text to its assembly form as well as the assembly form to its machine language representation are small and simple.
2. The source text is independent of the actual assembly language representation, that is by exchanging the transcriptor the program can be translated into any assembly language while the program sources remain static.
3. The assembly text is independent of the actual machine language representation, that is by exchanging the assembler the program can be translated into its executable form for any platform while the assembly language sources remain static.

We will thus develop three important basic tools, namely a transcriptor to transcribe the high-level *μODE* program texts to assembly listings including information about how the program is to be inserted into the target system; a simple assembler to translate the assembly listing into machine language; and a packer to transform the binary machine language representation and meta information of the program into a module loadable by the system. Table 1.1 lists these tools along some information about them.

Some words are in order regarding the principal trails of software development we will follow in the course of this magazine's articles. First, we use the designation OS/2 1.x for any version of OS/2 up to 1.3 or for 16-bit OS/2 in common speak. Likewise, OS/2 2.x is used for any version of OS/2 starting at version 2.0 up to 4.5 or for 32-bit OS/2. Next, the first programs presented in this and forthcoming articles will produce OS/2 programs that can run on any OS/2 system, on OS/2 1.x and up that is. Where and when using features of OS/2 2.x, or programming in 32-bit in general, is necessary we will do so and step into the development of OS/2 2.x programs from this direction. We will always carefully weigh different flavours of OS/2 programs and system features against each other so to arrive at a software system consisting of both 1.x and 2.x applications. This will result in a hybrid application and development environment in very much the same way OS/2 itself is (for historical reasons) which combines the best of both systems, OS/2 1.x and 2.x.

In using OS/2 system services, we will make any 1.x program compatible with OS/2 1.1 and any 2.x program with OS/2 2.11 respectively. New features of version 3 and 4 of the operating systems are neither of great use nor importance in the realm of these development projects. We will look at them, of course, but have our focus on providing similar services on base of the ground covered so far when we encounter such features. This is in accordance with object-oriented development philosophy where existing parts of a system remain untouched and new features are derived from them by additional components using existing ones instead of directly changing existing parts of a program. Following this approach especially will become of practical value when we reach the highest level of programming under OS/2, namely application class development with *WorkplaceShell*.

Projected Elementary Development Tools			
Tool Name	*Description*	*Technology*	*Language*
`tr` (Transcriptor)	Transcribes *µODE* source texts to ASM286/ 386 assembly.	DOS (*.COM), OS/2 (1.x .EXE), *µPMOS* LCM	ASM286, *µODE*
`as` (Assembler)	Translates output of `tr` to 80286/ 80386 machine code.	DOS (*.COM), OS/2 (1.x .EXE), *µPMOS* LCM	ASM286, *µODE*
`pk` (Packer)	Creates OS/2 1.x + 2.x .EXEs/ DLLs and *µPMOS* Loadable Class Modules (LCMs) directly from output of `as`.	DOS (*.COM), OS/2 (1.x .EXE), *µPMOS* LCM	ASM286, *µODE*
`ed` (Editor)	Simple line-based fullscreen text editor for programming purposes.	DOS (*.COM), OS/2 (1.x .EXE), *µPMOS* LCM	ASM286, *µODE*
Projected Advanced Development Tools			
`op` (Object Processor)	Creates instances from class modules created by `pk` on DOS and OS/2.	DOS (*.COM), OS/2 (1.x/ 2.x .EXEs)	ASM286, *µODE*
`ec` (Error Corrector)	Protected mode capable ECD tool (debugger) similar to `symdeb`.	DOS (*.COM), OS/2 (1.x/ 2.x .EXEs), *µPMOS* LCM	ASM286, *µODE*
`dm` (Dataflow Monitor)	Monitors the flow of data between objects created by `op` for interobject communication ECD.	OS/2 (1.x/ 2.x .EXEs), *µPMOS* LCM	ASM286, *µODE*

Table 1.1: Projected elementary and advanced development tools. These are the tools created in the course of the OS/2 development projects. They offer a smooth transition from the R&D to the new platform. Making DOS versions of the tools allows for cross-platform development on any computer capable of executing DOS programs in an appropriate VM.

On the hardware side, we treat the 80286 as primary target processor regarding system programming and the 80386 regarding the instruction set available for application development. This is done to set a limit of the instruction set level to make the development of our simple assembler reasonably easy. Consequently, instructions of higher-level processors need be implemented in software. This may sound awkward at glance; but the main rationale behind that is that the machine language representation of our software remains at a fairly simple level and is independent of any 80x86 processor higher than the 80386. Making software dependable on specific instruction set extensions is the worst one can do to introduce several anchor points of incompatibility into one's own line of products. Moreover, by keeping the machine language representation of our programs on a simplistic and consistent level, the foundation is laid for creating the aforemen-

tioned extensible processor in a relatively straightforward manner. Setting a limit on the complexity of the instruction set to be used opens the way to such modular processor implementations while keeping existing programming methodology absolutely stable and software modules backward compatible unlimited in time.

1.3 Tools for Basic Development Cycles

Using OS/2 as development platform is rewarding since there are no commercial high-level language compilers and development toolkits these programs require available from the manufacturer anymore. Of course, free and also open source compilers can be used to program with OS/2 in C, Pascal, and other languages and sets of header files and import libraries can either be created or existing ones made by programmers who have done this already be reused. Anyway, the intention of the OS/2 software development projects this magazine accompanies is to program OS/2 using only what the system offers after a default installation plus a macroassembler which is, besides a linker, the only elementary tool we need. The linker is included in OS/2. An assembler we need add since it was unfortunately never part of the OS/2 distribution. We need these two tools only for the first initial steps. They are substituted progressively for our own.

1.3.1 Choosing the Macroassembler

For the purpose of these article series, the Microsoft Macro Assembler (MASM) 5.10 is used as macroassembler of choice. Any assembler compatible with this version of MASM is usable, also, although we do not test nor develop for interchangeability of the tool. Readers using other assemblers, therefore, must look for any possibly necessary adaptations of the code in this publication by themselves. Those owning a copy of MASM 5.10 are recommended to use this one, anyway, especially if an OS/2 version is at hand or a bound variation such as MASM 5.10A.15, the one we use[5].

Maintaining compatibility with MASM 5.10 is intended for reasons of simplicity since the resulting high-level language should be plain and implementable in form of macros with reasonable ease. This is important since the macro implementation will serve as pseudo-code representation of a working system used during the development of the transcriptor program which is comparable with the macro part of the assembler. Any specialities of assemblers more advanced as MASM 5.10 thus should not be employed since using advanced assembler features makes the development of the simple assembler to be made part of the intended development system unnecessarily complex. Complexity is to be moved upwards, towards the high-level language level that is, not in the other direction. Over all, we wish to have intelligible assembly source texts.

Alternatively to MASM 5.10, Borland's Turbo Assembler (TASM) can be used yielding the same results. We successfully cross-checked the statement macro implementations and sample programs with TASM 3.2. Turbo Assembler is a DOS program that works flawlessly under OS/2.

[5] MASM 5.10A.15 of July 7, 1989, is part of the IBM OS/2 Device Driver Kit which was accessible free of charge after registration with IBM's website.

IBM's Assembly Language Processor (ALP) version 4.00.005, made available with OS/2 Warp Developer's Toolkit 4, is compatible with MASM 5.10 but, unfortunately, not an option. The version we tested had problems with macro parameter passing in connection with pass-dependent conditional assembly which we employ in several key parts of the language implementation. Anyway, it is not worth to find a solution to resolve this issue since few readers should have access to a copy of ALP, anyway.

MASM 5.10 consists of a single file named `masm.exe`. It is recommended to copy it in the `\OS2` directory on the boot drive where it then resides alongside the linker(s).

1.3.2 Choosing the Linker

The forgoing discussion defined the translation side of program creation. We found that only by using a decent macro assembler we can start program authoring on OS/2 independent of any other tools. In addition to a text editor, this is all that is necessary to produce object files which are used further to build executable OS/2 programs, dynlink libraries, and device drivers. The part of linking files containing machine language representations of the programs the assembler produces to yield loadable applications need be discussed next.

There are two OS/2 linkers available, one for each major flavour of the operating system:

1. The segmented executable linker `link` with which OS/2 1.x modules (programs, dynlink libraries, and drivers) can be created. It comes as single file `link.exe` with any copy of OS/2 and is located in the `\OS2` directory on the boot drive.

2. The linear executable linker `link386` with which OS/2 2.x modules (programs and dynlink libraries, the latter including presentation drivers) can be created. It comes as single file `link386.exe` with any copy of OS/2 2.x and is located in the `\OS2` directory on the boot drive.

Since both linkers are readily available with any copy of OS/2 no further concern need be spent regarding this point. However, the topic of linking in general is well worth considering since we can gain insights into the mechanisms our own replacement tool for the linkers must implement.

As above mentioned, we need create own simple transcriptor and assembler programs; but, as will be shown later, a linker is not necessary to write OS/2 programs. We must rely on a linker for the first steps we take; but we can drop it soon later on and it will be the first tool that becomes obsolete. The linker basically does two important things:

1. It takes the object files produced by the translators as input and produces an executable program image as output.

2. It performs all necessary inclusions of library object files and import libraries and resolves all external references we make in our programs for the loader to fix when the program is brought into memory and prepared for execution.

The first point relates to a simple function. Principally, given we create the output of the assembler straight enough, we can produce an OS/2 executable file using a simply structured binary template into which the translated program is embedded. This process can better be described as packing than linking and a packer is the third component we will create to produce executable program files, or dynlink libraries of our translated program texts. It would also be possible to enhance the assembler thus to create executable program files directly. But this is contradictory to these two points:

1. Simplicity of the assembler: letting the assembler produce more than raw machine language representations of program source texts would unnecessarily complicate the assembler.

2. Environment independence: the assembler would be bound to the application runtime environment, OS/2 in our case, and cross-environment development would become a goal harder to attain.

So, the packer is to the assembler what the assembler is to the transcriptor: it shields the assembler from the runtime component of the system. This way, we can use machine language representations of our programs created on, say DOS machines, directly on OS/2. The packer then would produce the platform dependent executable file. Moreover, the assembler needed to distinguish between programs and dynlink libraries, further complicating the picture. Besides, the assembler should be plain enough to become part of a microprogrammed hardware implementation, eventually. So, the simpler the assembler, the more likely it can be implemented on a chip and the more modular the overall development system becomes.

The main job of a traditional linker we will lay at a side when it comes to producing our packer: the merging of different object files to one executable file and the resolving of any intermodule dependencies. Because we will do so, our object files, the one we produce through the use of the macroassembler now, must be as simple and self-contained as possible. In other words, we shall not produce any intermodule dependencies to be resolved at linktime. The rationale behind that is threefold:

1. We will no be able to work on object files of that type with our own simple tools. Never produce now what you cannot use later.

2. The programs we construct because of this limitation become simpler which makes them smaller, both in their source text and binary representations, and more dependable on other such small entities.

3. The program systems we compose become more modular and their components more reusable and isolated (and, thus, easier to error-correct[6].

Realizing these points yields a program design philosophy and implementation methodology that match the original OS/2 design tenets astoundingly well. Once arrived at the level of objects and *WorkplaceShell*, we will see that following highly modular development approaches at the most plain levels of program construction pays. Besides, simple

[6] We will generally avoid the misleading term debugging and use the more precise designation error detection and correction, or EDC, for the toil of finding and eliminating program errors, bugs in common speak.

programs are more robust, more easy to master intellectually, and lend themselves perfectly to demonstrating programming techniques and system documentation purposes for which these series of articles are intended.

Tables 1.2 and 1.3 give an overview of the existing low-level tools traditionally used in OS/2 software development, their input and and output, and their substitutes we will create.

Input/ Output of development tools			
Tool	*Input*	*Output*	*Processed by*
`masm`	80X86 source texts (*.asm) *µODE* language packages + *µODE* source texts (*.ode)	Object code OMF files (*.obj)	`link` `link386`
`link` `link386`	Object OMF files (*.obj)	Executable/ loadable modules (*.exe/ *.dll)	`System`
`tr`	µODE source texts + µODE extensions (*.ode)	Simplified 80x86 source texts (*.a86)	`as`
`as`	Simplified 80x86 source texts (*.a86)	Raw structured binary files (*.rsb)	`pk`

`pk`	Raw structured binary files	Loadable Class Modules (*.lcm)
		
		Executable/ loadable modules (*.exe/ *.dll)
			`op`
			
			`System`

`op`	Loadable Class Modules (*.lcm)	Object/ Object Class (Run-time construct)	`System`

Table 1.2: Input/ Output of traditional and projected development tools.

OS/2 has always been lacked a simple EDC tool such as `debug` DOS shipped with or its enhanced symbolic version `symdeb`, an excellent tool. Microsoft offered their source-level debugger `CodeView` as IBM did with `IPMD` for `Presentation Manager` which was part of the *IBM CSet* but both were not true an option nor are they now good templates to draw from. Instead, we will develop a tandem of two simple tools for error detection and correction, `ec`, the Error Corrector, and `dm`, the Data Monitor. They are akin to `symdeb` and draw from `MiniBug`[7] also. Besides, they are closely related to the rest of the tools and offer a wonderful opportunity to show interprocess communication techniques and process monitoring in protected mode systems. Later, they integrate seamlessly into the object systems we will compose. They are simple low-level tools which can connect to various front ends and thus follow different concepts as usual EDC tools do. It is very important to take the issue of EDC into account *early* in the design phase of software systems and *before* their implementation. The lack of a built-in system for detecting and

[7] `MiniBug` is an excellent EDC tool that was part of the 386|ASM package for the 386|DOS Extender from Phar Lap Software Inc.

Development tools, their purpose, and replacements		
Tool	*Purpose*	*Replaced by*
`masm`	Translates 80x86 and µODE source texts to machine code in intermediate object file format.	tr, as
`link`+ link386	Creates loadable/ executable files	pk, op
`implib`	Creates import libraries for resolving external references through dynamic linking	tr, as, pk
`lib`	Creates and maintains object libraries used for static linking of precompiled modules	op
`cref`	Creates human readable cross-references files from binary output of masm	tr, as, pk
`exehdr`	Reads module headers and displays key information in human-readable form	pk
`CodeView`	Allows for source-level error detection and correction	ec, dm

Table 1.3: Traditional development tools, their purpose, and replacements.

correcting errors is even more fatal to the success of an operating environment as the absence of integrated language tools. A system that is not self-contained enough to be programmed is a mess; one that cannot be monitored, field-corrected, and maintained a catastrophe. It is always better to use simple tools embedded into their host systems that work than applying sophisticated external programs that in turn need their own support systems. We want to make environments and development tools as simple as possible so for us to finalize system development and focus upon the real goal, the development of useful yet simple programs to explore, educate, and create. This is where we want to go.

1.4 Preparing of a Working Minimal Development Environment

Because OS/2 actually is a 16/32-bit hybrid system, we will develop several different flavours of modules in the course of our development projects:

1. Executable modules (EXEs) of both 16-bit segmented and 32-bit linear type with the first running on all versions of OS/2 and the latter only on OS/2 2.x and higher. These modules are used by OS/2 to build processes.

2. Dynamic link modules, dynlink modules or libraries (DLLs) respectively, of both 16-bit segmented and 32-bit linear type with the same constraints that apply to EXE modules. These modules are used by OS/2 to merge their content, both data and function, with processes at runtime.

3. Device drivers, actually a special form of dynlink modules, of 16-bit segmented type, the only one available that can be used on any version of OS/2. These modules are used by OS/2 to merge them with the kernel instead of processes at system initialization time.

All 1.x modules can contain privileged I/O instructions in code segments that are executed at IPL 2, while 2.x modules cannot, a fact we will exploit. This IOPL code, however, can be executed from 2.x executable modules using appropriate 1.x DLLs. OS/2's compatibility features are impressive and can be used to great advantage in the construction of highly efficient hybrid 80x86 software systems. Therefore, all development projects will have a strong focus on OS/2 1.x application programming[8], which also allows for various simple implementations of programs with quite sophisticated function.

Device drivers are always 16-bit modules[9] and actually extensions of the OS/2 kernel. This classes them as highly critical components and they are, unfortunately. However, their construction actually is fairly simple. Using our self-devised means for automatic code generation and employing an incremental development methodology, writing OS/2 device drivers is a straightforward process. Because these modules need be constructed around a stringent interface, they lend themselves perfectly to testing them by means of simulation before threading them into the runtime structures at kernel level. There are various types of OS/2 device drivers which mainly differ in their runtime character, the point in time when they are called by OS/2 during requests from applications that is.

The structures of all these modules are very similar: executable and dynlink modules are nearly identical; and drivers differ only slightly from normal dynlink libraries by design. We can develop all these modules with the primitive environment described here. Caveats do exist, of course, any type of module comes with its own; but they cannot be treated well by using more sophisticated tools. Better to be aware of them during design phase and to adopt a defensive style of programming.

In conducting experiments and realizing our programming projects, we will not move away from the low level symbolic representation of programs and take advantage of high-level languages and their translators. By doing so, we can take three facts to advantage:

1. The binary representations of our modules are smaller since our code is not machine-generated and does not depend on runtime function libraries with which compilers come and which they include in our own code.

[8] It is disadvantageous to shy away from OS/2 1.x programming and composing applications of 2.x modules completely. It will be shown that the price paid for intermodule compatibility such as thunking, limited virtual address spaces, and LDT tiling is much less than that what comes along with size exploded modules, memory overmanagement of local heaps, and the impact of paging on the application. Whereas reloading a few segment registers on behalf of the programmer is a *completely deterministic* process, countless page faults caused by extensive memory use for handling large virtual address spaces are *not*. And while the programmer sees any issues involved with segment handling early in the design phase of her or his modules, the actual behaviour of large 32-bit applications is unpredictable in nature and becomes evident at runtime only. Programming with segments has its drawbacks, also; but these are less severe, can be managed intellectually much better, and alleviated by more simple means.

[9] Although it *is* possible to write drivers as 32-bit modules with higher versions of OS/2, there is no real advantage in doing so, not for our purposes at least. Where and when this should becomes necessary specialities will be touched.

2. Only our code need be tested for errors and error corrected and there are only dependencies between our code—our own modules—and the system but no interrelations with any third party products.

3. In order to program our modules on the symbolic level, we must necessarily implement our solutions in a simple way, keep our documentation clean and up-to-date, and thus arrive at small programs and program systems with a high grade of modularization.

The first two points proved invaluable in practice. Once program systems become very large, intermodule dependencies start to show up faster and side effects take over in more severe ways. Having code of language products threaded into each of the modules can lead to the strangest of all error situations which are tedious or, sometimes, impossible to correct—except, probably, for the introduction of workarounds, additional function layers, or even the exchange of certain modules for variants created with other translators. By excluding complex high-level language compilers from the set of variables in our program development equation, we have many subtle issues out of our way with nothing between our code and the system and, more importantly, between our intention and the actual code. The necessity to implement more simply and the production of small, efficient modules is an acceptable corollary, also, a better understanding of the very structure, purpose, and function of our modules aside. At last, these points help in documenting the system in a direct and clear way so to watch its mechanics directly, not through the shutters of a high-level language. They do not come for free, anyway.

From the previous discussion it follows, that we need not take any issues stemming from platform dependencies into account. However, time must be considered. Programming at the symbolic or assembly language level actually means that programs are composed in an instruction-by-instruction process, that is the grade of abstraction at this programming level is relatively low. Thus, implementing the flow of control inside a program which usually comes excellently with the overall structure of a high-level language can become a tedious and time consuming task at the assembly level. We can alleviate and even completely resolve this problem by defining appropriate design language structures through macros thus automatizing the generation of program code and combine the best of both worlds, abstract high-level language and efficient symbolic programming.

1.4.1 Preparing the Tools

As discussed in the previous section, we will develop any tool we require as we explore and document OS/2 and realize our projects and only need three basic tools to start:

1. MASM 5.10 or a compatible macro assembler which can produce code for both the 80286 and the 80386 (standard and privileged instructions) and generate output in relocatable object format, or object (.OBJ) files respectively.

2. `link` and `link386` or compatible linkers which accept .OBJ files as input and produce all various types of both OS/2 1.x and OS/2 2.x modules.

3. An editor with which ASCII text files can be created.

Additional tools are not necessary. This includes applications for source code management, sophisticated make utilities, and other such components.

Besides the aforementioned programs, we will make extensive use of OS/2's command line interpreter `CMD` and its ability to interpret batch programs with which we not only can reduce general bookkeeping work during module development; also, we can use them to test our programs and let them interact under our direct supervision. In addition to that, we will explore the system itself using the command line interpreter and later carry over this type of interface by giving each object on the workplace an own prompt view so, `CMD` is well worth studying.

The type of editor is of no great concern. Recommended is the use of `EPM` which comes with OS/2 2.x and higher, runs under *Presentation Manager*, and is a comfortable tool for programming. `EPM` aside, OS/2 lacks a good simple built-in editor for programming right from the start but there are numerous editors available in the public domain. Because OS/2 can execute DOS programs, a good and simple choice, also, is `EDIT` which can run in a DOS window on the workplace, side-by-side with other applications. We can use a wide variety of good DOS tools on OS/2 as long as our own are not finished, including editors and MASM compatible assemblers[10].

1.4.2 Preparing the Environment

In preparing the development environment, we only need take care of the addition of the macroassembler to the system.

MASM 5.10 is in a single, independent file called `masm.exe`. We simply copy it in the `\OS2` directory on the system's boot drive where `link.exe` and `link386.exe` (only on 2.x and higher OS/2 versions) are located already.

MASM looks for any files included in primary source file passed to it in the current directory. If the requested file cannot be found there, MASM looks up each directory specified by the `INCLUDE` variable in the environment. We will make use of this circumstance and store our *µODE* language files in an appropriate directory separate from our module directories and include the path to this directory in the environment. We prepare this environment with a small batch file instead of editing `CONFIG.SYS`. The latter method would require a system reboot and make the setting global which not always is a good choice. By moving environment alterations into separate batch files, they are made local to the processes that run in the same command line session that run the batch file. The file we use is called `ev.cmd` (for environment) and simply reads thus:

```
@ECHO OFF
SET INCLUDE=C:\µODE;
```

Listing 1.1: Simple batch file to set up the environment for `masm`.

This assumes that the language package directory is called `µODE` and located on drive `C:`. For simplicity, *all* language package files are placed in this single directory. We will add further statements to the environment batch file later if necessary. The batch file itself is stored best in the `\OS2` directory on the boot drive alongside the other tools.

[10] The website of the *FreeDOS* project at `www.freedos.org` is an excellent address to find tools suitable for our purposes.

To make the environment work, we simply open an OS/2 command prompt, either a windowed or fullscreen session, type `ev`, press `ENTER`, and we are ready to go.

1.4.3 A Simple Make Utility

We now devise a simple make utility with which we can conveniently call the actual development tools, macro assembler and linker. We call this make utility `mk`. First, a few aspects regarding the development process should be considered.

Because we develop our software following a modular concept each of our modules usually consist of a *single* source file containing the input for the assembler. We assign the extension `*.asm` to source files which contain pure assembly texts and `*.ode` to those containing either *µODE* language statement implementations or sources of programs written in *µODE*. MASM can use either files but assumes a default extension of `*.asm` so we need specify a full file specification on the command line for our *µODE* files. The file types are assigned thus for forward compatibility with our own tools since the assembler `as` will only accept `*.asm` files and the transcriptor `tr *.ode` files only. If we follow the proposed type convention now we can rework our existing projects with the new tools directly and have everything in the right place. MASM creates object files which are fed into the linker.

The OS/2 programming concepts themselves reflect highly modular methodology so we construct our software systems on a module-by-module basis, writing one module at time and store it each in a separate file as input for the assembler. Each such module source file usually is accompanied by a module definition file containing information about this module which is passed to the linker in order to construct it in the actual making of the binary module loadable by the system. The OS/2 linkers use default options when they shall link object files with no module definition specified but we will define these files right from the start. They have the extension `*.def`.

Tinkering with two different types of files to feed two different programs is not nice. Fortunately, we can automate the creation of the module definition file accompanying a program or library module through the macroassembler's `%out` feature which allows us to display any type of string on the console (or standard output device). The article *Principal Structure of OS/2 Programs at the Assembly Language Level* in this issue shows the actual use of the feature. In a nutshell, we will put parts of the module definition file at certain points in the source text of our programs which then are displayed on the screen along other output the assembler produces during assembly time. Once the source translates correctly, we simply suppress the statistics on successful assembly and the assembler's copyright notice. Data defined by `%out` then represent the only output. Redirecting it in a file creates an appropriate module definition file. For example, the command:

```
masm /t test1c.obj,test1c.lst,, > test1c.def
```

will create the program's object code `test1c.obj`, an assembly list file `test1c.lst`, and uses `CMD`'s redirection features to write any output we trigger during assembly in the module definition file `test1c.def` which accompanies the module. After assembly the program is then immediately ready to be linked:

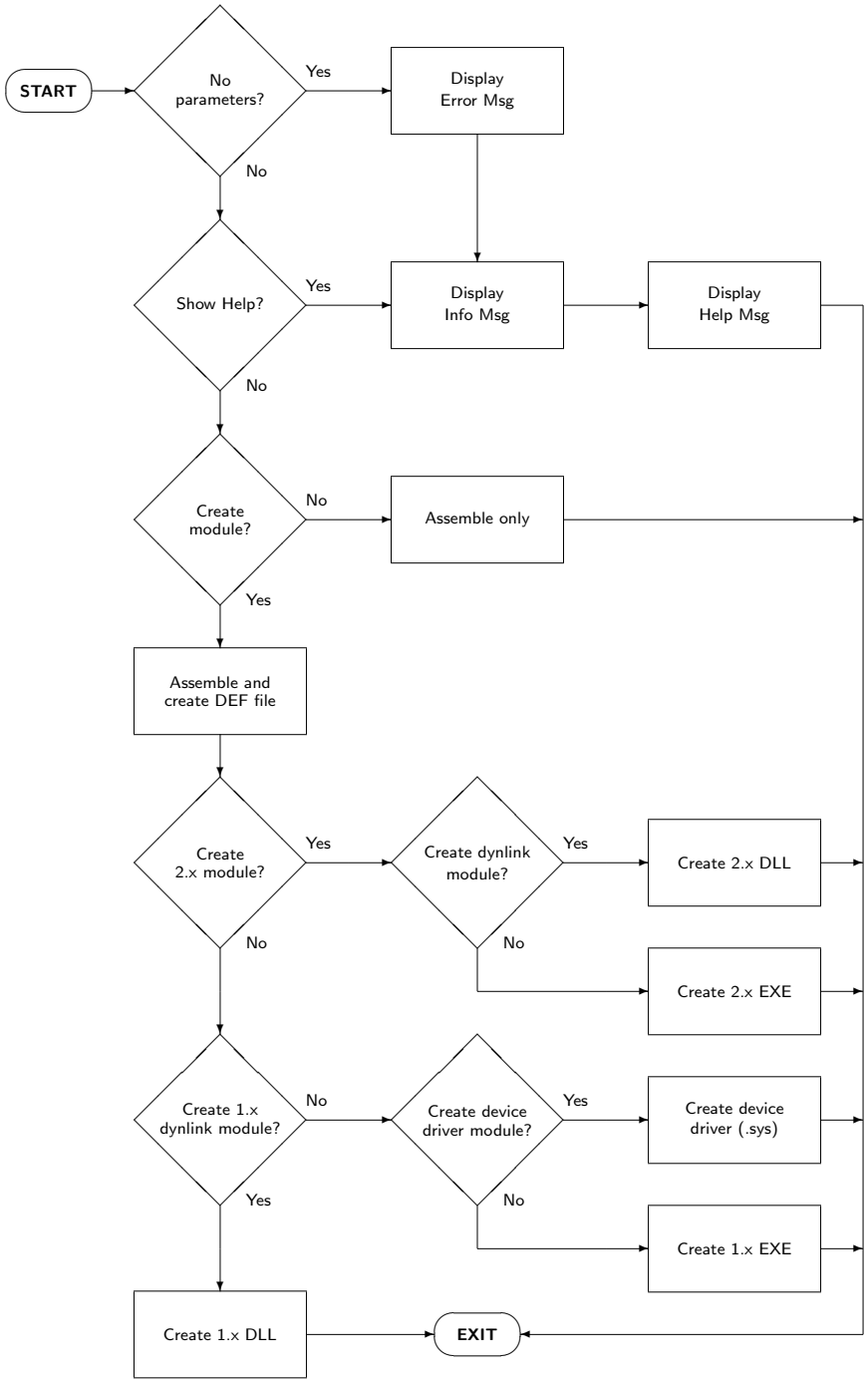

Figure 1.3: Flow chart of the simple make utility `mk`.

```
mk [/h] <source file>[.ext] [/f] [/32] [/lib] [/drv]
   Parameters    : <source file>
                     name of input file; obligatory.

                   [.ext]
                     extension of input file; optional. If not specified
                     *.asm is assumed.

                   /f
                     final option: create module; optional. If specified
                     alone, a 1.x EXE module is created.

                   /32
                     final option modifier: create 2.x module; optional.
                     This option cannot be combined with /drv.

                   /lib
                     final option modifier: create DLL instead of EXE;
                     optional. This option is mutually exclusive with /drv.

                   /drv
                     final option modifier: create device driver;
                     optional. This option cannot be used with /32.

                   /h
                     help option; displays program information



   Notes         : If /h is specified a help message is displayed and
                   all parameters following /h are ignored

                   Options can be given in all lower or all upper case.



   Example of use: mk test
                     -> process file test.asm, assemble only

                   mk test.ode /f
                     -> process file test.ode, create 1.x EXE

                   mk test /f /lib
                     -> process file test.asm, create 2.x DLL

                   mk test.asm /f /drv
                     -> process file test.asm, create device driver
```

Figure 1.4: Sytax of the `mk` utility.

```
link test1c,test1c.exe,test1c,,test1c
```

In order to make these invocations a little more comfortable, a simple batch file can be deceived thus:

```
@ECHO OFF
IF %2. == /F. GOTO Final
IF %2. == /f. GOTO Final
masm %1,,%1,,
GOTO Exit

:Final
ECHO Performing final assembly...
masm /t %1,,%1,,  > %1.def
ECHO Creating program module...
link /NOLOGO %1,%1.exe,%1,,%1

:Exit
ECHO.
ECHO Done.
```

Listing 1.2: Sample batch file to automatize module creation.

In this example, it is assumed that an OS/2 1.x executable is to be created. Taking the possible types of modules we will develop into account, we can specify the syntax of our smarter make utility `mk` as shown in Fig. 1.4. Its internal flow of control is depicted by Fig. 1.3. Listing 1.2 represents an implementation of the `mk` utility.

```
1  @echo off
2  REM ================================================================
3  REM  mk.cmd
4  REM  Syntax: mk [/h] <source file>[.ext] [/f] [/32] [/lib] [/drv]
5  REM ----------------------------------------------------------------
6  REM Check for help switch or if no parameters were specified
7  REM ----------------------------------------------------------------
8  IF %1. == /h.  GOTO Help
9  IF %1. == /H.  GOTO Help
10 IF %1. == /?.  GOTO Help
11 IF %1. == .    GOTO Error
12 GOTO Assemble
13
14 :Help
15 ECHO.
16 ECHO make: Assembles a source file or creates a module.
17 GOTO Info
18
19 :Error
20 ECHO.
21 ECHO ERROR: No source file specified. Check syntax.
22 GOTO Help
23
24 :Info
25 ECHO  Syntax: mk [/h] ^<source^>[.ext] [/f] [/32] [/lib] [/drv]
26 ECHO  All options can be given in upper case.
27 ECHO  If no options are given the source file is assembled only.
28 GOTO Exit
```

```
29  :Assemble
30  REM ------------------------------------------------------------------
31  REM Assemble only
32  REM ------------------------------------------------------------------
33  IF %2. == /F. GOTO Final
34  IF %2. == /f. GOTO Final
35  masm %1,,%1,,
36  GOTO Exit
37  REM ------------------------------------------------------------------
38
39  :Final
40  REM ------------------------------------------------------------------
41  REM Create module
42  REM ------------------------------------------------------------------
43  ECHO Performing final assembly...
44  masm /t %1,,%1,, > %1.def
45
46
47  ECHO Creating program module...
48  IF %3. == /32. GOTO 32
49  IF %3. == /lib. GOTO LIB
50  IF %3. == /LIB. GOTO LIB
51  IF %3. == /drv. GOTO DRV
52  IF %3. == /DRV. GOTO DRV
53  link /NOLOGO %1,%1.exe,%1,,%1
54  GOTO Exit
55  :LIB
56  link /NOLOGO %1,%1.dll,%1,,%1
57  GOTO Exit
58  :DRV
59  link /NOLOGO %1,%1.sys,%1,,%1
60  GOTO Exit
61
62  :32
63  IF %4. == /lib. GOTO LIB32
64  IF %4. == /LIB. GOTO LIB32
65  link386 /NOLOGO %1,%1.exe,,,%1
66  GOTO Exit
67  :LIB32
68  link386 /NOLOGO %1,%1.dll,,,%1
69
70  :Exit
71  ECHO.
72  ECHO Done.
73  REM ================================================================
```

Listing 1.2: Batch file implementation of the tiny make utility `mk`.

As can be seen in the listing, we simply call the assembler and linker with the correct switches to yield the desired output files. This is in contrast to common make utilities such as `nmake` from the OS/2 Developer's Toolkits which support incremental updates, partial compilation of files, the inclusion of different tools, etc, and even base on an own scripting language to set build processes in motion that span complex directory structures, subprojects, and possibly thousands of source code files. We will not follow this trail since it introduces flexibility into the development process were it is not needed. The process can be simplified to great an extent by keeping actual software crafting, that is the authoring of program text or writing program code respectively, on a module level and thus develop an application module by module *keeping each module small*. Applying this methodology makes complex make programs superfluous and creating project directory structures as proposed in the next section eliminates the need for any sophisticated source file repository management and its associated tools.

1.4.4 Simple Methodologies for Project File Management

In designing and implementing our projects, we will always work on two separate levels, a module and module system level. That is, a project is broken conceptually into as many independent modules as appropriate, usually as possible; then each module is created separately; and finally the devised interaction between these modules is implemented on a module system level by means of employing intermodule communication protocols and corresponding control sequencing schemes[11].

The consequence of this approach is that each module is small and that there are no intermodule relationships at the source text level as there are in the case of large monolithic projects that often comprise dozens or even hundreds of source code files. Practically, therefore, we work with very few files to create any one module and combine these modules later at runtime to yield a functional entity. Dependencies between modules that exist at runtime only are dynamic or transient and not static or persistent as they are as a result from source test interdependencies. We thus work with very few files per module, usually only one which contains the modules source text, and in some cases a few more to store resources or other module related data separately from the program source text. All other files are created by the development tools, such as object, list, module definition, and executable files. This fact simplifies source code management tremendously.

At runtime, each module will be used by others and often hierarchies of modules exist. Under OS/2, these hierarchies are called process trees. Process trees consist of parent nodes and child nodes and can take on any form. They can be modelled very well by creating a hierarchy of directories in a file system. This fact can be used to advantage for project file management in that the a main directory is created for the project, then each module gets an own subdirectory under the project main directory. Further modules are then inserted into the subdirectory structure thus that the structure itself documents the structure of the working module system at runtime. For example, an assembler project system might consist of a source code parser, file storage controller, translator, and console module and might take on the form as depicted by Fig. 1.5.

Each subdirectory in this tree represents one module and holds the source files of only this module. Later, when all modules are tested, the module in the top (the project's main) directory can be copied together with all other modules of its tree to a common subdirectory on the system drive which holds all modules in the system and started from there. This top module then builds the tree of modules and orchestrates their working and interacting[12]. The overall project management procedure reads thus:

1. Create a project directory and for each module that becomes part of that project an own subdirectory.

2. Create source files on a per-module basis and store them in the corresponding module directories.

[11] This sounds much like a mixture of event-driven and object-based programming and it is. It will be shown that the concepts of event-driven programming are independent of graphic user interfaces based on messaging systems, although programmers on the PC got in contact with events and messages mainly through such GUIs the first time. We will employ the concepts of events, messages, and objects early in our projects, long before we get in touch with *Presentation Manager*.

[12] This principal design will be implemented later in form of *WorkplaceShell* classes directly thus resulting in a simple, robust, object-oriented development environment.

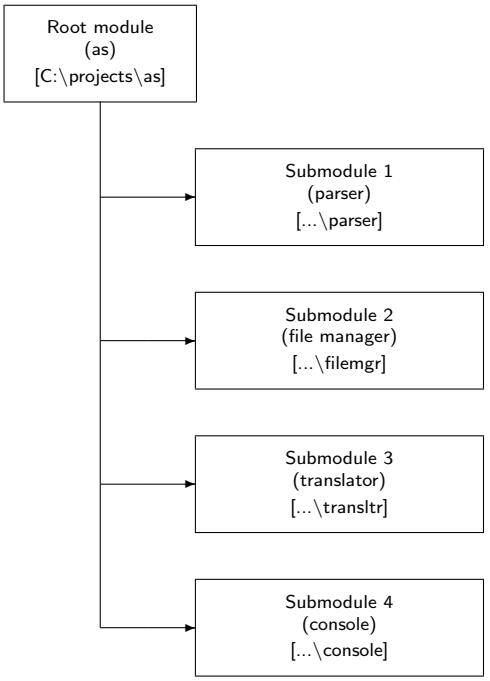

Figure 1.5: Sample project directory structure. The structure of the directory tree directly corresponds to the tree of processes at runtime created by the root module and thus is part of the project documentation.

3. Work on the nodules in an isolated way. Never create source text dependencies between modules. Note that intermodule dependencies exist during runtime only, may vary, and can take on any form.

4. Working on modules actually means to change into the module subdirectory and manipulate the files in those directories directly.

5. Under OS/2, open a new command line session for each module so you can work on various modules in parallel.

6. Place all development tools in a directory added to the `PATH` environment variable so that `CMD` can find them.

7. Call the `ev` batch file to prepare the environment for each command line session.

8. Modules are always independent so they can be run from the development console directly and then tested. Put any test drivers directly in the module's subdirectory and write a short batch file to pass data into the module and display data the module returns. This allows for an easy, interactive way of program testing.

An interesting fact follows from the proposed directory structure, namely that modules can be classed as private and public in the sense that any module that resides physically beneath a root module belongs to this root and any root can be a logical subordinate

module for any other module in the system of modules. Thus, in Fig. ??, the `as`, `pk`, `file manager`, and `console` modules can be considered public, whereas the modules `parser` and `translator` are private to `as`.

These public-private relationship is more of conceptual meaning, however, there are no root-submodule interrelationships at the program text level that is and thus each module is developed and tested in isolation, still. Private modules simply are not used outside the application of a certain root module.

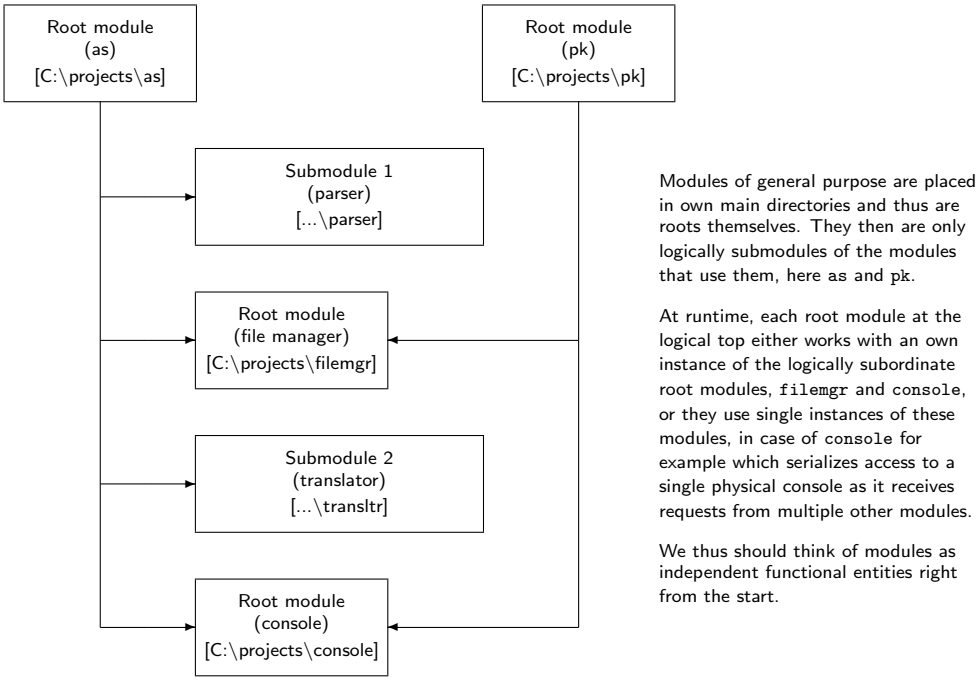

Figure 1.6: Sample project directory structure with multiple root modules. Each module of general purpose is assigned an own root directory. Under WorkplaceShell, a shadow of each general purpose root module directory can be created in the root module directory that uses other roots as subordinate modules so that the physical structure of each module project still reflects the logical structure of the module system at runtime.

Virtual inheritance, or virtual subclassing respectively, is another concept we draw from *µPMOS* and introduced early in the OS/2 development projects since it can be implemented using OS/2 processes nicely. To understand it, we need take a look at class trails and class trees.

In *µPMOS*, classes are modules that usually are merged with the runtime context of a generic or prototype task thus yielding a construct called an *object*. At the runtime level, deriving an object from a class thus means to splice the generic class handler of the prototype class with one class module registered with the system. This process also is called *morphing*. Although there is only one class from the viewpoint of the prototype task, this class actually might be a head of a train or (at runtime) trail of classes, that is it can build on another class, and this one on just another class, and so forth, until a tail class is reached. This class hierarchy normally is linear only and enables a class

designer to introduce new features into the class' functionality accessible through the head module of the class trail. Also, corrective updates can be threaded into the class trail by adding new modules at any place in the sequence of class modules. This is one point we carry over to our development tenet reflected by the environment we are preparing: Any module, private or public, can have any number of submodules strictly belonging to this module, residing in subdirectories below the module directory. This is depicted by Fig. 1.7 from which we can derive two important facts:

1. Subdirectory trees of actual modules holding the module trails do not get deeper than one level.
2. Intermodule relationships introduced by applying the concept of class trails are restricted to one type of modules only, namely those which are part of the trail but there are no intertrail dependencies.

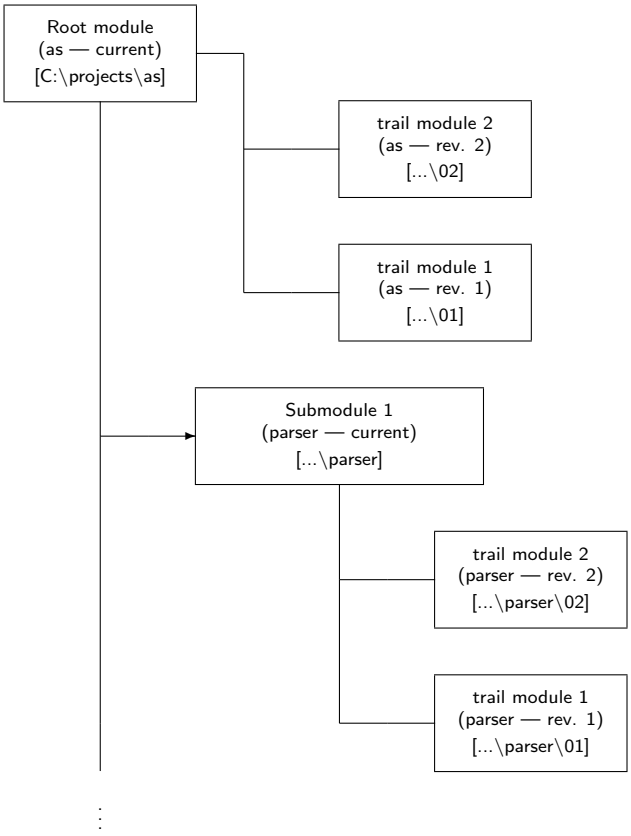

Figure 1.7: Inheritance schemes can be realized independent of the proposed directory use. Either the type of inheritance is fully virtual or intermodule and trails become logical constructs, or the inheritance is a hybrid scheme of intramodule (intern to trails) and intertrail (see text).

In common object-oriented environments, such intertrail connections do exist. It is usual that there is a single root class from which *all* other classes derive so that certain

basic features are common to every class in the system[13]. Adaptations are then made by means of subclassing as in the trail concepts but now *all* classes derive from the preceding root class and can also have further classes descending from them in the same manner. This yields a tree of classes with principally any number of branches with each branch being more or less related to one another but, eventually, all leaves of the tree *are* related in *some* way since all stem from the same root class (or set of root classes) at the very top of the structure. The direct result is flexibility; but as inherent the tree concept is to object-oriented programming as many side effects are inherent to its implementation. Imagining a tree with many branches and two leaf classes, one at the extreme right, the other at the extreme left of that bushy tree, then it is clear that these two classes are completely different in function, yet they are related in some way. Intermodule dependencies exist and wind through the whole tree building complex paths. Since all those modules run in the same context of an object, eventually, a great spectrum of side effects evolves that might or might not materialize in erratic behaviour but if and when it does the resulting failures are usually complex, of unpredictable character in both time (when they occur during runtime) and context (at what occasion they appear). This is one serious drawback of object-oriented software systems.

To alleviate the situation, *µPMOS* applies the concept of *virtual inheritance*. This simply means that intermodule dependencies exist inside class trails only and wind up along that trail until its tail is reached. Now, to let the trail stem from a common strain, the link to the *head of another trail* is established *dynamically at runtime*. That is, the intermodule dependency that would be introduced by letting one module descend from another is substituted for an intertrail dependency. Both trails exist in separate runtime contexts so they are encapsulated and even work asynchronously (and, therefore, do not even have to share the same local machine). Where the head of one branch of a class tree interacts with the leaf of another one branch by calling procedures of the ancestor class in the common model, the tail of one class trail interacts with the head of another trail by means of intermodule communication. In practice, two separate processes exist, each implementing one runtime implementation of a module trail. They are physically independent yet functionally dependent, still. The ancestor trail thus gives the descendant trail its features by means of interprocess communication instead of intraprocess calls between modules. Hence the inheritance scheme is called virtual. The results of both models are exactly the same but in the virtual inheritance scheme each class trail can be assigned another one trail at runtime[14], temporarily or persistent over the life time of the application. This way, errors can be detected and traced logically along a full inheritance path yet their analysis and treatment can be done inside each physically isolated fragment which simplifies error correction. In practice, most errors spread along a trail and then show up at the communication port to the ancestor trail(s) where they can be intercepted and eliminated but they cannot infect a complete system of class modules. This is one important premise in the construction of robust, dynamic, and highly complex distributed software systems. Besides, changes to ancestor classes often are critical since they affect *all* descending classes. Virtual inheritance makes it possible to run multiple versions of one ancestor trail side by side and let various subtrails execute in parallel with each of the available ancestor trails. This way, erratic

[13] An architecture with which we become acquainted when we explore *SOMobjects* and *WorkplaceShell*.

[14] Or even multiple trails thus implementing a robust form of multiple inheritance.

trail interoperabilities can be defeated that might affect only specific descendant trails or new experimental features can be introduced by substituting a tested ancestor trail for an advanced version, including a fallback to the prior edition on failure of the new one[15]. Figure 1.8 depicts the concept of virtual inheritance which, besides, has no impact on the directory structure we maintain in our simple environment, it is a runtime feature.

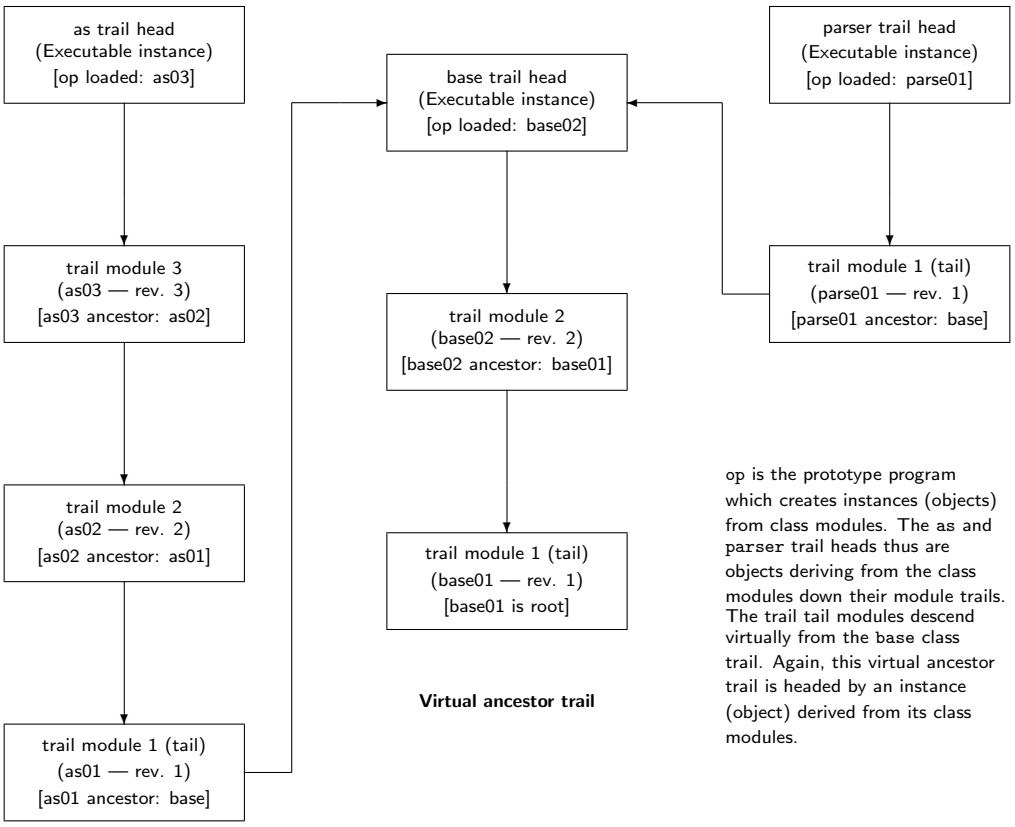

Figure 1.8: The interaction between trail tail modules and their virtual ancestors takes place based on intertrail (object) communication rather than invoking class module functions and is transparent to the trail head modules and—depending on implementation—even to themselves.

Each module still is created and maintained in isolation and any possible module trail is hidden behind a single module other modules see. Virtual inheritance is a feature whose discovery is beyond the scope of this article and thus need be discussed as part of the interprocess communication (IPC) services of OS/2. The important point to note at this time is that complex systems of highly intercommunicative modules can be built on top of OS/2 with ease, only using the simple tools presented here which work in a bare-bones development environment as the one we devised.

[15] This sort of dynamic class replacement is used in the implementation of *ObjectWorkplace ED* for OS/2. Here, only a single workplace object class exits in the workplace process which is dynamically connected to different document class trails running separately from *WorkplaceShell*.

1.5 Release Notes

This article explained how to prepare a minimal command line centred software development environment under OS/2 by simplest means only using components included in the system distribution, except for a macroassembler which need be added and, probably, a text editor with which the reader is most familiar. Also, it showed two simple batch file utilities one of which helps in making the creation of executable modules much more comfortable.

Besides the few necessary tools required for studying the operating system and creating further components for software construction, the rationale behind such studying was explained and how a new open architecture can be derived from research taking place on a matured platform such as OS/2. In addition to a design exposition and implementation language, at least three basic development tools to create the new architecture and prototype its systems software layers can be constructed using the proposed environment, a transcriptor, assembler, and packer, making the basic development tools with which software development starts, namely linkers and macroassemblers, superfluous.

The foundation was laid for software engineering by simplest means, modular system construction, reusability of program functionality by distributing finished modules instead of code, and the elimination of static linking as well as the management of complex source code repositories. Following the guidelines and design tenets proposed in the article, the way is open to explore the OS/2 operating system at all its levels and let open and freely usable solutions evolve on the experiences so gained.

References

[1] Fischer, Carla. *The µPMOS Primer.* cefischer. ISBN-13 978-3-944037-83-7.

[2] Jamsa, Kris. *Concise Guide to MS-DOS Batch Files.* Microsoft Press. ISBN-10 1-55615-638-3. 1994.

[3] Duncan, Ray. *Advanced OS/2 Programming.* Microsoft Press. ISBN-10 1-55615-045-8. 1989.

[4] Iacobucci, Ed. *OS/2 Programmer's Guide.* Osborne McGraw-Hill. ISBN-10 0-07-881300-X. 1988.

[5] Deitel, H.M. and Kogan, M.S. *The Design of OS/2.* Addison-Wesley Publishing. ISBN-10 0-201-54889-5. 1992.

[6] Letwin, Gordon. *Inside OS/2.* Microsoft Press. ISBN-10 1-55615-117-9. 1988.

2

Principal Structure of OS/2 Programs at the Assembly Language Level

2.1 Introduction

This article shows, by a simple example, the principal structure of OS/2 programs when seen from the assembly language level. Therefore, it reveals the most basic programming architecture OS/2 has to offer. The actual binary structure of programs at runtime, or their memory footprints respectively, as well as the layout of executable files resulting from the build process after linking are touched in passing.

In the course of this article, the designation OS/2 1.x comprises all OS/2 versions from 1.1 up to 1.3 (the last of the 1.x series), and OS/2 2.x all versions from 2.0 to 4.5 (the last of the 2.x series). In contrast, the widespread terms 16-bit OS/2 and 32-bit OS/2 principally are not used because they are technically imprecise and confusing. OS/2 is a 16/32-bit hybrid system; and any application under OS/2 can (and will) execute in either 16-bit or 32-bit code segments at any one time. Besides, it is a marketing myth that applications composed of only 16-bit segments generally execute slower than their counterparts made of only 32-bit segments. Many of the OS/2 development projects will show that there is no general gain in performance when moving code from 16-bit to 32-bit addressing schemes. The overall performance of applications generally depends on the structure of the underlying software system; and in case of OS/2, which is a multiprogrammed system that can execute multithreaded program modules, also, the overall performance especially depends on additional factors such as the number and size of modules composing the application, intermodule communication, thread synchronization, and other crucial points.

The articles introduces a minimal working OS/2 program in assembly language which can either be made a segmented executable module that runs on all versions of OS/2 or a linear executable module compatible with OS/2 2.x only.

2.2 Layout of OS/2 1.x and 2.x Programs in Assembly

As mentioned above, OS/2 is a 16/32-bit hybrid system, it consists of both 16-bit and 32-bit components that is. Put more precisely, it is a system that employs two different protected memory models and addressing schemes, namely a 16-bit segmented model as introduced by the 80286 processor which addresses objects in memory by variable 16-bit long selectors and 16-bit long offsets; and a 32-bit linear or flat model introduced

by OS/2 2.0 the first time[1] in which objects in memory are addressed by an invariant[2] selector and a 32-bit long offset. This differentiation is important during examination of the system at the binary level and for runtime related analysis since the two memory models are implemented differently by the 1.x and 2.x variants of the operating system. However, all OS/2 versions starting at 2.0 are backward compatible with any 1.x version. Support for both, the complete 1.x memory model and 1.x programs on higher versions of OS/2 is integrated into the system absolutely seamlessly so, little to no differences show up at the assembly language level. Basically, the assembly language programmer sees an OS/2 program as depicted by Fig. 2.1.

Before discussing the parts that make up an OS/2 program, some words are in order regarding the runtime character of applications under OS/2. We will see later that the principal OS/2 application unit is the *process* which is a compound of resources (code, data, files, pipes, etc) and processing time (threads). In its most simple form, a process refers to three elementary runtime building blocks:

1. An own stack *solely* used by (the first thread in) the process (the so called *main thread* which is started by the system when the process is created). It is important to note that the OS/2 kernel uses its own stack, that is stacks are switched automatically when execution enters the kernel (during system service invocation).

2. A data segment that contains the data upon which the process works. Although this segment is defined by the programmer, a new copy of it is assigned to each new process of the same type.

3. A code segment that contains (all or at least key components of) the application specific program text executed during the lifetime of the process. All processes of the same type use this single code segment.

The data segment mentioned in point two is called the *automatic data segment* since it is created automatically for each new process the systems starts. It contains the stack, also. Physically, therefore, per-process data and stack are stored in one and the same segment. The automatic data segment constitutes what is called an *instance* of a program whose functional characteristics are defined by its code segment. As many instances of a program can be created as the user desires, or the amount of system memory and internal control resources permit respectively, which all share a single copy of the code segment that is brought into memory by the loader the first time a process referring to that code is started. Thus, OS/2 programs are *reentrant*, that is we must be aware that our code segment is entered by many instances of a program simultaneously. Figure 2.1 illustrates these points.

---

[1] Although this linear model was implemented on the 80386, it is not 80386 specific, much in in contrast to the segmented model of OS/2 1.x which was specifically developed for the 80286, although it is much more complicated than the underlying hardware would suggest.

[2] The term invariant means that the selector value does not change during the lifetime of a process (in contrast to the segmented model in which any segment has its own selector). However, it is not zero as the common 0:32 notation for the linear addressing scheme might suggest. The zero in this notation must be read as “don’t care” selector value so, effectively, only the offset is necessary to address any datum in the segment so referred. Anyway, the selector refers to a true segment in the linear (or virtual) address space of the process.

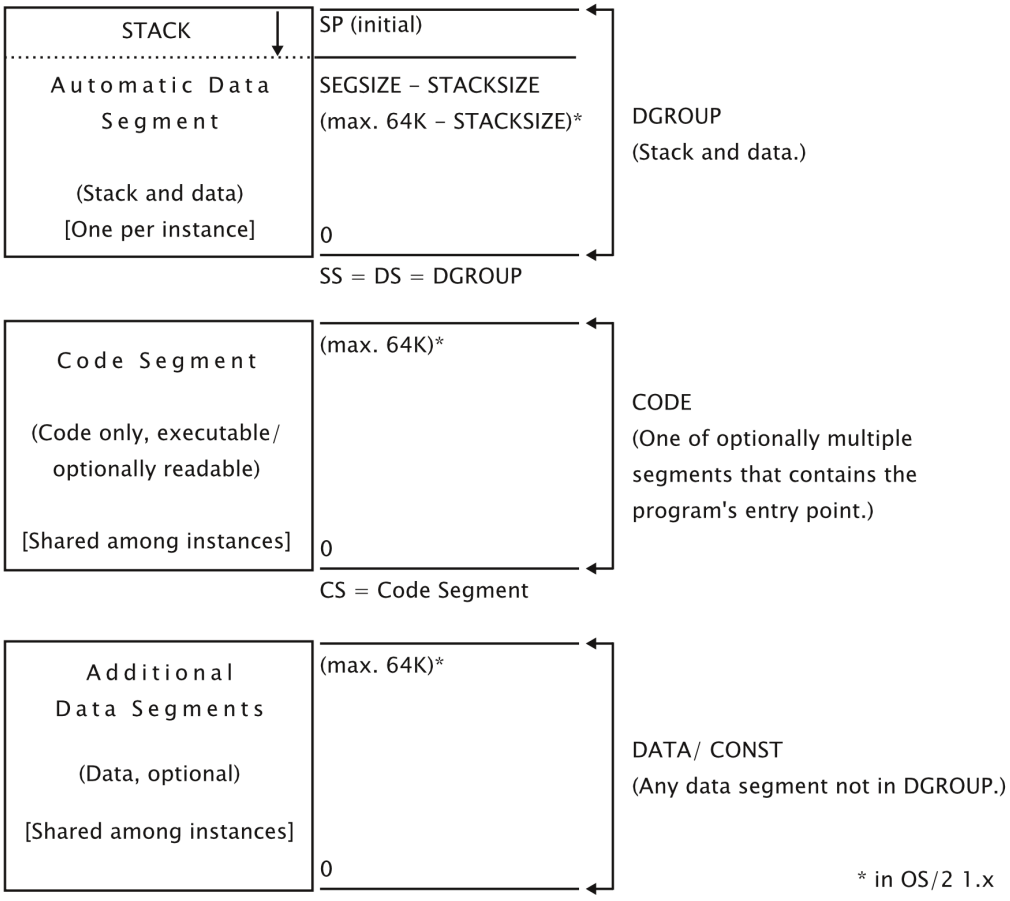

Figure 2.1: Structure of OS/2 programs at the assembly language level.

The data portion of the automatic data segment is defined by the programmer. In order to denote what data go into this automatic data segment, logical segments are grouped together using the group name `DGROUP`[3]. Under OS/2 1.x, the size of the `DGROUP` cannot exceed 64 Kbytes, including the size of the stack which is denoted as `STACKSIZE` in the figure. The data segment is not thus size restricted under OS/2 2.x whose flat addressing scheme allows for a linear address space of 512 Mbytes[4]. Anyway, one should be very

[3] For simplicity, it is sufficient to define a single data segment which holds all the program's data and is to be instantiated by the system. The distribution of data definitions among multiple logical segments is of little practical value and should be avoided in order to make assembly source texts forward compatible with `as`.

[4] For compatibility reasons. If OS/2 was are pure 32-bit system, the linear (virtual) address space of each process would comprise all 4 Gbytes addressable by a 32-bit offset. Although it sounds tempting, one should not make use of such extended memory addressability features, not in defining the automatic data segment at least, since a new copy of these data is created by the system for each new process of the same type (referring to the same module). Exhausting huge address spaces only makes sense in building big monolithic applications which should be avoided.

selective what data goes into the automatic data segment so, usually 64 Kbytes are sufficient, especially because OS/2 programs can allocate additional memory at runtime as necessary and only need store compact word-sized selectors (or double word pointers in 2.x programs) to reference these additional objects.

Data segments defined in the source text that do not go into the DGROUP are treated like the code segment, that is shared between all instances of a program. They are collected

```
1  ;===========================================================================
2  ; TEST01
3  ;  Input : None
4  ;  Output: 1 - Normal termination
5  ;---------------------------------------------------------------------------
6  ; Assembler directives
7  ;---------------------------------------------------------------------------
8                  PAGE    64,128
9                  TITLE   OS/2 1.x minimal program #01
10
11                 .286            ; select instruction set
12
13 ;---------------------------------------------------------------------------
14 ; System calls
15 ;---------------------------------------------------------------------------
16 EXTRN            DosExit:FAR
17
18 ;---------------------------------------------------------------------------
19 ; Instance data (automatic data segment)
20 ;---------------------------------------------------------------------------
21 DGROUP  GROUP    Data
22 Data             SEGMENT PARA PUBLIC 'auto'
23                 @Buffer         DB       256 DUP(0)
24 Data             ENDS
25
26 ;---------------------------------------------------------------------------
27 ; Shared data
28 ;---------------------------------------------------------------------------
29 Const            SEGMENT PARA PUBLIC 'shared'
30                 @Info           DW       16 DUP(0)
31 Const            ENDS
32
33 ;---------------------------------------------------------------------------
34 ; Program code
35 ;---------------------------------------------------------------------------
36 Code             SEGMENT PARA PUBLIC 'code'
37                 ASSUME  CS:Code, DS:DGROUP, ES:Const
38
39         Sanity:
40                 PUSH    Const           ; make Const accessible
41                 POP     ES              ; through ES
42
43         Program:
44                 NOP
45
46         Exit:
47                 PUSH    1               ; End process (all threads)
48                 PUSH    1               ; Set return code to 1 and exit
49                 CALL    DosExit         ;   Through OS/2
50 Code             ENDS
51
52 ;---------------------------------------------------------------------------
53 ; Entry point
54 ;---------------------------------------------------------------------------
55                 END Sanity
56 ;===========================================================================
```

Listing 2.1: A minimal OS/2 1.x program in assembly language.

in a separate segment called additional segment in Fig. 2.1 which is optional. Constant data like resources are common items stored in such additional segments.

This principal program layout is the same for both 1.x and 2.x OS/2 program modules (and, some minor modifications of this scheme aside, for dynlink modules, or dynamic link libraries (DLLs) respectively, also). Although all these modules appear differently at runtime, the basic structure at the assembly language level remains the same.

2.3 Minimal Working OS/2 1.x Assembly Program

Carving a minimal working program out of the concepts presented so far results in two files we need make:

1. An assembly language source file which contains the segments OS/2 uses to compose a working process in memory. Here we implement our program, actually.
2. A module definition file the linker uses to create a loadable module from the output the assembler has generated.

The structure of the assembly language file almost always can be left completely unchanged regardless of the nature of the modules we create. This way we can let a program generate this structure, relieving us from all the bookkeeping tasks, so we can avoid clerical errors and concentrate on actual programming work. Anyway, a thorough understanding of the program structure at the symbolic level is necessary since we get in touch with it in the transcriptions of our own utilities and during error correction, also. In a later step, we will derive the first high-level skeleton program from the simple template introduced next.

2.3.1 Assembly Source File of the Minimal Program

Keeping the information from the preceding section in mind, we can now implement the structure of Fig. 2.1 as shown in Listing 2.1.

The automatic data segment is defined in lines 21 to 24. There is only one segment that makes up the group named `Data`. The align type `PARA` aligns data items to `WORD` boundaries. This is the default setting but it was specified to set the segment aside from those we define for 2.x programs which are `DWORD` aligned. A stack was not defined in the source file explicitly as one would do under DOS. Although it could have been defined this way, this is done much better by means of the *Module Definition File* or DEF file respectively (Listing 2.2). This is a control file used by the linker to set up the module correctly, especially its header. The stack is defined in the DEF file by a simple statement and then configured for the program by OS/2. Since the default stack is a system-supplied runtime feature it need not be defined in the assembly source file from a conceptual viewpoint and not technically, either, since it would go into the object file where it unnecessarily took up space[5]. The `Data` segment is to receive any instance

[5] Anyway, the stack can be changed once the program is running. In this case, a separate segment is allocated through OS/2 which is made the new stack. The selector to this stack segment must be placed in the automatic data segment (otherwise all instances of a program would use the same stack). The default stack then is no longer in use and does not need any special treatment since it is located in the automatic data segment. Space reserved for the stack in the segment can then be reused.

related data items. The `Buffer` item in line 23 was defined for illustrative purposes only. Again, it should be noted that this segment should be kept small, the smaller the better, since a fresh copy is created for every instance of the program. Data vital to an instance but which is generated at runtime, should therefore be stored in a separate segment. This approach is more flexible, requires only a word be reserved in `Data`, and comes with the additional benefit of aiding in error detection and correction (debugging).

An additional segment is defined in lines 29 to 31 which is not part of `DGROUP` and hence shared among all instances of the program by default. It is called `Const` and intended as storage location for any kind of data to which `all` instances need *frequent* access. Shared segments there are usually many during runtime but not always they are shared between all instances of a program. Such data should go into dynamically allocated segments later. This also is true for data which only are accessed seldom or temporarily. We should always strive for a strict separation of data *by utilization* so that information can be discarded, refreshed, replaced, etc, at our own discretion. The definition of the `Info` item line 30 again is for illustration only.

The code segment is defined in lines 36 to 50. The `ASSUME` statement in line 37 only is of interest to the assembler in order to create offsets appropriately during translation. We need not set up the vital segment registers `CS`, `DS`, and `SS` on entry of the program to match these assumptions, as we had to do under DOS, since OS/2 initializes them so that `CS` points to `Code`, `DS` to `Data`, and `SS` to the default stack[6]. Only `ES` need be loaded with the segment we wish to access through it, `Const` in the example. This is done in lines 40-41 by a `PUSH`/ `POP` sequence. The rationale behind applying this sequence is that segment registers cannot be loaded with constant values, only from a register or memory location. If a register transfer was used the contents of at least one register would be destroyed but OS/2 passes other information on entry of the program through the registers we have not yet processed so, it is most easy to circumvent register use at this point in program execution.The stack provides a convenient way to achieve this goal.

Actual program code follows the `Program` label. The further structure between `Program` and `Exit` can be chosen arbitrarily. For example, a typical top-down approach with procedures organized in a hierarchy would result in a single `CALL` to the top main procedure coded after `Exit`, followed by further subroutines. Alternatively, each part of the program could be written directly between `Program` and `Exit` but the epilogue following `Exit` is obligatory. To satisfy system conventions, it consists of a call to an API function, `DosExit`, at a minimum. Coded as shown in lines 47 to 49 it simply ends the process in an orderly fashion.

The `END` statement in line 55 marks the end of input to the assembler and defines the entry point into the program. This is the offset to a location in the `Program` segment that will become the initial value of `IP` when the program has been prepared for execution and control is handed over to it. Here, the label `Sanity` is defined as entry point. It is used as general point for initialization and it is good practice to define it before the actual program text. It should contain instructions that not only set up the instance data for operation but that also reset the program to a known, safe state whenever this should become necessary, for example in serious error situations in which processing is aborted

[6] To recap, `DS` and `SS` point to the same physical segment, the automatic data segment, or to `DGROUP` respectively, since the stack is part of that group.

somewhere in the `Program` section and control transferred to the beginning. Hence the use of the term `Sanity` for this place.

2.3.2 Module Definition File of the Minimal Program

The second part of the program connects to the linking process necessary to create a loadable module of the assembler's output, the object (OBJ) file. In order to instruct the linker what sort of module to create, what its name is, the size of the stack, etc, a control file is created, the module definition (DEF) file.

Similar to the structure of the assembly source file, the DEF file principally need be created only once. It is used to instruct the linker to produce modules that fit our requirements but, for the most time, the linker's default options can be used. We need only redefine very few key items in the DEF file for every module and can thus automatize the generation of the file well. Where an when special options are required, their inclusion in the DEF file can easily be done by means of a simple macro. Listing 2.2 shows the DEF file for the minimal program module.

```
1  ;============================================================================
2  ; Module definition
3  ;----------------------------------------------------------------------------
4  NAME "TEST01" WINDOWCOMPAT
5  DESCRIPTION "OS/2 1.x minimal program #01"
6  PROTMODE
7  STACKSIZE 4096
8  ;----------------------------------------------------------------------------
9  IMPORTS
10 DosExit = DOSCALL1.5
11 ;============================================================================
```

Listing 2.2: Module Definition File (DEF file) for the minimal OS/2 program.

The DEF file is the same for both linkers we use, the segmented and linear one.

Line 4 is the most important. `NAME` tells the linker to create an executable (program) module and assigns a name to this module. The name need not but can be set in quotes (which is unavoidable when we use the assembler to create the DEF file). The second parameter defines the type of the module. There are three available from which to choose:

1. `WINDOWCOMPAT` which allows for execution of the program in a windowed OS/2 session under *Presentation Manager* (PM). This is the default.
2. `NOTWINDOWCOMPAT` which excludes execution of the program in a windowed OS/2 session under PM, the program only runs in a fullscreen session that is.
3. `WINDOWAPI` which defines the program as PM application.

For the time being, we use `WINDOWCOMPAT` for all our modules.

Line 6 makes the module executable in protected mode only. This is the default and need not be specified. We only make protected mode modules, besides, but keep the `PROTMODE` keyword for documentation so that the DEF file is a little more verbose in regards to describing the assembly file it accompanies.

Line 7 is important since it defines the size of the default stack. The unit of the number is bytes. We set it to 4,096 bytes or 4K here which usually is sufficient.

Line 9 opens the `IMPORTS` section where all external function references are listed. We use the `IMPORTS` feature to get access to the OS/2 API through the loader by letting the linker define a table with module dependencies. The next section defines this feature in more detail.

2.4 Accessing System Services

OS/2 itself makes intensive use of dynamic linking. Since we strive for distributed computing and a highly dynamic program environment, dynamic linking is a key feature for our projects alike. Usually, therefore, we avoid to involve the linker in statically resolving external references and in [1] a method is introduced by which programs can resolve references to external functions on their own. However, at this early time in the development process, the linkers can be a used to yield immediate results.

In a nutshell, each module is assigned a module dependency tables which lists the names of modules, usually dynlink libraries (DLLs), and the features (procedures) they provide. Any feature that resides in a DLL can be referenced by choosing a name used as symbol in the assembly language file. Referring back to Listing 2.1, we see in line 16 how this is done by example of the `DosExit` system feature:

```
EXTRN  DosExit:FAR
```

This instructs the assembler to create an external symbol and a corresponding record in the OBJ file. The symbol is declared as `FAR` since the feature resides in another module[7], here one supplied by the system, namely the basic `DOSCALL1` module. The declaration of an external symbol allows for using that symbol as if its target was defined in the same source file. When the program is linked, the linker uses the external symbol record in the OBJ file to lookup the `IMPORTS` table for an appropriate entry as shown in line 10 of Listing 2.2:

```
DosExit = DOSCALL1.5
```

This entry instructs the linker to create an entry in the module dependency table reading the name of the module, here `DOSCALL1`, and the so called *ordinal*, the index with which the feature is exported by the module that implements it. Modules can export features by name, also, and in these cases the `IMPORTS` entry can read thus:

```
FeatureLabel = MODULE.FeatureName
```

[7] In OS/2 2.x, the feature also resides in another module but the label actually is a 32-bit and, therefore, a NEAR pointer when used in a 32-bit code segment. This is possible since the address of the external feature, here the system API function, is mapped into the flat linear address space of the process.

The import by ordinal, however, is faster and there are some modules, `DOSCALL1` being an example, that export their features, or a subset of them, by ordinal number only. Anyway, given each external symbol defined by the assembler can be resolved by the linker to update the appropriate table in the resulting module, the loader can in turn resolve the corresponding import entry thus making the external library and the desired feature accessible to the process through the external label. This linker-loader scheme is the most simple to apply, hence we use it here.

Principally, every system service can be imported this way. The article *A Simple Skeleton Program for Test Purposes* in this issue shows how the process can be simplified by employing appropriate high-level language statements.

Also, there is the possibility to declare all external symbols possibly required in an external header file included in any program and then define a so called import library which defines the import definitions for whole sets of system functions. This library then is used during linking to resolve all module dependencies automatically through the linker. We will not follow this approach, however, since it makes our modules dependable on the linker, and another tool, `implib`, which is required to create the import library. Besides we tripped the wire of creating and maintaining header files which resembles typical toolkit authoring. Since we want to become independent of additional tools and avoid any additional header files, we opt for the aforementioned approach so far and let the assembler generate the linker-loader relevant statements automatically. Although this does not eliminate the dependency on the linker and requires import statements knowing the ordinals or export names of the features to be imported, no other tools are involved and it is the most simple way to go until a completely dynamic import feature has been implemented.

2.5 Creating OS/2 2.x Programs

Making an OS/2 2.x program of the 1.x variant is not difficult. The assembler only need be instructed to use a proper align type for the segments which flags them appropriately to the linker and causes the generation of 32-bit encoded instructions in code segments. Also, the instruction set of the 80386 need be activated so that all extended instructions, registers, and addressing modes become available. Referring to Listing 2.1, the following steps are necessary:

1. In line 11, the `.286` directive is substituted for `.386`.
2. In lines 22, 29, and 36 the align type `PARA` in the `SEGMENT` statements is substituted for `USE32`.

This enables 80386 programming in assembly. Besides using a 32-bit architecture of the processor, the only change noticeable is that segments now can grow beyond 64 Kbytes. This circumstance can be used to advantage in certain applications.

The aforementioned changes affect the whole program, all segments that is. The stack is accessed 4 bytes a time, no longer two; default addressing is 32-bit, not 16-bit, which affect operand sizes; and items in the data segments are aligned to `DWORD` boundaries to optimize access. Of course, it is possible to make hybrid programs containing both

16-bit and 32-bit code segments which call each other, and even the use of two differently sized stacks is possible. Neither the assembler nor the processor is a hindrance for such applications but their practical value remains questionable in the creation process of *new* modules. These should either be all 16-bit or all 32-bit. OS/2 2.x provides us with excellent support to use both types interchangeably at the same time and as long as they are kept encapsulated the system's compatibility features work seamlessly and efficiently.

The DEF file remains the same for both 1.x and 2.x modules and even the workflow remains unchanged, except that the linear executable linker `link3686` is used to create the program module instead of `link`. The next section explains the process in detail.

2.6 Invoking the Tools

The sample program introduced in this article only need be assembled and linked. The assembler is invoked as shown in Figure 2.2.

```
masm test01,,test01;
```

Figure 2.2: Invocation of the assembler to translate the minimal OS/2 programs. The command is the same for any type of OS/2 program source texts, be it modules for 1.x or 2.x programs, libraries, or device drivers.

The first parameter is the name of the assembly source file. `masm` assumes `*.asm` as extension of the file if not otherwise specified. The second parameter specifies the name of the output file. It is omitted since the default is to store the output in a file with the extension `*.obj` and the same name as the input file. We leave it this way. The third parameter is the listing file. By default, no listing is generated. We should always create a listing, however, it will become an important part of the program's documentation. We let the assembler create a listing file with the same name as the input file. Its default extension is `*.lst` so we need not specify one. We then end the input with a semicolon thus omitting the specification of a cross-reference file[8]. The invocation of the assembler to generate output of OS/2 2.x program sources is the same. Once the program sources assemble without errors or warnings, the resulting OBJ file need be linked. Figure 2.3 shows how the linkers are invoked.

The syntax of both the segmented and linear executable linker is the same, only the program names differ. The first parameter is the name of the object file, the extension `*.obj` is assumed. Next comes the name of the output file. The linkers assume that an executable program file with the extension `*.exe` and the same name as the input file is

[8] A good cross-reference file is a valuable source of information during error detection. It lists all symbols used by the assembler and assigns each symbol the numbers of lines in which it is referenced in either the assembly listing or source text file (`masm` generates references to the first). We do not use this feature here since the cross-reference file `masm` creates is not human-readable but intended as input for the `cref` utility on which we do not want to depend. Also, we will not support the proprietary `*.crf` file format by own tools. Instead, our own assembler `as` and the transcriptor `tr` will create cross-reference information by default which can also be input directly into `ec` for error detection.

```
link test01,,test01,,test01

link386 test01,,test01,,test01
```

Figure 2.3: Invocation of the segmented and linear executable linkers to create the minimal OS/2 program modules.

the target. We keep it this way and can omit the second parameter. It should be noted, however, that the name of the module was specified through the `NAME` statement in the DEF file and that the actual filename of the module can differ from that. The third parameter specifies the mapping file. It lists all segments of the module as the linker sees them in a similar way as `exehdr` does and we always generate it for documentation. The default extension is `*.map`, which we accept, so only the name is specified. By default, no mapping file is generated. The fourth parameter is used for passing the name of further object code files, often also called library files, to the linker. These additional object files are searched by the linker for external references in the primary object file and resolved. Resolving intermodule dependencies at link time is one of the primary features of a linker but we do not make use of it. Instead, we instruct the linker to build a module dependency table to resolve external references through the DEF file and later substitute resolving module dependencies at linktime for dynamic methods as explained. We thus leave the forth parameter blank. The last parameter specifies the name of the DEF file. The default is that no DEF file is used so we specify the name of our own. The extension `*.def` is assumed.

```
Operating System/2 Executable File Header Utility
Version 3.00.002 Mar  1 1995
Copyright (C) IBM Corporation 1988-1995
Copyright (C) Microsoft Corp. 1988-1992.
All rights reserved.

Module:                    TEST01
Description:               OS/2 1.x minimal program #01
Data:                      NONSHARED
Initial CS:IP:             seg   3 offset 0000
Initial SS:SP:             seg   1 offset 0000
Extra stack allocation:    1000 bytes
DGROUP:                    seg   1
Runs in protected mode only

no. type address  file  mem   flags
  1 DATA 00000000 00000 00100
  2 DATA 00000000 00000 00100
  3 CODE 00000200 0000a 0000c
```

Figure 2.4: Output of `exehdr` for the minimal OS/2 1.x program.

```
 TEST01

 Start     Length     Name                   Class
 0001:0000 00100H     DATA                   AUTO
 0002:0000 00100H     CONST                  SHARED
 0003:0000 0000CH     CODE                   CODE

 Origin   Group
 0001:0   DGROUP

Program entry point at 0003:0000
```

Figure 2.5: The MAP file created by `link` for the minimal OS/2 1.x program.

Invoking the tool `exehdr` on the resulting EXE file reveals its structure. The output of `exehdr` is depicted by Figure 2.4. It is not necessary to have `exehdr` since the same data can be found in the mapping file the linker produces as Fig. 2.5 shows. `exehdr` is only useful to gather information of modules for which no MAP file is available. This is not the case for our own modules so, the linker's MAP file and the assembler's listing file contribute profoundly to the documentation of our programs.

2.7 Release Notes

The minimal assembly program introduced in this article provides the very basis on which further OS/2 software development can evolve. Besides depicting the structure of both OS/2 1.x and 2.x executable modules at the assembly language level, it effectively demonstrates that only two simple yet powerful tools are necessary to create a working OS/2 program, access system services directly without employing any other programs or toolkits, and to yield highly compact program modules accompanied by a sufficiently detailed set of information created automatically during the make process.

References

[1] Duncan, Ray. *Advanced OS/2 Programming.* Microsoft Press. ISBN-10 1-55615-045-8. 1989.

[2] Iacobucci, Ed. *OS/2 Programmer's Guide.* Osborne McGraw-Hill. ISBN-10 0-07-881300-X. 1988.

[3] Deitel, H.M. and Kogan, M.S. *The Design of OS/2.* Addison-Wesley Publishing. ISBN-10 0-201-54889-5. 1992.

3

A Simple Skeleton Program for Test Purposes

3.1 Introduction

This article presents a simple yet flexible and extensible way of defining the principal structure of OS/2 programs by means of high-level language statements implemented in form of assembly language macros. Based on the information given in the preceding two articles of this issue, a framework of statements is derived whose transcription yields the minimal working programs given in assembly language in the article *Principal Structure of OS/2 Programs at the Assembly Language Level*.

The statements are an extension of the *μODE* programming language and take the support of both OS/2 1.x and 2.x modules into account. This article only deals with statements for program construction so that a simple skeleton program for further development becomes available. The article in [1] uses the statements developed here and adapts them appropriately to develop DLLs by the same simple methodology.

Besides the development of the necessary language extensions, the article introduces into the principal process architecture of OS/2 from a high level and shows the development of simple but effective test drivers in form of batch file utilities.

3.2 Programming with μODE on OS/2

μODE is an extensible plain block structured design exposition and assembly language devised for the creation of true objects. Besides, it is a system design and implementation language. It combines the advantages of the development approaches of both symbolic and high-level programming languages offering a well-balanced set of self-explanatory terms with which software object designs can directly be expressed in simple English statements of self-documenting nature. All statements are defined and implemented thus that they can directly be transcribed to assembly sources of uniform structure. The articles in [4,5,6] give further insight into *μODE* while [7] contains the complete specification and assembly macro implementation of the language as part of its documentation.

μODE is a transcriptor language but since the language is implemented to completion in form of assembly language macros first, it lends itself perfectly to the simple tools used to realize the OS/2 programming projects this magazine accompanies. While we do not discuss general language elements, we do focus on the extensibility features of *μODE* and will develop, step by step, a full-fledged language extension package to cover all important fields of OS/2 programming.

3.2.1 Specification of Basic Program Structures

Although our ultimate goal is the creation of true objects and object systems on OS/2, we first must develop our own basic tools introduced in the article *Preparing a Minimalistic Software Development Environment for OS/2* in this issue[1]. Therefore, we will first write language extensions with which simple, directly executable programs can conveniently be defined. Then we need similar extensions to write dynamically linkable program libraries as well as device drives. Also, in parallel to these activities, we must add appropriate system-level statements in order to access the underlying environment, the kernel and subsystems of OS/2 that is, in a systematic fashion. This article describes the first part, the definition of extensions for program authoring. The articles [1] and [3] introduce to the language extensions for dynlink library and device driver programming respectively.

As mentioned above, we must start working on a level below objects, that is we first need write programs in the more traditional sense so to be able to create our basic development tools. We will, therefore, use only elementary *μODE* statements and internal procedures but define a basic program layout from scratch. It is intended that this layout be as general as possible and shields us from the idiosyncrasies of the underlying operating system alike. Figure 3.1 depicts the proposed basic structure for our program. We will stick with this structure throughout all OS/2 projects when we need create executable programs that are made processes by OS/2 at runtime[2].

```
PROGRAM "Test"
 IS <Type>
  <Description, runtime options, ...>

 REQUIRES
  <External feature imports>

 INSTANCEDATA
  <Private data definitions>

 SHAREDDATA
  <Public data definitions>

 FUNCTION
  <Entry, start, termination sections>
ENDOFPROGRAM "Test"
```

Figure 3.1: Outline of the basic `PROGRAM` block structure.

The `PROGRAM .. ENDOFPROGRAM` block definition follows general *μODE* style. All enclosed statements become part of the OS/2 language package, however. Although some oft them do appear in the standard packages, they are not imported from there but

[1] Simply put, to create the first true object we cannot in turn use objects but first must define their elementary building blocks.

[2] And across platforms for any program that is of such a basic nature that it is not nor cannot be part of any higher-level runtime environment.

completely redefined since they bear very different meaning under OS/2 as they do on other platforms[3]. As shown in [1,3], dynlink libraries and device drivers can use the very same statements which need be adapted only slightly to meet the different modules' requirements. This way all modules can be made symmetrical from a high-level language point of view[4]. The statements inside the `PROGRAM` block are defined thus that their transcription to assembly yields the principal structure of an executable OS/2 program as introduced in the preceding article. Figures 3.2 to 3.6 show the specifications of each statement.

`IS` directly follows `PROGRAM` and defines the three obligatory key characteristics of the module, namely its type, a free form description in form of a text string, the size of the default stack (as part of the automatic data segment), and the target processor which either is the 80286 or 80386. Choosing the first processor makes the program compatible with OS/2 1.x and thus a segmented executable module is created; choosing the second makes it a linear executable program which runs under OS/2 2.x only. All three characterizing statements must be specified.

```
IS "<WINDOWABLE, FULLSCREEN | PMAPPLICAION>"
   Must be contained in: PROGRAM

   Must be followed by : DESCRIPTION "<Program description>"
                         STACKSIZE <Size of default stack in bytes>
                         TARGET <286 | 386>
                         REQUIRES (closes block)

   Notes               : The number following STACKSIZE is base 10.

   Example of use      : The following example makes the program compatible
                         for execution in an OS/2 windowed session, sets the
                         size of the default stack in the automatic data
                         segment to 1 Kbyte and marks it as native OS/2 1.x
                         application:

                         IS WINDOWABLE
                          STACKSIZE 1024
                          TARGET 286
```

Figure 3.2: Specification of the `IS` statement and its followers.

`INSTANCEDATA` and `SHAREDDATA` open the instance data and shared data sections respectively. All data declarations following each statement go in either the automatic or

[3] For illustrative purposes, the *μODE* standard macro packages are not included in these examples to avoid clutter. Once the statements defining the structure of OS/2 modules and system API access are finished, the standard packages are added so programs can actually be written entirely in *μODE*. Till then, a fallback to plain assembly inside the *μODE* structures is made (as it is always possible) which is ideal doing for the purpose of system documentation, besides.

[4] Symmetry of the module creation processes is provided through the simple make utility `mk`. In the most abstract sense, `mk` is a filter, taking a single *μODE* module source text file as input and creates its directly usable binary representation as output.

default shared segment and both statements open and close the segments appropriately. While INSTANCEDATA must be specified, SHAREDDATA is optional. Since the second statement closes the segment definition of the first during transcription, a missing SHAREDDATA statement cannot close the segment of INSTANCEDATA. As usual in μODE, an internal flag is used by each statement to mark it as open block which is checked by the next statement in line to close it. If that first next statement is optional, the second next statement takes on the role of its optional predecessor and closes the foregoing block. Here, the next statement of SHAREDDATA is FUNCTION which would finalize the segment definition started by INSTANCEDATA should it be open, still. This form of transcription context checking is typical in μODE and protects the sequencing of statements by simplest means. The technique is shown later in section 3.2.2. Also, [4,5,6] explain this feature in detail. The specifications of the two data section declaration statements are identical so only the one of INSTANCEDATA is shown in Fig. 3.3.

```
INSTANCEDATA
   Must be contained in: PROGRAM

   Must be followed by : SHAREDDATA or FUNCTION (either closes block)

   May contain         : DEFINE, all \mode\ default advanced data declarations,
                         standard assembly declarations

   Notes               : INSTANCEDATA opens the section of the program whose
                         data declarations go into the automatic data
                         segment. All data here are allocated statically and
                         a new copy of them is created by the system for each
                         new instance of PROGRAM in the system memory.

                         The statement can appear only once in a PROGRAM
                         since each program can only have one automatic data
                         segment. INSTANCEDATA must be specified.

   Example of use      : The following example opens the instance data
                         section of the program and declares an array of
                         100 words, a flag word, and two double word
                         attributes:

                         INSTANCEDATA
                          DEFINE @a AS ARRAY OF 100 WORDS
                          DEFINE @status AS FLAG AT BIT 1 OF 1 WORD
                          @dword1  DD  0
                          @dword2  DD  0
```

Figure 3.3: Specification of the INSTANCEDATA and SHAREDDATA statements.

The FUNCTION statement physically opens the code segment of the program. There can only be one FUNCTION section and thus only one code segment but this merely is a convention. OS/2 1.x programs, relying on a 16-bit selector:offset addressing scheme with variable selectors, can comprise many code segments, all defined in a single or multiple source files, with each segment having a maximum size of 64 Kbytes. In contrast,

OS/2 2.x programs, using a linear 32-bit memory addressing model, one with invariable selectors that is, have just one code segment but, in practice, approximately 6,000 times as big[5]. We will not make use of either feature, anyway. When we write OS/2 1.x programs we will only create a single static code segment per program module, namely that defined by `FUNCTION`. OS/2 2.x programs naturally go along with the single segment `FUNCTION` opens and although it can get much larger than 64 Kbytes, we will set ourselves a limit to keep 32-bit code segments as small as possible. We will see that the segments of our 2.x programs seldom get larger than 96 Kbytes, exceeding the 256 Kbyte being an exception. These are no limitations that needed to be respected under pressure. The figures represent marks derived from praxis and simply come with the overall design of the software systems we create. Anyway, we will make an interesting use of several OS/2 1.x system functions to work with code segments on a dynamic basis without resorting to the system supplied dynlink library features. This sort of program supplied dynamic loading and linking is used by `op` to instantiate objects from class modules. Therefore, although only one code segment can be defined using `FUNCTION`, many more can be created during runtime and made accessible by the program thus augmenting its functionality. Hence our programs can consist of more than just one code segment, but only one of them can be introduced by any one module. Because it is defined when the program is written we can call this single code segment the *static* code segment to contrast it with the segments added once the program runs or the *dynamic* ones respectively. Code segments thus are treated in very much the same way as data segments: what does not fit in one static segment, either physically or conceptually, is added later by dynamically creating segments, for each of which a selector need be stored within the automatic data (or even static code) segment[6].

`FUNCTION` defines three subsections, designated by `SANITY`, `ENTRY` and `EXIT`. Physically, `SANITY` marks the entry point to the program for the system, a one-time entry into the program under normal circumstances. It should be used thus, that is whenever a program returns to this point, a complete restart is implied, hence the term `SANITY`: go back here—and the program is reset to a safe and known state to start everything from scratch. The opposite point is `EXIT`: go there—and the program terminates in an orderly fashion and leaves the system gracefully. Correct entering and exiting a program are operations critical to the overall quality and robustness of an application and, eventually, a whole system so we need take good care of these special sections in `FUNCTION`. One thing that `SANITY` does is to save the initial state of the program at entry so that this state can be restored whenever `SANITY` is reset. Resetting our programs thus has the very same

[5] Theoretically, the 32-bit linear address space of OS/2 2.x programs cover segments as large as 4 Gbytes but for compatibility with OS/2 1.x, a limit of 512 Mbytes per process is imposed. Practically, however, not all of these 512 Mbytes can be used by the application, the utmost limit lies somewhere near the 384 Mbyte mark. This is of not thus great importance for real-world programs, anyway. Applications using a private code base of that size can safely be rated as principally "misdesigned".

[6] This should not give rise to the simple idea just to scatter a program's function about a fabric of many segments this program creates and owns. Such an approach would take the valuable feature of segmentation to only little advantage—only to easier error detection and program monitoring—as little the advantage of distributing large, monolithic data structures over multiple segments is. When it comes to segmentation and programming, one must think in human (object and associative) terms, not in that (linear and one-dimensional) of a compiler. Working with huge self-managed linear address spaces mapped onto a segmented architecture is awkward at best, inefficient in the worst case. Metaphorically, a big linear program in a segmented environment is a square peg in a round hole. Segmented software is much more composed than simply compiled.

```
FUNCTION
   Must be contained in: PROGRAM

   Must be followed by : SANITY
                         ENTRY
                         EXIT
                         ENDOFPROGRAM (closes block)

   Notes               : EXIT is usually followed by TERMINATE. If not,
                         ENDOFPROGRAM defines a default termination
                         operation. SANITY marks the initial program entry
                         point, the first instruction following SANITY thus
                         is the first executed after the program received
                         control the first time. Control falls through from
                         SANITY to ENTRY to EXIT by default.

   Example of use      : The following example defines a no-operation
                         function for the program and terminates with a
                         condition code of 0 (the termination is defined
                         by default if EXIT is not followed by TERMINATE):

                         FUNCTION
                          SANITY
                          ENTRY
                          EXIT
```

Figure 3.4: Specification of the `FUNCTION` statement and its followers.

effects like a warm boot of a computer system: the current runtime context mostly is not affected but the operational state is brought back to a known status from which the (probably failed) program can be analyzed, fatal operations reversed, data fields reset, or the complete status reinitialized. Such a feature is not necessarily easy to implement but extremely useful once the grade of interaction between programs advances. In the current implementation, the `SANITY` statement is not implemented. Its implementation reserves an article of its own.

Following `SANITY`, the `ENTRY` statement marks the actual beginning of the program's true function. Here all features, functions or procedures in common speak, are defined, the program's operational states and the sequencing through them. When control enters the program from the system, the operational state of the application is well defined. The `SANITY` section leads from this point through basic initialization and bookkeeping sequences, then control falls through to `ENTRY` where normal program operation begins. This resembles a two-tire system of initialization. As with `SANITY`, returning to `ENTRY` means to rewind to the start of the program but not as some means of taking the thumb off a dead man's button. Instead, any return to `ENTRY` from somewhere in the program is a usual, orderly step to end a current operation: go there—and you have completed whatever you have done and can take on the next assignment. This concept nicely leads to the communicative nature of the programs we will create, namely that one program cannot do everything on its own, it requires others to carry out its tasks as others need

it to carry out theirs so, we plan for a requestor-server-requestor or peer architecture right from the start. The implementation of `ENTRY` will reflect that when the section is filled with default language blocks such as `SEQUENCING`, `FEATURE`, `STATE`, `EXCEPTION`, and `SIGNAL`.

`EXIT` has been mentioned already as the opposite to `SANITY`. Here the program is terminated and whatever has been done in the `SANITY` section is made undone in `EXIT`. Usually, `EXIT` features the `TERMINATE` statement as its very last one to take care of obligatory cleanup steps. If not, `ENDOFPROGRAM` will issue a call to `TERMINATE`. `TERMINATE` is a system level statement as well as `REQUIRES`. Both are defined next.

3.2.2 Specification of Special System Level Statements

The `REQUIRES` block, following `IS` and closed by `INSTANCEDATA` is obligatory. It is a default language block defined for use under *μPMOS* and its purpose is to construct a so called *Class Dependency Table* or CDT for short. The CDT lists all features offered by other object classes the object defining the CDT needs to request from other objects to perform. Each entry in the CDT lists a name of the feature along with the class which implemented it. On class load time, the system resolves each feature name into a class-specific index, or ordinal number. Later, when the object creates other objects, it passes to its objects the indices of the features it wants them to carry out. This is called an *External Feature Request* or EFR. The feature index then is used by the receiving object to actually invoke its feature regardless whether it has implemented it by itself (in its own class module) or just inherited from its parent. The internal use of feature indices aside, they play a key role in the inheritance scheme and the CDT is the corresponding data structure inherent to any object. Also, the system is treated as object (or pseudo object, the kernel of an object environment itself cannot be an object), namely one to which all objects have immediate access. As any object, the system has a (pseudo) class, designated by `.System` which is used in EFRs[7] and we will follow that convention. The `.System` designation stems from the `.<class name>` syntax of *μODE* we will adopt, also. Under OS/2, then, `.System` refers to the OS/2 kernel. `REQUIRES` is obligatory because:

1. No program can execute under OS/2 without at least one call into the kernel, namely to the above-mentioned `DosExit` feature.
2. No program we will create can execute alone without requesting service from other programs of the same kind.

Thus we need at least one external feature and hence the statement to access it. Besides, it is advantageous to adopt the `REQUIRES` language feature early since the objects we will create later base on highly dynamic inheritance schemes and we can make use of OS/2's dynlink library services in order to import external features in a completely dynamic way, without the need for the linker or loader to resolve the addresses of external references as we have done so far. Figure 3.5 depicts the specification for `REQUIRES`.

`REQUIRES` is not implemented to completion in this article since a thorough investigation into the system's dynamic link features is necessary prior to such an attempt. The

[7] Which masks calls into the micro object system kernel, or MOSK for short, as normal EFRs and thus ensures symmetry of feature requests across object boundaries. At the assembly language level, EFRs to `.System` are transcribed differently than EFRs to usual objects, however, since the MOSK's few features are accessed through a simple but efficient `INT 2xh` interface.

```
REQUIRES
   Must be contained in: PROGRAM

   Must follow         : IS

   Must be followed by : FEATURE <feature name> FROM <object class> AS <alias>
                         SERVICE <feature name> FROM .System AS <alias>
                         ALL <SERVICES | FEATURES> FROM <object class | .System>
                         INSTANCEDATA (closes block)

   Notes               : If ALL is used (to force a general import) no aliases
                         can be defined and the default export names must both,
                         be used and known.

                         .System is a reserved class and refers to system
                         kernel type functions.

   Example of use      : The following example imports three system services,
                         one feature from a self-defined object class and
                         all features from a datatype class:

                         REQUIRES
                          SERVICE DosExit         FROM .System
                          SERVICE DosEnterCritSec FROM .System AS $atomic_ON
                          SERVICE DosExitCritSec  FROM .System AS $atomic_OFF

                          FEATURE $binary_to_hex  FROM .OEBase

                          ALL FEATURES FROM .OEArray
```

Figure 3.5: Specification of the `REQUIRES` statement and its followers.

implementation shown in the next section bases on the linker-loader supported mechanism to resolve external references in executable modules at load time.

A more complete first implementation can be introduced for `TERMINATE`. It is a simple statement that receives a single parameter which is a numeric code returned to the process that started `PROGRAM`. `TERMINATE` calls the `DosExit` system service and thus takes care of a proper "return"to the operating system[8]. The specification of `TERMINATE` is depicted by Figure 3.6.

Besides the `REQUIRES`, `SERVICE .. FROM`, and `TERMINATE` statements introduced so far, we will add further system-level statements which are specific to OS/2, one important of which is `THREAD .. ENDOFTHREAD`. We will discuss these extensions in the course of the development projects as we get in touch with the corresponding system features.

[8] `DosExit` is not truly a return since the operating system was not called on entry and control remained in the program. Instead, the system call resembles something like a one-way trip into the system kernel with a complete self-destruction of the process as its final destination.

```
TERMINATE WITH <termination code>
   Must be contained in: EXIT

   Must be followed by : ENDOFPROGRAM

   Notes               : If TERMINATE is not issued explicitly within EXIT,
                         this is done by ENDOFPROGRAM which returns a zero
                         termination code.

                         TERMINATE can only be used explicitly within the
                         EXIT section since it ends the entire process
                         unconditionally. In order to stop the program at
                         once from somewhere else within function, a proper
                         control transfer to an exit STATE should be made
                         specifying EXIT as next state.

                         More than one TERMINATE statement can appear in EXIT.

                         If not explicitly specified, the termination code
                         following WITH is encoded in the default radix of
                         the transcriptor/ assembler.

                         Termination codes are WORD sized in muODE.

   Example of use      : The following example terminates the program and
                         specifies a termination code of 3AFFh to be passed
                         to the program's parent:

                         TERMINATE WITH 3AFFh
```

Figure 3.6: Specification of the `TERMINATE` statement.

3.2.3 An OS/2 Application Structure Package

We can now start to implement the first elementary language statements. We will add each new statement to the same basic language extension file which we call `OS2APP.ODE`. This file need be included into any other module source text file since it will contain all basic system level statements. It will later be accompanied by two other such files, namely `OS2LIB.ODE` and `OS2DRV.ODE` each of which will either purge, or overwrite respectively, existing statements and introducing new. All extension files should be placed in an own subdirectory such as `\μODE` which must be added to the `INCLUDE` variable of the environment for the session in which the assembler runs. Refer to the article *Preparing a Minimalistic Software Development Environment for OS/2* in this issue how to prepare the environment for use.

Listing 3.1 shows a complete working implementation of the statements we have drafted in the last section. Parameter, attribute range, and transcription context checking were left out for reasons of clarity. The final implementation of `OS2APP.ODE` includes these checks to ensure a robust transcription.

```
;;=============================================================================
;; os2app.ode
;;  muODE definitions for OS/2 application programs
;;-----------------------------------------------------------------------------
                PAGE    64,128
                .xall


;;-----------------------------------------------------------------------------
;; PROGRAM
;;-----------------------------------------------------------------------------
PROGRAM         MACRO op1
if1
 @mode_progname EQU op1
 %out ;=============================================================================
 %out ; Module definition
 %out ;-----------------------------------------------------------------------------
endif
                ENDM


;;-----------------------------------------------------------------------------
;; IS
;;-----------------------------------------------------------------------------
$mode_IS_output macro op1,op2
 %out NAME op1 op2
endm

IS              MACRO op1
if1
 ifidn <PMAPPLICATION>,<op1>
  $mode_IS_output %@mode_progname, WINDOWAPI
  exitm
 endif

 ifidn <WINDOWABLE>,<op1>
  $mode_IS_output %@mode_progname, WINDOWCOMPAT
  exitm
 endif

 ifidn <FULLSCREENAPPLICATION>,<op1>
  $mode_IS_output %@mode_progname, NOTWINDOWCOMPAT
  exitm
 endif

;; Neither or unknow program type specified.
;; Output error message only once and end transcription prematurely.
if1
 %out
 %out >>> TARGET: No or invalid program type specified!
 %out >>> Transcription is aborted.
 %out
endif
.err  ;; force error unconditionally
END   ;; mark end of input to stop assembler at once. This is a workaround
      ;; since MASM 5.10 does not abort assembling on .err as expected.
                ENDM


;;-----------------------------------------------------------------------------
;; DESCRIPTION
;;-----------------------------------------------------------------------------
DESCRIPTION     MACRO desrc
                TITLE desrc           ;; control listing file output
;; output description and default values to DEF file
if1
 %out DESCRIPTION desrc
 %out PROTMODE
endif
                ENDM
```

```
71  ;;-----------------------------------------------------------------------
72  ;; STACKSIZE
73  ;;-----------------------------------------------------------------------
74  STACKSIZE       MACRO size
75  ;; output size of default stack to DEF file
76  if1
77   %out STACKSIZE size
78  endif
79                  ENDM
80
81
82  ;;-----------------------------------------------------------------------
83  ;; TARGET
84  ;;-----------------------------------------------------------------------
85  TARGET          MACRO op1
86  purge DESCRIPTION       ;; get rid of statement macros
87  purge STACKSIZE         ;; we no longer need
88  @mode_target = 0        ;; preset internal variable
89
90  ;; check for 80286
91  ifidn <op1>,<80286>
92                  .286
93                  .287
94   @mode_target=286
95   exitm
96  endif
97
98  ;; check for 80386
99  ifidn <op1>,<80386>
100                 .386
101                 .387
102  @mode_target=386
103  exitm
104 endif
105
106 ;; Neither or unknow target proccessor specified.
107 ;; Output error message only once and end transcription prematurely.
108 if1
109  %out
110  %out >>> TARGET: No or invalid target processor specified!
111  %out >>> Transcription is aborted.
112  %out
113 endif
114 .err  ;; force error unconditionally
115 END   ;; mark end of input to stop assembler at once. This is a workaround
116       ;; since MASM 5.10 does not abort assembling on .err as expected.
117                 ENDM
118
119
120 ;;-----------------------------------------------------------------------
121 ;; REQUIRES
122 ;;-----------------------------------------------------------------------
123 REQUIRES        MACRO
124 ;;
125 ;; currently, output to DEF file only
126 ;;
127 if2
128  %out ;--------------------------------------------------------------------------
129  %out IMPORTS
130 endif
131                 ENDM
132
133 ;;-----------------------------------------------------------------------
134 ;; SERVICE
135 ;;-----------------------------------------------------------------------
136 ;; internal procedures
137 $mode_service_output_export    macro lib, name, ordinal
138  ;; output imports statement to DEF file
139  %out name = lib.&ordinal
140 endm
```

```
141 $mode_service_system_resolve   macro name
142  ifidn <name>,<DosExit>              ;; check for DosExit
143   @mode_service_ord = 5
144   exitm          ;; resolved, so exit
145  endif
146  ifidn <name>,<DosExitList>          ;; check for DosExitList
147   @mode_service_ord = 7
148   exitm          ;; resolved, so exit
149  endif
150 endm
151
152 ; implementation
153 SERVICE          MACRO feature, p_op1, class
154 @mode_service_called = 1
155 EXTRN            feature:FAR         ;; define external label
156
157 ;; Define the import directive depending on known class name
158 if2                                  ;; only during second pass
159  irp cls,<.System,.SYSTEM,.system>  ;; check for class, all spellings
160   ifidn <cls>,<class>
161    $mode_service_system_resolve feature
162    $mode_service_output_export DOSCALL1, feature, %@mode_service_ord
163    exitm                             ;; test successful, exit prematurely
164   endif                              ;; from irp cls,<.System,.SYSTEM,.system>
165  endm                                ;; endm of irp
166 endif
167                  ENDM
168
169
170 ;;-------------------------------------------------------------------------
171 ;; INSTANCEDATA
172 ;;-------------------------------------------------------------------------
173 INSTANCEDATA     MACRO
174 ;; final output to DEF file
175 if2
176  %out ;============================================================================
177 endif
178 purge TARGET             ;; get rid of macros no longer needed
179 purge REQUIRES
180 purge SERVICE
181
182 ;; open automatic data segment and set type depending on target
183 ;; processor selected.
184 @mode_instancedataopen = 1
185 DGROUP  GROUP    Data
186 if @mode_target eq 286
187 Data             SEGMENT PARA PUBLIC 'auto'
188 else
189 Data             SEGMENT USE32 PUBLIC 'auto'
190 endif
191                  ENDM
192
193
194 ;;-------------------------------------------------------------------------
195 ;; SHAREDDATA
196 ;;-------------------------------------------------------------------------
197 SHAREDDATA       MACRO
198 ;; close automatic data segment
199 if @mode_instancedataopen
200 Data             ENDS
201  @mode_instancedataopen = 0
202  purge INSTANCEDATA
203 endif
204
205 ;; open default shared segment and set type depending on target
206 ;; processor selected.
207 @mode_shareddataopen = 1
208 if @mode_target eq 286
209 Const            SEGMENT PARA PUBLIC 'shared'
210 else
```

```
211  Const           SEGMENT USE32 PUBLIC 'shared'
212  endif
213                  ENDM
214
215  ;;-------------------------------------------------------------------------
216  ;; FUNCTION
217  ;;-------------------------------------------------------------------------
218  FUNCTION        MACRO
219  ;; if automatic data segment is still open close it since there is no
220  ;; default shared segment that would normally have done
221  if @mode_instancedataopen
222  Data            ENDS
223   @mode_instancedataopen = 0
224   purge INSTANCEDATA
225  endif
226
227  ;; if default shared data segment is still open close it. Do not care
228  ;; about the automatic data segment since it has been closed by the
229  ;; SHAREDDATA
230  if @mode_shareddataopen
231  Const           ENDS
232   @mode_shareddataopen = 0
233   purge SHAREDDATA
234  endif
235
236  ;; open default (static) code segment and set type depending on target CPU
237  if @mode_target eq 286
238  Code            SEGMENT PARA PUBLIC 'code'
239  else
240  Code            SEGMENT USE32 PUBLIC 'code'
241  endif
242                  ASSUME  CS:Code, DS:DGROUP, ES:Const
243                  ENDM
244
245
246  ;;-------------------------------------------------------------------------
247  ;; SANITY
248  ;;-------------------------------------------------------------------------
249  SANITY          MACRO
250  ;; get rid of macros no longer needed
251  purge FUNCTION
252  ;; define initial program entry point
253          @mode_Sanity:
254                  NOP
255                  ENDM
256
257
258  ;;-------------------------------------------------------------------------
259  ;; ENTRY
260  ;;-------------------------------------------------------------------------
261  ENTRY           MACRO
262  ;; get rid of macros no longer needed
263  purge SANITY
264  ;; define actual begin of program
265          @mode_Entry:
266                  NOP
267                  ENDM
268
269
270  ;;-------------------------------------------------------------------------
271  ;; EXIT
272  ;;-------------------------------------------------------------------------
273  EXIT            MACRO
274  ;; get rid of macros no longer needed
275  purge ENTRY
276  ;; open EXIT section and define exit point
277  @mode_EXITopen = 1
278          @mode_Exit:
279                  NOP
280                  ENDM
```

```
281  ;;-----------------------------------------------------------------------
282  ;; TERMINATE
283  ;;-----------------------------------------------------------------------
284  TERMINATE       MACRO p_op1, op1
285  @mode_EXITopen = 0   ;; close EXIT section and terminate process
286                  PUSH    1               ; End process (all threads)
287                  PUSH    op1             ; Set return code to op1 and exit
288                  CALL    DosExit         ;   Through OS/2
289                  ENDM
290
291
292  ;;-----------------------------------------------------------------------
293  ;; ENDOFPROGRAM
294  ;;-----------------------------------------------------------------------
295  ENDOFPROGRAM    MACRO
296  if @mode_EXITopen    ;; if TERMINATE has not been issued, do it now
297   TERMINATE WITH 0
298  endif
299
300  ;; close the default code segment
301  Code            ENDS
302                  END @mode_Sanity
303                  ENDM
304  ;;=======================================================================
```

Listing 3.1: Complete listing of first working edition of the OS2APP.ODE package.

The listing contains several interesting techniques applicable to macro programming with `masm` that merit a closer look. Although transcription context checking was omitted for clarity, there are two places where its simple application is demonstrated, namely in lines 22-57 and 82-117 as part of the implementations of the statements `IS` and `TARGET` respectively. The assembler's `%out` feature is used to output a message on the screen in case of errors and an immediate abortion of the assembly process is enforced by issuing an END statement thus marking the end of input. This error handling section is executed only if control has not prematurely left the macro through one of the `exitm` statements. The `%out` feature also is used to output linker control statements to the module definition file during program text transcription so that both object code and DEF file are created in one cycle. The assembler thus creates the complete input for the linker. Because `masm` is a two-phase assembler, any `%out` is part of an `if1` or `if2` conditional block in order to avoid that messages and DEF data are output twice, during both of the assembler's passes over the source text file.

Some words are in order regarding the simplified implementation of `REQUIRES` and its companion statement `SERVICE`. In this early version of the language package, `REQUIRES` does nothing else but to output an `IMPORTS` control statement to the DEF file so to have the linker prepare the module dependency table during linking, filling it with the entries following `IMPORTS` created by the `SERVICE` statements during transcription. This is the easiest way to use dynamic linking and thus connecting to the system, also. `SERVICE` in turn depends on some knowledge about the modules containing the features to be imported and the features as well, namely the name of the exporting module, the name of the feature, and the ordinal number assigned to it. This approach works for the very first programs we write but it is inflexible in that any new feature requires a revision of the `SERVICE` statement which would grow larger and larger. In respect to the size of the OS/2 kernel API, the statement would turn out to be real a bloat. Since we need to access the APIs of all the many subsystems of OS/2, like `Vio`, `Kbd`, and `Mou` calls and the wealth of *Presentation Manager's* `Win` and `Gpi` features, just to name a

few, the current approach of `SERVICE` would soon turn out impractical if followed too far. Although we can completely replace this scheme in the implementation of `FEATURE` which uses a variation of the aforementioned CDT mechanism for the dynlink libraries and class modules we will create, creating an interface to the system is much more complex. It is of advantage that OS/2 can considered stable so, its APIs will not change which will ease our work. Unfortunately, while the CDT mechanism bases on static feature names which are resolved to dynamic indices, some OS/2 modules only export their features by ordinal numbers, not names, so we need to hardcode the ordinals in the statement implementations, anyway. Fortunately, we can move much of the work out of the statement sphere and can use some compact routines which rely on tables created by the `SERVICE` statements to resolve all references to system modules at runtime in a similar way as our implementation of `FEATURE` will do. Also, we can instruct the later tool `pk` to pick up these tables during packing, remove them from the binary module and instead use them to fill the module's dependency table appropriately in very much the same way the then obsolete linkers do to let the loader resolve the function references, eventually. So far, however, we will use the simplistic approach of the current `SERVICE` implementation.

Another point of interest in this context is the explicit import of a system function which only is used implicitly inside a statement, as in the case of `DosExit` which is encapsulated by `TERMINATE`. Besides `DosExit`, there are a few further system features we will *always* use or sometimes even must use. Most of them also are used only inside other statements and do not appear elsewhere in the program. These features still need be imported, of course. Anyway, we need not do this on our own as we have done now for illustrative purposes. Instead, when `REQUIRES` prepares its tables, these obligatory default imports are handled automatically and added to the head of the tables either for the runtime functions or `pk` to pick up. Besides, invocations of system services as done by `TERMINATE` should *never* appear explicitly in program texts. Instead, new statements will be added which use these features, encapsulate the idiosyncrasies of the underlying API, and which can be used in more general contexts. In addition, these statements are seldom used frequently throughout the program text. This would result in scattering numerous system function invocations about both the high-level program text and, to even greater extent, the transcribed assembly listings. Not only does this make the program sources less readable; all the many `CALLs` into the system start to obfuscate the actual function of the program. It is here where to investigate into alternative methods of accessing the system's features such as special system service states or system proxy objects both of which are means to funnel requests to the operating system to one common location in the program. All these advanced features require a study of the OS/2 system calling conventions. The article in [2] provides a starting point.

3.3 A Simple OS/2 Skeleton Program

Finally, we can put all pieces together to create a self-documenting minimal OS/2 program. It follows the minimalistic program introduced in the article *Principal Structure of OS/2 Programs at the Assembly Language Level*. The program is self-explanatory and shown in Listing 3.2.

```
1  ;============================================================================
2  ; TEST01
3  ;  Input : None
4  ;  Output: 1 - Normal termination
5  ;
6  ; Language definitions:
7                  INCLUDE OS2APP.ODE
8  ;----------------------------------------------------------------------------
9  PROGRAM "TEST01"
10  IS WINDOWABLE
11   DESCRIPTION "OS/2 1.x minimal program #01"
12   STACKSIZE 4096 BYTES
13   TARGET 80286
14
15  REQUIRES
16   SERVICE DosExit       FROM .SYSTEM
17
18  INSTANCEDATA
19   @Buffer        DB      256 DUP(0)
20
21  SHAREDDATA
22   @Info          DW      128 DUP(0)
23
24  FUNCTION
25   SANITY
26    NOP
27
28   ENTRY
29    NOP
30
31   EXIT
32    TERMINATE WITH 1
33 ENDOFPROGRAM "TEST01"
34 ;============================================================================
```

Listing 3.2: Minimal OS/2 1.x program for test purposes in µODE using OS2APP.ODE package.

This program is the very basis we will use and refine throughout the OS/2 development projects. Both, an OS/2 1.x and 2.x module can be made of the same program text, only the parameter passed to `TARGET` need be changed as shown in the next section.

3.3.1 Changing to OS/2 2.x Programs

The `TARGET` statement simplifies making a "32-bit version" out of the "16-bit program". `TARGET` instructs the assembler to use a proper align type, process 32-bit offsets to perform address calculus correctly, and encode 80386 instructions in order to produce 32-bit code segments. The linker, `link386` in this case, does the rest and creates a linear OS/2 executable. To demonstrate, we change the line:

```
SHAREDDATA
 @Info          DW      128 DUP(0)
```

into:

```
SHAREDDATA
 @Info          DW      65535 DUP(0)
```

Clearly, a 128 Kbyte buffer exceeds a 64K segment and the assembler complains appropriately:

```
test1c.ASM(25): error A2050: Value out of range
test1c.ASM(27): error A2102: Segment near (or at) 64K limit
```

To remedy the situation, we simply change the `TARGET` line to:

```
TARGET 80386
```

and re-assemble, then link with `link386` to yield an OS/2 2.x linear executable as `exehdr` reveals in Figure 3.7.

```
Operating System/2 Executable File Header Utility
Version 2.10.000 Feb 02 1993
Copyright (C) IBM Corporation 1988-1992.
Copyright (C) Microsoft Corp. 1988-1992.
All rights reserved.

Module:                          TEST01
Module type:                     Program
Number of memory pages:          00000001 (1)
Initial CS:EIP:                  object 3 offset 00000000
Initial SS:ESP:                  object 1 offset 00001103
Automatic data object:           1
Stack allocation:                00001000 (4096) bytes

 no. virtual  virtual  map      map      flags
     address   size    index    size
0001 00010000 00001103 00000001 00000000 READABLE, WRITEABLE, 32-bit
0002 00020000 0001fffe 00000001 00000000 READABLE, WRITEABLE, 32-bit
0003 00040000 0000000e 00000001 00000001 EXECUTABLE, READABLE, 32-bit
```

Figure 3.7: Output of `exehdr` for the 2.x version of `TEST01`. The structure of the program can clearly be seen. However, the output does not reflect the physical picture of the program in memory. The segments are virtual and mapped into a single linear address space through paging.

The general statement `TARGET` is obligatory and applicable to executable modules and dynlink libraries so the path to create 1.x and 2.x modules of either type is the same. In order to test the program using the simple test driver introduced in the next section, both the 1.x and 2.x version of the program should be created.

3.3.2 A Simple Test Driver

Any program should be tested before it is used together with other programs. Such tests can take on various forms, depending on the type of the program and its function so, we need a flexible way to create test drivers which pass data into our programs and get back both, results and termination codes which are then output in a human readable form and displayed on the screen, send to a printer, or stored in a log file. Test drivers should be made simple, straight, and easy to use. They should test only one module at a time or,

sometimes, several modules of one type. As an example for the latter case, two revisions of the same module can be tested by the driver in sequence so to compare their results at a glance. Anyway, as we stick with the approach of developing small, intercommunicative modules each with a dedicated function, we should do so when it comes to the creation of their test drivers. For simplicity, we use batch files as test drivers as demonstrated in this section.

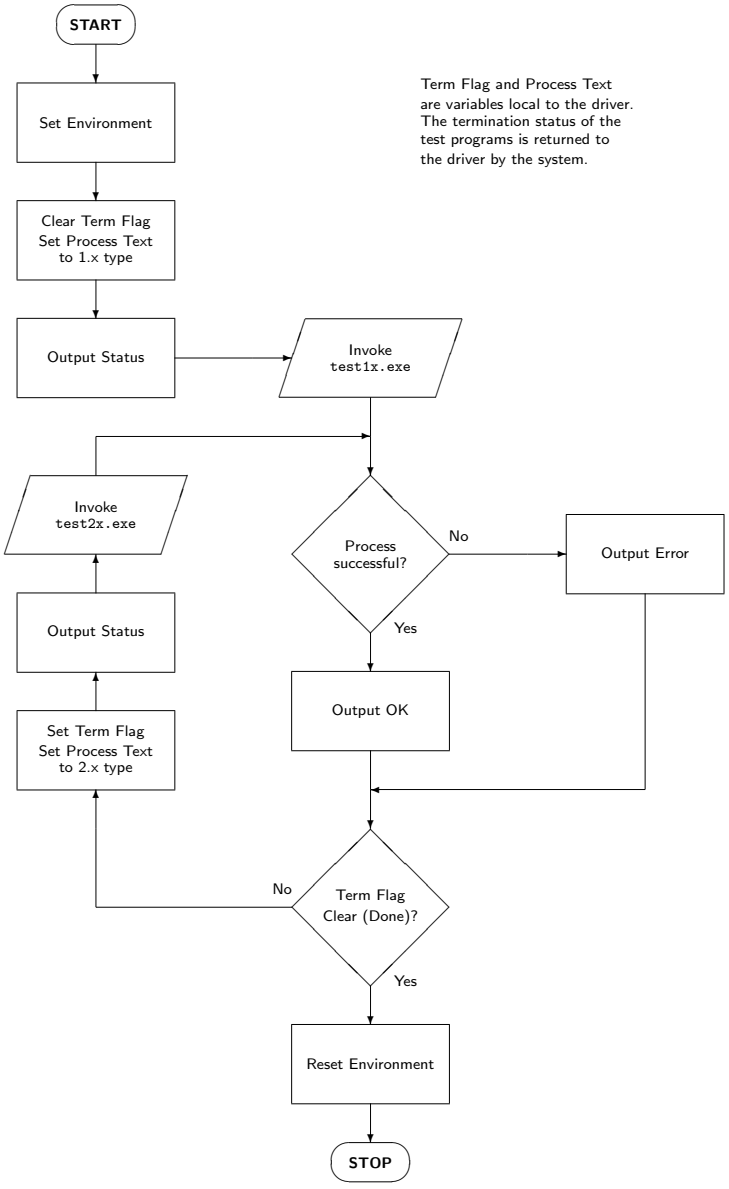

Figure 3.8: Flow chart of the simple test driver for the 1.x and 2.x version of `TEST01`.

Although test drivers should be simple, they need be designed with some grade of thoroughness, anyway. Therefore, we should always make a sketch of the driver to make and document it appropriately. Figure 3.8 depicts a flow chart of the driver. Since we have two versions of the same program to test, it is sufficient to write a single driver program as the one depicted. The implementation of the driver in form of a batch file shows Listing 3.3. It is assumed here that the file name of the 1.x variant of the program is `test1x.exe` and that of the 2.x variant `test2x.exe`.

```
1  @ECHO OFF
2
3  SET COMPLETE=0
4  SET PROC=TEST01 1.x
5  ECHO Now invoking 1.x TEST01...
6  test1x
7  GOTO CHECK_RC
8
9  :INVOKE_NEXT
10 SET PROC=TEST01 2.x
11 SET COMPLETE=1
12 ECHO.
13 ECHO Now invoking 2.x TEST01...
14 test2x
15
16 :CHECK_RC
17 IF ERRORLEVEL == 1 GOTO RC_1
18
19 SET RC=0
20 SET RC_TEXT=Abnormal termination
21 ECHO %PROC% encountered an error, rc = %RC%
22 GOTO EXIT
23
24 :RC_1
25 SET RC=1
26 SET RC_TEXT=Normal operation
27 ECHO %PROC% completed successfully, rc = %RC% (%RC_TEXT%).
28
29 :EXIT
30 IF %COMPLETE% == 0 GOTO INVOKE_NEXT
31 SET RC=
32 SET RT_TEXT=
33 SET PROC=
34 SET COMPLETE=
```

Listing 3.3: A simple test driver in form of a batch file.

The implementation of the test driver shows some methods used frequently in batch file programming like conditional statements, labels, and the use of the environment to define variables used throughout the program. Also, the use of the narrow but robust communication channel between the test driver and the program it tests, namely the `ERRORLEVEL` feature, is shown. In this program, the termination code is passed to the driver from the test program upon completion (through OS/2's `DosExit` function). Later we will develop a protocol that lets us pass a selector to a segment containing more verbose information and even big amounts of data to the test program and from there back to the driver. All this is possible without resorting to more complex tools or advanced programming techniques. In fact, we will see that the test drivers remain pretty simple, however elaborate program intercommunication may become.

The test driver gives us a good opportunity to take a look at one of the most basic building blocks of OS/2 programming, namely processes, as well as their principle in-

terrelationships and the way of elementary interacting supplied by the system. Figure 3.9 demonstrates what happens when we run the test driver on the command line and that much insight into OS/2's process management mechanisms can be derived from this simple runtime scenario, using the primitive programs we have created.

As can be seen in the figure the most basic element of program systems under OS/2 is the process as encapsulated runtime entity and the parent-child relationships into which it enters once it starts other processes, even a replicate of itself. Every process under OS/2 thus has a parent and this way process trees evolve. As separate articles need show, these process trees are more or less structured and organized by default but their level of organization also can be raised thus that tiers of processes evolve which adhere to certain rules and communication schemes and resemble self-maintaining subtrees also called *cells*. Here, we only have two processes, the command line session in which we invoke the test driver (by means of interpretation so no extra processes are involved yet) and the programs it starts, one after the other. Two elementary system services are used, namely `DosExecPgm` and `DosCWait` as explained in the figure. We do not need to worry about these two functions yet since the command line interpreter issues them for us.

For those who insist in writing "Hello world!" programs, Listing 3.4 does the job. It is a simplified variant of the test driver and has the benefit of demonstrating the baseline of conditional processing in batch files as well as the necessity to process any possible output another process returns. Whether or not the informational value of the output this very "Hello world!" program produces makes sense remains questionable, though[9].

```
1  @ECHO OFF
2
3  test1x
4
5  IF ERRORLEVEL == 1 GOTO SUCCESS
6  ECHO An error occured, cannot say hello. ;)
7  GOTO EXIT
8
9  :SUCCESS
10 ECHO Hello world!
11 GOTO EXIT
12
13 :EXIT
```

Listing 3.4: A simplified "Hello world!" program with error checking.

[9] All humour aside, the interpretation of output from one process which is the input to another in the realm of weighing its sense or finding the probability of its correctness is of importance in object-based processing where objects rate the quality of the data they are input both for internal use and distribution of this information to other objects in the system. This makes a software system autonomous in rating any of its parts regarding their output quality, correctness, ability of processing, and so forth. How this works in practice is not difficult to imagine. In the sample, the driver only had to substitute the `ECHO` directives for some instructions to store *its interpretation* of the data returned by `test1x`, for example it could increment a counter for any good answer and decrement it for any bad one. Given the driver calls different versions of programs for the same purpose it can, over time, then report on the output quality of any such program. This is like a "feeling" which can be communicated to other drivers which are likely to use the same programs to evaluate the robustness or trustworthiness of these external processes. This gives birth to auto-adjusting systems and autonomous error-handling. An interesting concept that has proved valuable a tool in practice already and can be explored with the simulation of object systems on OS/2 very well. Refer to [8] for more information about distributed computing and error weighing.

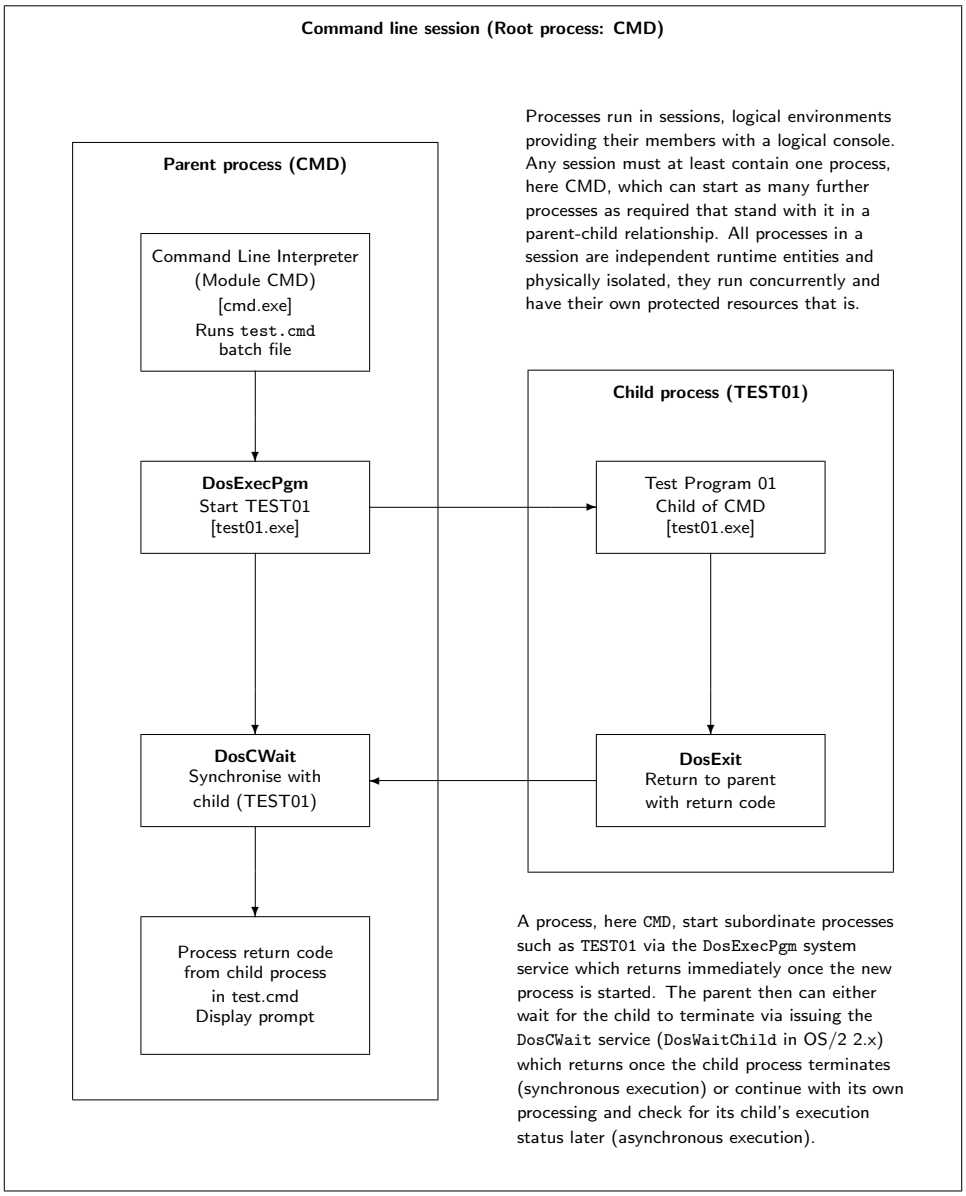

Figure 3.9: Runtime trail of the `TEST01` program execution. This simple example depicts the bottom line of the OS/2 application programming paradigm which centres on processes, the parent-child relationships between them, and their logical grouping to sessions or screen groups respectively. Threads belong to a process' internals only, come next in line, and are not even necessary to include in a module's design. Applied correctly, they are indeed useful tools but should be used judiciously.

Again, one should note how this simple program demonstrates the very nature of true OS/2 applications, namely:

1. The dissolving of monolithic programs by conceptually dividing tasks into functional blocks.

2. The implementation of these conceptual blocks in form of several isolated runtime entities called processes.

3. The establishment of communication schemes to let these processes cooperate in order to fill the required function.

Outputting a human-readable message is not part of the innermost function of the application which shall demonstrate the structure of a most simple OS/2 program and the runtime trail the proper execution and termination of this program creates. Thus, `test1x` implements this function and leaves it to another entity to interpret the outcome of its operation. This is done by the driver program `hello` which calls `test1x`. We leave it to `hello` how to interpret the data returned by the worker process. This also makes this process independent from such highly system and application dependent issues like human interfaces, consoles, user environments, etc. Although clearly overkill for this sort of sample program, it is good to learn this kind of programming right from the start. It leads to modularity, reusablitity, and robustness of programs which are invaluable design goals to attain in any kind of software engineering project.

We will stay in using batch files for test drivers. They are easy to write, are interpreted so they can be changed quickly when necessary, and, most importantly, the system supplied communication channels between the processes we start from the drivers are already existent and reliable. Here we used environment variables and the `ERRORLEVEL` feature for this purpose. Later we can employ redirection techniques for more sophisticated data transfers. Once a module has been tested using a batch driver, it can be integrated into more complex systems. Keeping with the system's conventions, this integration is a seamless, hassle-free process. Besides, we will draw from the experiences made with the batch drivers in creating our EDC (Error Detection and Correction) and DFM (Data Flow Monitoring) tools.

It has proved good practice to put the test driver—or drivers respectively, each special test case should be treated separately—in the same directory as the module under development, assign them names in the form `test_<purpose>.cmd`, keep them *all* even after finishing the project and create a *new* driver for *any* later refined edition of the module.

3.4 Release Notes

This article introduced into the utilization of a simple yet comfortable high-level language to create a minimalistic skeleton program for use in further studies. A first working implementation of a language extension package was shown with which self-documenting executable program modules can be written. The development of simple test drivers also was demonstrated which pointed to elementary concepts of OS/2 as sophisticated multiprocess runtime environment.

An easy way to produce either type of OS/2 modules, 1.x or 2.x, with minimal effort in a simple but working development environment, as well as to test their function, thus is now available.

References

[1] *A Simple Skeleton Dynlink Library for Test Porposes* in: OS/2 Lab Notes, Issue 2. cefischer. ISBN-13 978-3-944037-51-6.

[2] *OS/2 API Calling Conventions* in: OS/2 Lab Notes, Issue 2. cefischer. ISBN-13 978-3-944037-51-6.

[3] *A Simple Skeleton Device Driver for Test Porposes* in: OS/2 Lab Notes, Issue 3. cefischer. ISBN-13 978-3-944037-52-3.

[4] *Use of Natural Language Elements in Programming with µODE* in: Lab Notes, Issue 1. cefischer. ISBN-13 978-3-944037-40-0.

[5] *Realizing Control Structures in Two-phase Program Source Text Transcriptions (1): Implementing a High-level Language Iterative Statement* in: Lab Notes, Issue 1. cefischer. ISBN-13 978-3-944037-40-0.

[6] *Realizing Control Structures in Two-phase Program Source Text Transcriptions (2): Implementing a High-level Language Iterative Statement* in: Lab Notes, Issue 2. cefischer. ISBN-13 978-3-944037-41-7.

[7] Fischer, Carla. *The µODE Language Specification and Programming Guide.* cefischer. ISBN-13 978-3-944037-38-7.

[8] Fischer, Carla. *The µPMOS Primer.* cefischer. ISBN-13 978-3-944037-83-7.

[9] Fernandez, Judi N. and Ashley, Ruth. *Assembly Language Programming for the 80386.* McGraw-Hill. ISBN-10 0-07-020575-2. 1990.