May 1995

OS-9 International

Vol. 4, Issue 2

EFFO OS9

European Forum For OS-9

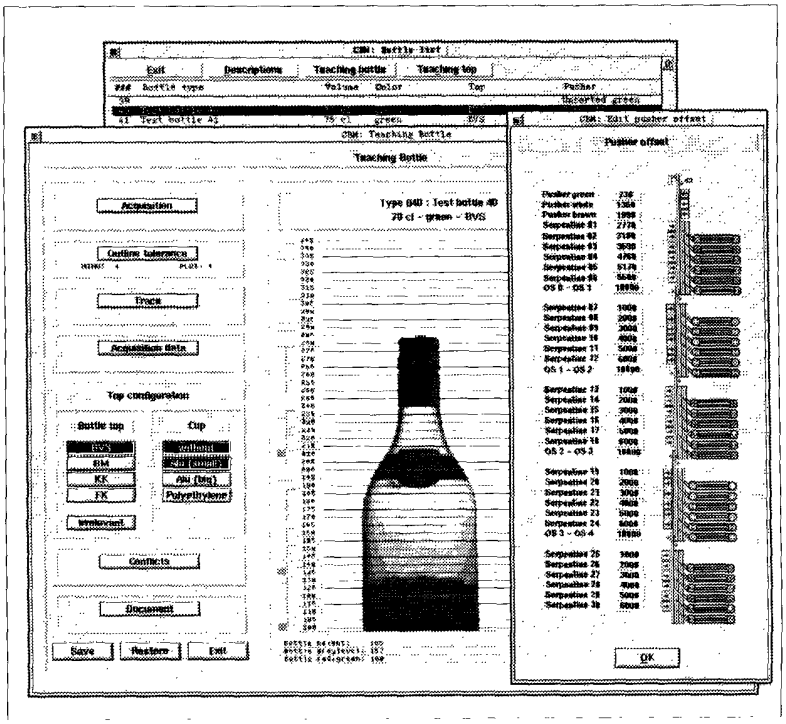
8606 Greifensee, Switzerland
Fax +41 1 940 38 90
email os9int@effo.ch

sFr. 10.00
ISSN: 1019-6714

SYSTEM-PAK I/MGR

The window system for OS-9/68xxx, LynxOS and VxWorks:

Plant Control
Image Processing
Data Presentation
Software Development



Example dialog from the teaching phase of an automatic high-speed bottle sorter. The graphical user interface is running under the MGR window manager and has been designed using the MGR/ALib application library.

reccoware systems

New Address: *reccoware systems*, Wolfgang Ocker, Rapperzell, Föhrenstraße 8, D-86576 Schiltberg, Phone ++49 (0) 82 59 10 48, Fax 10 49, Email: reccoware@recco.de

Reprinted with kind permission by CHESTONAG Automation AG, CH-5706 Boniswil

startup

This issue of OS-9 International is dedicated to word processing. At least, about half a year ago we had the idea that it would. But reality is different.

At first sight, word processing under OS-9 appears to be an anachronism. In these days, when the appearance of a printed article is considered more important than its content, any fixed-font output is certainly obsolete. The minimum configuration for creating a printed document consists of a Macintosh PC so that WYSIWYG. Producing the journal OS-9 International this way has made two things very clear to us: i) you never see what you get, and ii) normally, you are happy that you don't get what you see.

What can be done on a typical OS-9 system? First, we need an editor. Unless you decided to never give up your favourite one, we recommend Daniel M. Lawrence's µEmacs. It is available on virtually any computer system and lets you type a text much faster than with an over-featured and mouse-controlled word processor.

Second, we need something to format the output. Donald E. Knuth's type setting system TEX is the formatting program of choice. If a scientific article has to be published, TEX is often the only accepted format, because the publisher denies the existence of an alternative. If TEX is good enough for scientific articles, why should it not be suitable for other purposes?

Last but not least, a printed output must be generated. Almost every desktop publishing program is able to produce a PostScript file, and also TEX may do so via the *dvips* utility or similar. This does not represent a problem if you own a PostScript printer or your document is intended to be produced by a printing-office. If, however, this is not the case, L. Peter Deutsch's *Ghostscript* interpreter fills the gap. It generates an appropriate printout on nearly every available printer including 9-pin zwieback milling machines.

As you can see, only three high-level PD program packages are required for word processing under OS-9. So far our dreams – unfortunately, two of the packages mentioned did not reach the required quality to be released, although various versions are available via network and on mailboxes. Actually, the effect that several ports exist, represents the problem: each of them comes with specific nice features but none includes them all. On the other hand, the main problem is not the lack of a feature, but the lack of the required stability. This is an Example For Failed Organisation!

Porting such large packages requires a different strategy than hacking a tiny little utility. The first goal should be to convince the authors of the original program to include support for OS-9 in the official release in order to make future upgrades reasonably painless and to reduce the number of ports. As a prerequisite, all major Unix development tools are now available under OS-9: in addition to the already existing GNU C/C++ compiler, Bourne shell and Unix C library, the appropriate GNU make has now been released as can be read in this issue. In consequence, there is some hope that the above mentioned tools will be released soon.

Werner Stehling

PCMCIA Cards and Non-PC Operating Systems

Lukas Zeller and Axel Berghoff

Introduction

Memory and I/O devices based on an interface standardised by the Personal Computer Memory Card International Association (PCMCIA) are widely available at competitive prices. Unfortunately, they cannot easily be used on computers with a non-PC operating system such as an embedded OS-9 computer. The aim of the current article is to analyse the existing difficulties and to propose a procedure to overcome them.

History

The principle of memory cards has been invented by the Frenchman Roland Moreno in 1974. Such memory cards can be classified into cards with contacts and contactless ones and have been designed to fulfil many purposes such as data transfer, memory extension, personal identification, telephone cards etc. Pin layout and electrical characteristics in cards with contacts and data transfer methods in contactless cards were mainly defined by the particular manufacturer. Only in 1990, the pin layout and electrical characteristics of memory cards with contacts were standardised by the Japan Electronics Industry Development Association (JEIDA). Two types are primarily used, the 68-pin JEIDA-4 and the 88-pin JEIDA-5 standard. JEIDA-4 cards have an 8- or 16-bit wide data bus, their memory capacity ranges from 64 kByte to 16 MByte, but the number of address lines allows for a maximum of 64 MByte. JEIDA-5 cards are available with 32- or 36-bit wide data bus so that, in principle, memory cards can be designed being large enough to replace mass storage devices.

In view of the requirements of the rapidly growing market for portable computers, the importance of memory cards was recognized, and the PCMCIA was founded in 1990. Its main goal is the creation of an appropriate standard for a card-size interface comparable to the memory cards with the additional ability to connect mass storage devices such as hard disks via this interface. This standard was called PCMCIA-1.0 and was based on the JEIDA-4 memory cards. Furthermore, it was the aim to define a multi-purpose interface that uses the same pin layout and allows to connect many other types of I/O devices to a portable computer, e.g. a serial controller for modem and an Ethernet network interface. Therefore, the initial PCMCIA 1.0 standard was supplemented with functions such as "I/O Read", "I/O Write" etc. and published

as PCMCIA 2.0. This enhancement was possible, since some of the 68 pins of the JEIDA-4 standard have been left unassigned so that the above functions could be realised without extending the connector.

Today, the PCMCIA standard has become very successful, and card-size devices are available for many purposes at a very competitive price. It is, therefore, highly desirable to use such devices also with computers that work under operating systems other than the one used on the normal PC. These computers must not necessarily be portable, but memory cards can ideally be used for software upgrades in embedded systems. Furthermore, card-size I/O devices offer the possibility to temporarily connect such systems to phone lines, networks and mass storage devices. Several VMEbus manufacturers have already presented products equipped with a PCMCIA interface. A prominent example is the BAB-40 CPU board (Eltec, Mainz, Germany), that relies completely on PCMCIA technology; it has one internally and two externally accessible PCMCIA slots.

Technical Background

The electrical specifications and especially the pinout of the JEIDA memory cards were, obviously, designed to be compatible with memory chips approved by the Joint Electron Design Engineering Council (JEDEC). Since PCMCIA, however, was primarily interested in creating a standard for IBM compatible PCs, it is obvious that they tried to make the PCMCIA 2.0 interface as compatible to the ISA bus as possible. JEIDA-4 already defined a mechanism to identify a memory card. This technique is called Card Identification Structure (CIS) and, basically, represents a linked list of small records of information stored under a well-defined ID. There are records defining size, speed, JEDEC ID, manufacturer etc. This identification strategy was incorporated into the PCMCIA 2.0 standard; it required, however, the definition of additional records, e.g. for I/O-purposes.

Drivers for Non-PC Operating Systems

In principle, the PCMCIA interface technique can be used under any operating system. Therefore, its implementation on operating systems such as OS-9 should, generally, not differ from implementing any other interface technique by writing an appropriate driver. The only prerequisite, however, is that the device fully conforms to the standard and the manufacturer publishes the complete interface specifications. Different approaches are required for i) transparent memory cards, ii) standardised mass storage devices, iii) standardised modems and iv) any other I/O devices using proprietary interface definitions.

Transparent Memory Cards

Transparent memory cards (static random-access memory, read-only memory) can be used without major difficulties. It is either possible to make the addressable memory region visible to the operating system at boot time, or to use an already available RAM-disk driver. Under OS-9, the standard *ram* driver can be used for this purpose, if the descriptor contains the RAM card's absolute start address (long word) at offset *M$Port* (0x30).

Standardised Mass Storage Devices

Since PCMCIA 2.0 already has ISA-like properties, it was possible to re-use another PC standard, the AT bus. Under the name PC-AT attachment (ATA) this technique dating from the early eighties was resuscitated by integrating it almost unmodified into the PCMCIA standard. The fact that not a state-of-the-art standard was defined but an outdated technique was re-used is the reason for many problems encountered when dealing with PCMCIA devices – especially in a non-PC environment. There is also an Auto Indexing Mass Storage (AIMS) standard as part of PCMCIA, which was designed as a newer alternative to ATA, but it is used very rarely.

Standardised Modems

The interface between the CPU and a modem is relatively simple due to its single, serial channel with moderate speed as compared to high-volume data links such as network or graphics. Therefore, one of the early I/O applications of PCMCIA devices were card-size modems for use not only in notebooks but also in subnotebooks, palmtops and Personal Digital Assistants (PDAs). The use of PDAs is remarkable insofar as they contain other processors than members of the 80x86 family. This was possible since the modem interface is documented as part of the standard. Recently, a generic driver for a PCMCIA modem connected to a non-PC notebook was released (MC680x0, series 500 PowerBook, Apple).

Other I/O Devices

Other type of PCMCIA devices such as graphic controllers, network interfaces, A/D converters etc., follow the standardisation only in the way the memory is addressed, but are just small ISA/EISA cards in all other aspects. Therefore, they require specific drivers for either not or insufficiently documented controllers. In the PC world, an installation disk normally accompanies such a device that allows using it on a standard PC. Since OS-9 or any other non-PC operating system is completely unable to execute such installation procedures or programs, these devices cannot be used. But even if drivers were available or the individual controllers were well documented, other restrictions may apply: industrial computer systems have differ-

ent requirements with respect to product life-time, warranty etc. than standard office automation systems. Finally, mechanical properties and thermal characteristics of most of the initially used PCMCIA devices may not be acceptable in the industrial environment.

PCMCIA 3.0

The newly 1995 released PCMCIA 3.0 supplements this standard in two ways. Firstly, the remaining not yet implemented ISA/EISA signals were added to the PCMCIA connector. Secondly and most importantly, a PCI-derived standard called CardBus was integrated to PCMCIA. Among others, this standard enables the host to determine specific device properties, e.g. whether a card supports 5V or 3.3V PCMCIA, or CardBus (always 3.3V). This new standard is so important because, for the first time, a bus is supported not being dedicated exclusively to 80x86-based PCs. Finally, since People Can't Memorise Computer Industry's Acronyms, the name PCMCIA is replaced by "PC Card standard" from version 3.0 onwards.

What Is Needed to Make PCMCIA Suitable for Industrial Applications?

Up to now, PCMCIA devices are mainly used in 80x86-based portable computers and office automation systems. This market segment needs large quantities, low cost but has few requirements with respect to industrial quality, extended temperature range, longevity and documentation. It may not be easy for PCMCIA manufacturers to enter such a different market. On the other hand, PCMCIA devices offer a wide variety of functionality that is not easily achievable using other components. For example, CPU boards may be extended with various memory circuits and hard disks in a flexible way, data transfer between different computer systems is highly facilitated and I/O interfaces may be connected temporarily to embedded systems. In order to be acceptable to industrial customers, manufacturers of PCMCIA devices must provide

- confirmed product availability of minimally three years, better five years, after first release,
- stability of the electrical properties during the entire production cycle,
- unrestricted access to all relevant technical data, at least for system developers,
- extended warranty period of up to five years,
- optionally extended temperature range.

The higher price of products with such enhanced quality most probably will be accepted by industrial customers. Indeed, there is already an emerging segment of industrial PCMCIA applications and cards that fulfil at least some of the above requirements. PCMCIA manufacturers may have realised that their products cannot compete with their non-miniaturised counter-

parts in cases where mechanical dimensions are not an issue. For instance, data transfer using a floppy disk in office automation systems is certainly less expensive and probably not worse than using a PCMCIA SRAM card.

Request for Comments

It is well conceivable that embedded OS-9 systems would profit from the availability of an open PCMCIA market. Although a certain interest of PCMCIA manufacturers in industrial customers can be noticed, much more activity is needed to bring these two worlds together. In order to determine the interest in such an activity, OS-9 International has installed the email address <pcmcia@effo.ch>. Comments, proposals and any other kind of contributions are welcome.

Lukas Zeller developed PCMCIA interfaces and software for a Swiss company. He can be reached via email at <luz@zep.ch>.

Axel Berghoff works, among others, as consultant in the VMEbus market. His email address is <aberghoff@aberg.pfm-mainz.de>.

Software + Hardware + Know-how + Service ...

No matter if you are interested in CPUs, graphics, image processing or system configurations:
ELTEC offers high-quality products and services providing industry suitable solutions for complex problems in process automation.

Modular flexibility from low-cost to high-end offers, for example, the **EUROCOM°17** board:
- 1 or 2 MC68(EC)040 CPUs upgradable to 2 MC68060 CPUs
- DRAM
- optional SVGA graphics (4 bit overlay, 1152 x 900 pixel, 256 out of 2^{24} colors)
- optional network
- SCSI-2
- 4 serial and 2 parallel interfaces
- LEB for IPIN mezzanine cards

The IPINs Intelligent Serial Interface Controller (IPIN 1700) and flexible Camera Interface (IPIN 1900) open up the fields of
- telecommunication and
- image processing

Especially for the fields of industrial I/O and control ELTEC offers a modified **EUROCOM°17** board as carrier for
- MODULbus and
- M-Module

mezzanine cards.

Special software modules offer a transparent use of 2 CPUs under OS-9 with MGR and other operating systems.

elektronik mainz

ELTEC Elektronik GmbH · P.O.Box 421363 · D-55071 Mainz
Phone ++ 49 (6131) 918-0 · Fax ++ 49 (6131) 918-198

or our distributor in Switzerland:
SPECTRALAB · Brunnenmoosstraße 7 · CH-8802 Kilchberg
Phone ++ 41 (1) 715 38 07 · Fax ++ 41 (1) 715 54 47

... the Winning Development-Platform Under OS-9!

OS-9 Meets PLC

Hans Wiedemann

Introduction

Plant control systems are normally realised using Programmable Logic Controls (PLC) that do not offer the development support and real-time capabilities inherent in the OS-9 operating system. These and other advantages make OS-9, in principle, ideally suitable for plant control systems but, for the time being, only few systems have been based on OS-9. This article explains the differences between PLCs and OS-9 systems and presents a solution to bring them together.

OS-9 and its Normal Environment

A VMEbus CPU board powered by a Motorola MC68xxx processor is still the standard development platform of the OS-9 operating system. Such systems are normally equipped with a SCSI interface, serial ports for terminal and other serial connections, an oscillator for the system tick, a real-time clock (RTC), a built-in graphic controller, a network connection (e.g. Ethernet) and, optionally, digital or analogue I/O. Cross development systems running on Unix workstations (Unibridge) and PCs (PC Bridge) can be used, too. More recently, FasTrak that has a graphical user interface gained wider acceptance. FasTrak is available on a variety of Unix platforms and even on PCs running MS-Windows. Although this wide range of development environments is available, target systems, in principle, do not differ very much from each other nor from the classical MC68xxx-based OS-9 development system as described above. This makes it, for example, possible to realise a graphical user interface for purposes such as process visualisation even on a target system. The increased complexity of such systems does not necessarily represent a problem for programming and debugging, since one of the advantages of the OS-9 operating system is that all host debugging tools can be made available on the target as well. Using the described OS-9 target hardware in plant control systems, however, has two important disadvantages: firstly, it has a relatively high price and, secondly, the normal PLC programmer may find OS-9 not terribly easy to conquer.

The Classical PLC

Programmable Logic Controls are programmed using a special technique in industrial automation technology. Basically, a number of logical input states are combined using Boolean algebra to form a logical output state. Whilst first generation PLC systems used a relay-based technique for this purpose, the state-of-the-art hardware is, of course, based on microprocessor technology. Despite this evolution, PLC programming is still done using an assembly-like syntax that mainly consists of logical and integer operations; bare floating point operations are tough to deal with. On the other hand, today's ergonomics and safety requirements in plant control systems are usually satisfied with technologies such as computer graphics and networks. Such technologies can no longer be realised with classical PLC. Its functional elements, however, are still needed, since PLC has a long and successful tradition and is supported by many leading companies having accumulated a large body of PLC-specific know-how. Traditionally, the installation costs of a PLC-based system are calculated as a multiple of the number of atomic PLC points; a small and apparently unimportant increase in the price of a single point may, therefore, result in an important cost factor for the entire system. In consequence, it is only possible to introduce a new PLC concept, if the costs per point do not exceed the currently accepted range. This applies even to systems having enhanced capabilities such as a graphical human/machine interface and network connectivity.

The Best of Both Worlds

The idea was, therefore, born to integrate both the power of OS-9 and the tradition of PLC into a common concept. Such a system, called Smart I/O, consists of a low-cost MC68302-based CPU without VMEbus interface, the processor's inherent serial interface capabilities and a DC/DC converter — everything integrated in an appropriate industrial housing. Sockets for EPROM and/or Flash EPROM, DRAM, SRAM, and serial EEPROM are available. I/O functionality may be extended via the MC68302's serial communication port [1]. This port allows to exchange status and control information with a variety of serial devices, using a subset of the Motorola serial peripheral interface (SPI).

Programming the Smart I/O Board

Standard OS-9 Programming

Programming can either be done on an OS-9 VMEbus system or on one of the above named cross development systems. Since Smart I/O computers are equipped with a nearly complete Extended OS-9 system, and appropriate drivers are available for all I/O modules, virtually any existing software can be used provided that it does not require special hardware. This software strategy ensures protection of software investments and a high level of compatibility from high-performance MC68060 based VMEbus CPU boards down to Smart I/O systems.

PLC Programming

The second method is the newly created link from OS-9 to the PLC world. Driven by unsatisfied customers, the IEC committee defined a programming standard (IEC 1131-3) that includes sequential function charts, function block diagrams, ladder diagrams, instruction lists and structured text. All of them can be combined without restriction to form a specific application. There is even a way to call standard routines written in ANSI-C language (see below). A full implementation of the IEC 1131-3 standard is available as a commercial product (ISaGRAF) from CJ International (Grenoble, France). When used in conjunction with OS-9, the ISaGRAF kernel is executed in the same way as any other user task and takes control over the downloaded PLC application. The development platform for ISaGRAF is a PC under MS-Windows.

Mixed Programming

Actually, the simultaneous availability of OS-9 and PLC represents the important innovation of Smart I/O: a PLC programmer can program it using the traditional way without being confronted with languages and techniques he never wanted to know. At the same time, the OS-9 specialist can program it using C language and take advantage of his beloved operating system without being confronted with programming strategies he thinks have been abandoned long time ago.

The Communication

An important aspect of PLC systems has not yet been discussed: communication of several PLCs between each other and also to supervising process control systems. On a standard VMEbus OS-9 system, this would probably be done using Ethernet network communication. Price re-

striction and tradition of PLC systems, however, dictate that a traditional serial communication be used. Such serial communication would be a field bus, the only question is which one to use. Data of the most popular field buses are given in the following table:

	Profibus	**FIP**	**Bitbus**	**CAN**	**Interbus-S**
Standard	DIN 19245	UTE-C46-6xx	IEEE 1118	ISO/DIS 11898 CIA/DS201-205,207	DIN 9258
Access procedure	Multi-master Master/slave	Producer/ distributor/ consumer	Master/Slave	Multi-master	Master/slave
Medium	twisted pair, fiber optic	twisted pair, fiber optic	twisted pair + optical clock	twisted pair, fiber optic	twisted pair (ring)
Transfer rate	9.6 - 500 kB/s (1.5 MB/s)	31.25 kB/s, 1 MB/s, 2.5 MB/s, 5 MB/s (FO)	62.5 kB/s - 2.4 MB/s	up to 1 MB/s	300 kB/s (500 kB/s)
Hardware support	68302 & 68360 µcode, 8051, V25, SPC, PBS01	FIPART, FIPIU, FULLFIP	80C152, 8044	ICAN 82526/527, BCAN 80C200, MC68HC705X4/16, NEC µPD72005	SuPI, MA1
Max. data per frame	246 Bytes (32/246 Bytes)	128/256 Bytes	250 Bytes	8 Bytes	512 Bytes
Overhead per frame	9 Bytes	6/12 Bytes	13 Bytes	6 Bytes	6 Bytes
Partici-pants	127 nodes	65536 objects/ 16.7 million msgs	sync. 28 nodes, self-clocked 250 nodes	2032 idendifiers	256 nodes

With respect to Smart I/O, the important advantage of the Profibus is that the bus protocol (OSI level 2) is nearly completely implemented in the micro code of the MC68302 and MC68360 processors. In addition, Profibus is market leader in Europe and gains a lot of popularity world-wide. Therefore, it was decided to equip all Smart I/O modules with Profibus firmware (OSI level 7) by default utilising the already existing hardware. Support for other buses must be ordered separately.

Since the Extended OS-9 run-time license includes TCP/IP, it allows to go even one step further, i.e. to realise transparent network communication. Already implemented is the Serial Line Internet Protocol (SLIP) that runs via the RS232 interface and, additionally, the implementation of TCP/IP on top of Profibus is currently under development. This will open Smart I/O for all socket-based applications including even NFS.

The Link between ISaGRAF and C

There are three ways to write specific user functions in C language and to connect them to ISaGRAF:

Enhancing the ISaGRAF Kernel

Additional user functions can be developed in C language, compiled on the PC using PCBridge or FasTrak and linked to the ISaGRAF kernel. This newly created kernel is then downloaded to the Smart I/O module. The disadvantages of this method are that the additional user functions cannot be accessed from outside the ISaGRAF kernel and that a new kernel has to be created each time a new function has been added.

Trap Handler

Another method is to use a trap handler that may be created automatically using an already existing procedure on the development system. The functionality of this trap handler, if downloaded in addition to the standard ISaGRAF kernel, may then be used by the ISaGRAF kernel and also by other tasks.

Independently Running Tasks

The required functions can be realised in an independent task that concurrently runs with the ISaGRAF kernel. Common access to global data is best achieved using an OS-9 data module. The OS-9 task uses the standard C functions to link to the data module; ISaGRAF provides an equivalent functionality using standard communication commands. It is even possible to let the ISaGRAF kernel access RAM at absolute addresses. If the latter method is used, such memory regions should be made inaccessible to the OS-9 kernel, for instance, by excluding them from the memory list in the init module.

Conclusion

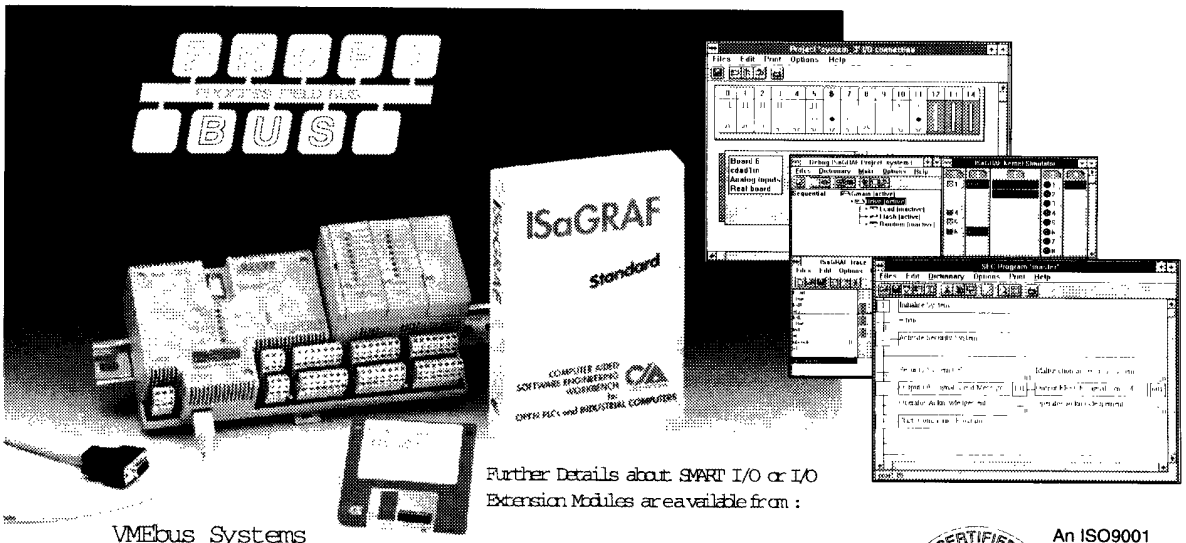
Up to now, OS-9 based VMEbus computers and PLC controllers lived in two different worlds. The realisation of Smart I/O modules as presented herein shows that the basic concept of the OS-9 operating system is flexible enough to integrate even a PLC kernel. This allows for the first time to design homogeneous plant control systems that are based on the same operating system from low-level logic control up to high-level system supervision being managed via graphical user interface and providing network connectivity.

Reference

[1] Motorola (1991) *MC68302 Integrated Multiprotocol Processor User's Manual*, edition 2, Motorola Ltd., European Literature Center, 88 Tanners Drive, Blakelands, Milton Keynes, MK14 5BP, England.

Hans Wiedemann works in business marketing at PEP Modular Computers GmbH. His main interest centres on real-time operating systems, field buses and graphics software. He can be reached via email at <HANSWI@pep-kaufbeuren.de>.

SMART I/O Starter Kit
Micro-PLC + Computer + IEC1131-3 + PROFIBUS



VMEbus Systems
Software Compatible

Further Details about SMART I/O or I/O Extension Modules are available from :

PEP Modular Computers GmbH,
Apfelstrangerstr. 16
87600 Kaufbeuren
Germany.

CERTIFIED DIN ISO 9001 QUALITY SYSTEM
An ISO9001 Certified Company

Modular Computers®

Memory Modules

Werner Stehling

Introduction

Memory modules represent a main design aspect of the OS-9 operating system. They are especially important for real-time behaviour, because the required data and code can be loaded into memory before programs are started; this reduces unpredictable timing due to access of mass storage devices. In addition, the memory module concept greatly facilitates the creation of embedded ROM-resident systems.

Typically, a user program resides in a program module, but it can also be stored in a subroutine module or in a user trap library. Data may not only reside in static variables but also in data modules. These module types are less known, although they probably merit more consideration. It is, therefore, the aim of this article to give an overview about the module concept and to present data modules, subroutine modules and user trap libraries in more detail.

OS-9 Module Concept

Up to now, OS-9 defines ten different types of memory modules. They can be classified into three main groups:

Module group	Module type	M$Type	Description	Examples
System-state program modules	Systm	12	System module	kernel
	Flmgr	13	File manager	rbf, scf, sbf
	Drivr	14	Device driver	rbteac, scscc, sbviper
User-state program modules	Prgm	1	Program	user programs
	Sbrtn	2	Subroutine module	tcp, udp
	Multi	3	Multi-module	reserved for future use
	TrapLib	11	User trap library	cio, csl
Data - containing modules	Data	4	Data module	inetdb
	CSDData	5	Configuration Status Descriptor	not standard
	Devic	15	Device descriptor	h0, t1, mt0

All modules have the same structure: a 24-word common module header including a 1-word header parity is followed by a module specific header extension of up to 8 long words, the

module body and a CRC checksum value. Data-containing modules normally do not require a header extension.

Module Integrity Checks

The first two bytes of a valid OS-9 module are the so-called Sync Bytes ($4AFC). Validity can further be checked by XOR-ing together all 16-Bit words of the header which must result in $FFFF. A third check considers the CRC checksum value that is calculated over the entire module. Practically, the CRC value is only tested when a module is loaded into memory, but the header parity is checked whenever a process links to the module. This makes it possible to modify the module body of a memory resident module but any change to the module header will result in error 236 (*E$BMHP*). Attempting to load a module with an invalid CRC value leads to error 232 (*E$BMCRC*) but, from OS-9 version 3.0 onwards, this does not apply to data modules anymore.

Data, Subroutine and Trap Library Modules

While the functions of a program or a device driver module are rather obvious, the tasks of data, subroutine and trap library modules are defined less strictly. These latter three module types are supported by the same kernel functions *F$Load*, *F$Link*, *F$SetCRC*, *F$Unlink* and *F$Unload* but are, however, treated differently when produced by the *l68* system linker.

Data Modules

Data modules are frequently used to exchange data between processes. These processes must not necessarily be user-state programs, but also drivers and file managers can link to a data module. Data exchange is best done by globally defining a data structure, the pointer of which is assigned to the entry of the data module.

A common problem centres on the synchronisation of write and read accesses, since only one program may be permitted to write at a time, and reading must be inhibited while data are invalid. This problem is frequently solved using events but, in principle, any method of inter-process communication can be used. Another approach is to declare an access flag in the data module itself. An appropriate instruction must be used to test and set this flag, because this action must be indivisible. It is recommended, whenever possible, to have only one writing process and one or more reading processes, since this makes synchronisation easier.

Subroutine Modules

Historically, the subroutine module was probably invented in order to provide an adequate module structure for good old Basic09's I-code. Of course, it is not limited to this special application. In contrast to data modules, a subroutine module is intended to contain executable code, and the linker takes care of referencing global data relative to the global data pointer *a6*. It also sets the *M$Mem* field in the header extension appropriately.

User trap libraries

User trap libraries are more complex than data and subroutine modules. A specific kernel command *F$TLink* is available for their installation. Furthermore, the linker not only sets the total amount of global data storage, but also prepares lists at offsets *M$IData* and *M$IRefs*. These lists are required to allow for position-independent code.

Trap handlers are relatively common in OS-9 systems: the C I/O library (*cio*), the C shared library (*csl*), the mathematical function library (*math*) and, of course, the kernel itself. Nevertheless, a trap handler is often a synonym for complication, unexpected results and a technique that is difficult to manage. This represents an interesting psychological phenomenon that is probably due to two facts. First, the name "trap", in general, does not suggest confidence; secondly, traps are managed in the same way as all other exceptions, namely interrupts, bus errors, address errors, divide by zero etc. Indeed, this close relation to all those bad things that can happen to an OS-9 system lets traps appear even less attractive. There is, however, nothing dangerous with trap handlers, as they represent a safe and reliable method to provide library functions that reside only once in memory but are accessible to more than one process.

Summary

Under normal conditions, an OS-9 programmer does not need to worry about user trap libraries or subroutine modules. A simple command line option of the C front end instructs the linker whether to produce a program that uses the C trap library or one that does not. Subroutine modules are automatically created by Basic09 but are not supported officially for other purposes. However, OS-9 is flexible and transparent enough to allow for using these modules and also data modules in a very specific way to fulfil the needs of given system requirements.

Werner Stehling works as hardware and software engineer in the Radio Astronomy Group of the Swiss Federal Institute of Technology. He can be reached at <stehling@effo.ch>.

Shared Libraries Using Subroutine Modules

Carsten Emde

Introduction

Irrespective of how often an OS-9 program is running concurrently on the same CPU, there is only one copy of the program present in memory. This behaviour being well known to OS-9 users and programmers is called re-entrant. It helps to design powerful systems without wasting memory, since a general law in system integration says "the same code doesn't need to stay in memory more than once". If, however, different programs use the same library, this library code is linked into every single program so that this code definitely stays in memory more than once. The question, therefore, arises of how to make a library re-entrant so that it behaves like a program module. Other operating systems have solved this problem by shared library concepts or by dynamic link libraries (DLLs). Under the OS-9 operating system, trap handlers are normally used for this purpose. They represent a basic working mechanism of the 68k processor family, and many commonly available programs make use of trap handlers; nearly all OS-9 utilities use the C library trap handler (*cio* in OS-9 2.4 or *csl* in OS-9 3.0) and the *math* trap handler that are both part of the OS-9 standard delivery. Even the OS-9 kernel is, in principle, a trap handler (trap number 0) and every system call is a trap handler call to the kernel. Writing a trap handler is not very difficult, and from version 2.4 of OS-9 Professional onwards, example programs of how to create a trap handler library are part of the standard delivery (*/dd/C/SOURCE*).

Many OS-9 programmers and system integrators, however, do not like and, thus, do not use trap handlers. Frequently used arguments against trap handlers are their lack of flexibility, slow calling interface, poor documentation and non-trivial integration into an existing make environment. It is often proposed to use subroutine modules instead. Unfortunately, most of the above arguments apply to subroutine modules as well. In addition, subroutine modules normally cannot have initialised global variables nor are they managed in any other way by the kernel. They just are accepted as memory modules. The only existing development support is that the linker appropriately sets the global data requirement in the module header. On the other hand, a working development environment to use subroutine modules for shared libraries has not yet been made available. It is, therefore, difficult to decide which one a trap handler or a subroutine module is better suited for a given purpose. The aim of the current article is to develop and to present a shared library concept for OS-9 that is based on subroutine modules.

Principle

General

A subroutine module that contains a shared library is best composed of two main code sections, a jump table that is written in assembly language and the actual shared library functions. In the example presented herein, the latter are written in C language but, in principle, they could have been written in any other language. When a program requires a particular function from the subroutine module, it must be linked against a special library that manages the access to the subroutine module containing that specific function. This management library is written partly in assembly and partly in C language.

In addition to these two functionally different parts, subroutine module and management library, an example program written in C is provided. Names and purposes of the various parts are given in the following table:

Name	**Purpose**	**Destination**
Subroutine modules		
submod.a	Subroutine module body with jump table	submod0, submod1
submodprog0.c	Shared library functions	submod0
submodprog1.c	Another set of shared library functions	submod1
Management library		
submodlib_a.a	Data initialisation and pointer management	submodlib.l
submodlib_c.c	Data allocation and handle management	submodlib.l
Example program		
submodlib.l	Complete management library	usesubmod
usesubmod.c	Example program that uses a shared library	usesubmod

Global Data Allocation and Pointer Management

Since the main program and the subroutine module are linked independently from each other, both global data pointers start from the relative position 0. As a consequence, write accesses to global data in the subroutine module would destroy the data in the main program and vice versa. It is, therefore, necessary to provide an initialisation function that allocates the required amount of global data before any function of a subroutine module is called. Information about the required amount of memory is provided by the linker in the module header at offset *M$Mem* or *_mh._mdata*. In addition, prior to every call of a subroutine function, the global data register *a6* must be set to point to this newly allocated data space. It must be reset to point to the main program's data space whenever program execution leaves the subroutine module and resumes

in the main program. Finally, a termination function should be available that returns the granted memory to the system when a particular subroutine module is no longer needed.

Global Data Initialisation

Unfortunately, it is not sufficient to simply allocate the required data space. Languages such as C define data types that are automatically set to 0 (global data) or are even set non-procedurally to any given absolute or relative value (initialised global data). In a normal OS-9 program, this initialisation is partly done by the linker and partly by the kernel when executing an *F$Fork* call. The kernel's action is needed, because OS-9 does not use a mapping memory management unit so that the positions of code and data are only known at run-time and must, therefore, remain position-independent until then. The linker provides two lists in the program module for this purpose; one of these lists is located at header offset *M$IData* or *_midata* and contains offsets and initialisation values for non-remote (16-bit offsets) and remote (32-bit offsets) global data. The other list is located at header offset *M$IRefs* or *_midref* and contains offsets and initialisation values for pointer data. Correctly speaking, the second list again consists of two lists, the first list contains offsets to data locations that need to be corrected by the start address of the program module and the second list contains offsets that need to be corrected by the start address of the global data space. All these lists are generated, when the linker produces a normal OS-9 program or a trap handler – subroutine modules normally do not contain such lists. The concept for shared libraries as presented herein is, therefore, based on the principle that a trap handler is produced in a first step and only transformed into a subroutine module later. This procedure forces the linker to correctly set-up the initialisation lists.

Realisation

Creation of the Subroutine Module

Subroutine modules can only be created by an assembly language directive that specifies the adequate module characteristics:

```
psect    submod_a,(SubMod<<8)+Objct,(ReEnt<<8)+Revision,1,0,entry,0
```

As mentioned above, in a first step a trap handler and not a subroutine module is created. Thus, the appropriate assembly directive to create the module is

```
psect    submod_a,(TrapLib<<8)+Objct,(ReEnt<<8)+Revision,1,0,entry,0
```

The label "entry" points to a table header

```
          org 0
SM$Magic  do.l      1                ; Magic number
SM$IFRev  do.w      1                ; Interface revision
SM$ModRev do.w      1                ; Module revision
SM$Funcs  do.l      1                ; Number of functions
```

that is followed by a jump table that contains as many

```
          org 0
SE$Flag   do.w      1                ; Flag
SE$Return do.l      1                ; Return buffer size
SE$Func   do.l      1                ; Function offset
```

entries as functions indicated at offset *SM$Funcs*.

In addition, the module must contain information to reserve global memory as required by *cstart* in form of

```
          vsect

_sttop:   ds.l      1                ; stack top
_mtop:    ds.l      1                ; current non-stack memory top
_stbot:   ds.l      1                ; current stack bottom limit
```

etc.

```
          ends
```

statements.

Thus, the remaining code for the trap handler module that will, later on, become a subroutine module with two example functions *getprop()* and *putprop()* has the form

```
          use       <oskdefs.d>
          use       submod.d

Revision  equ       0

          psect     submod_a,(TrapLib<<8)+Objct,(ReEnt<<8)+Revision,1,0,entry,0

entry     submod_header              2

firstfunc

          sm        SUBMOD_INT,0,getprop
          sm        SUBMOD_INT,0,putprop

lastfunc
          ends
```

The required macros are defined in *submod.d*:

```
submod_header macro
          dc.l      SUBMOD_MAGIC     ; magic
          dc.w      SUBMOD_IFREV     ; interface revision
          dc.w      \1               ; module revision
```

```
            dc.l        (lastfunc-firstfunc)/ENTRYLEN   ; number of functions

            endm

sm          macro
            dc.w        \1
            dc.l        \2
            dc.l        \3
            endm
```

Metamorphosis from Trap Handler to Subroutine Module

In order to convert a trap handler that has been created as explained above into a subroutine module, a special program called *traptosub.c* is needed. The only important source code lines are:

```
mod->_mh._mtylan = mktypelang(MT_SUBROUT,ML_OBJECT);
_setcrc(mod);
```

The call to the *traptosub* program is part of the automatic make procedure. The *makefile* as well as other code segments not shown here are available on the OS-9 International code disk.

Global Data Allocation

Before a program can use code located in a subroutine module, the module must be loaded into memory or, if already there, its link count must be incremented. This is done in the function *init()* from the management library. It must be called prior to any function call in the subroutine module. The *init()* function expects the name of the subroutine module as first and a revision number as second argument. The name of the subroutine module is passed to the *_init_c()* function that attempts to link to the module or, if this fails, to load the module. If both fail, an error is generated and the function exits. Otherwise, *_init_c()* looks for a free entry in the module handle list. This handle list is a dynamically growing list the entries of which contain the start address of a subroutine module, the start address and the size of its global memory as given in the following structure type definition

```
typedef struct submodhandle {
  mod_exec *submod;
  char     *globmem;
  int       globmemsize;
} SUBMODHANDLE;
```

If a free handle can be found, the memory for the handle is allocated, and the subroutine module's start address is written to the structure element *submod*. The total amount of required global memory is then taken from the module header at offset *_mdata* and allocated from the system; its address and size are also written to the structure. If a free handle cannot be found, e.g. during the first call to the *init()* function, the handle list is expanded by

HANDLECHUNK that is currently set to 32. This part of the library is written in C language and has the following main elements:

```
/*
 * _ i n i t _ c
 */
int _init_c(char *submodname)
{
  int       i, rv, datasize;
  mod_exec *submod;

  if ((submod = modlink(submodname, mktypelang(MT_SUBROUT,ML_OBJECT))) ==
(mod_exec *) -1) {
    if ((submod = modloadp(submodname, MP_OWNER_EXEC, NULL)) == (mod_exec *) -1)
      return((int) submod);
  }

  datasize = (submod->_mdata + 8) & 0xfffffffc;  /* add 4 plus alignment */

  if ((rv = findfreehandle()) == -1) { /* no free handle found */
    if ((_handles = (SUBMODHANDLE **) realloc((char *) _handles,
      sizeof(*_handles) * (_handlenum + HANDLECHUNK))) == NULL)
      return(-1);
    memset((char *) (_handles + _handlenum), 0,
      sizeof(*_handles) * HANDLECHUNK);
    _handlenum += HANDLECHUNK;
    rv = findfreehandle();
  }
  if ((_handles[rv] = (SUBMODHANDLE *) malloc(sizeof(**_handles))) == NULL)
    return(-1);
  _handles[rv]->submod = submod;
  _handles[rv]->globmemsize = datasize;
  if ((_handles[rv]->globmem = (char *) malloc(datasize)) == NULL) {
    free(_handles[rv]);
    return(-1);
  }
  memset((char *) _handles[rv]->globmem, 0, datasize);
  return(rv);
}

/*
 * f i n d f r e e h a n d l e
 */
static int findfreehandle()
{
  int i;

  for (i = 0; i < _handlenum; i++) {
    if (_handles[i] == NULL)
      break;
  }

  return(i == _handlenum ? -1 : i);
}
```

```
/*
 * s h o w s u b m o d s
 */
void showsubmods(void)
{
  int i;

  for (i = 0; i < _handlenum; i++) {
    if (_handles[i] != NULL)
      fprintf(stderr, "Subroutine module '%s' has %d Bytes at %08X\n",
        (char *) _handles[i]->submod + _handles[i]->submod->_mh._mname,
         _handles[i]->globmemsize, _handles[i]->globmem);
  }
}
```

The function *showsubmods()* writes a list of all currently known subroutine modules together with size and start address of its global memory to the standard error path. It is not really needed, but is intended for debugging purposes during program development.

Global Data Initialisation

As already mentioned above, global data initialisation is normally done by the kernel. Since the subroutine module requires exactly the same procedure, the code in the *initdata()* function is probably not very different from the code that is part of the kernel's *F$Fork* call.

The two functions *_init_c()* and *initdata()* are called by the actual *init()* function that, again, is written in assembly language. In addition, if the subroutine module is not planned to be linked with a C trap handler (*cio* or *csl*), this function performs the initialisation of all global variables that are needed by the C library, mostly for stack checking and buffered I/O.

```
init:
          move.w    d1,_revision(a6)   ; save expected revision
          bsr       _init_c            ; do memory initialisation in C
          tst.l     d0                 ; test return value
          blt.s     _init99            ; end, if error
          move.l    d0,_handle(a6)     ; save handle
          movem.l   d0-d3/a0-a3,-(a7)
          move.l    _handles(a6),a2    ; base of handles
          move.l    d0,d2              ; our handle nummer
          move.l    (a2,d2.l*4),a2     ; our handle
          move.l    SI$SubMod(a2),a0   ; get address of our subroutine module
          move.l    SI$GlobMemSize(a2),d0 ; get size of our static memory
          move.l    SI$GlobMem(a2),a2  ; get address of our static memory
          move.l    a0,d2              ; save address of our memory module

          move.l    a0,a1
          bsr       initdata           ; initialise global data
```

imitate cstart

```
            movem.l    (a7)+,d0-d3/a0-a3
            move.l     _handle(a6),d0      ; restore our handle number
            rts
```

The function that performs the jump to the code in the subroutine module is defined in a macro in *submod.d*

```
func macro
\1:
            movem.l    d0-d3/a0-a3,-(a7)
            move.l     a6,-(a7)
            bsr        _init               ; returns module in a0, mod entry in a2
            bcs        _reverror
            lea.l      SM$Table(a0),a1     ; start of entry table
            move.l     SE$Func+sm$\1(a1),d2    ; function offset
            move.l     d1,d0               ; first argument
            jsr        (a2,d2)
            bra        _end
            endm
```

so that the actual call of the *getprop()* or *putprop()* function in the library is short and easy:

```
            func getprop
            func putprop
```

Should the library be supplemented with other functions, they may simply be appended here using the above given macro *func*.

Finally, a library function is needed that is called prior to every jump to the subroutine module and that takes care of the global data pointer. In addition, it checks whether the subroutine module is valid and has a correct revision number.

```
*
* _ i n i t
*
* Check revision and set submod's global memory pointer
*
* Input:   d0         handle number
*
* Output:  a2         start of submod
* Output:  a0         entry into submod
*
_init
            move.l     a6,a1               ; save main's global data pointer
            move.l     _handles(a6),a0     ; base of handles
            move.l     (a0,d0.l*4),a0      ; our handle
            move.l     SI$GlobMem(a0),a6   ; set submod's global data pointer
            cmp.l      #0,a6               ; initialised?
            beq.s      _init99             ; no!
            adda.l     #$8000,a6           ; bias
            move.l     SI$SubMod(a0),a0    ; start of submod
            move.l     a0,a2               ; save it
            add.l      M$Exec(a0),a0       ; module entry
            move.l     SM$Magic(a0),d3
            cmp.l      #SUBMOD_MAGIC,d3    ; match?
            bne.s      _init99             ; no!
```

```
            move.w      SM$ModRev(a0),d3   ; get module interface revision
            cmp.w       _revision(a1),d3   ; at least expected revision?
            bhi.s       _init99 no!
            rts
_init99     ori.b       #1,ccr             ; set carry
            rts
```

Shared Library Functions *getprop()* and *putprop()*

The entries *getprop* and *putprop* are already referenced in the subroutine module's jump table but not yet defined as a valid code segment. They must be defined in the program section that is intended to be made available in form of the subroutine module. Such programs are normally written in the C language. Since global data and even initialised global data are supported, all elements of the C language can be used without any restriction. The following program *submodprog0.c* is intended to serve as an example and does not perform any useful action except testing. A similar program (*submodprog1.c*) is available that is linked into a second subroutine module (*submod1*). Both subroutine modules are called from a test program in order to show the ability of the shared library concept to support more than one simultaneously linked subroutine module. Here are some example lines from the source code (*submodprog0.c*) that contains the first set of the two shared library functions *getprop()* and *putprop()*:

```
char *strconst;

char *initstrconst = "I am an initialised string constant\n";

int initint = 12345678;

int putprop();
int (*function)() = putprop;

char **straddr = &initstrconst;
int *initaddr = &initint;

/*
 * g e t p r o p
 */
int getprop(char *str)
{
  int i;

  errno = 216;
  strconst = "I am a string constant\n";
  if ((i = readln(0, str, 255)) > 0)
    str[i - 1] = '\0';
  else
    str[0] = '\0';
  printf("You entered '%s'\n", str);
  printf("Setting errno to %d\n", errno);
  return(getpid());
}
```

```
/*
 * p u t p r o p
 */
int putprop(char *str)
{
  char  buffer[128];
  char *cp;

  writeln(1, str, strlen(str));
  writeln(1, "\n", 1);
  writeln(1, "Printing a string constant:\n", 80);
  writeln(1, strconst, strlen(strconst));
  writeln(1, initstrconst, strlen(initstrconst));
  writeln(1, *straddr, strlen(*straddr));

  cp = getenv("TERM");
  printf("TERM is '%s'\n", cp == NULL ? "unknown" : cp);

  printf("Does buffered I/O work?\n");

  printf("Initialised integer = %d\n", initint);
  printf("Initialised integer = %d\n", *initaddr);

  sprintf(buffer, "Our process ID is %d\n", getpid());

  printf("This function is located at address %08x\n", putprop);
  printf("This function is located at address %08x\n", function);

  printf("This is an MC%d CPU\n", _getsys(D_MPUType, 4));

  return(0);
}
```

Test Program

Last but not least, a test program is needed. The following code lines are part of a program that links and tests *submod0* first, then links to *submod1* and, finally, uses them both interchangeably:

```
main()
{
  char     *submodname0 = SUBMODNAME0;
  char     *submodname1 = SUBMODNAME1;
  int       submodno0, submodno1, retval;

  printf("Initialising...\n");
  submodno0 = init(submodname0, IF_REV);
  if (submodno0 == -1)
    exit(_errmsg(errno, "can't init subroutine '%s' due to ", submodname0));

  printf("Asking for input:\n");
  retval = getprop(submodno0, str);
  str[strlen(str) - 1] = '\0';
  printf("errno set to %d\n", errno);
```

```
    printf("Input was '%s'\n", str);
    printf("Function returned %d.\n", retval);

    strcpy(str, "OUTPUT");
    printf("\nOutput will be '%s'\n\n", str);
    if ((retval = putprop(submodno0, str)) < 0)
      exit(_errmsg(errno, "can't write due to "));
    printf("\n\n");
    printf("Function returned %d.\n", retval);

    printf("Initialising another one...\n");
    submodno1 = init(submodname1, IF_REV);
    printf("Done (%d).\n", retval);
    if (submodno1 == -1)
      exit(_errmsg(errno, "can't init subroutine '%s' due to ", submodname1));

    showsubmods();

    strcpy(str, "ANOTHER OUTPUT");
    printf("\nOutput from second subroutine module will be '%s'\n\n", str);
    if ((retval = putprop(submodno1, str)) < 0)
      exit(_errmsg(errno, "can't write due to "));
    printf("\n\n");
    printf("Function returned %d.\n", retval);

    strcpy(str, "OUTPUT");
    printf("\nOutput from first subroutine module will be '%s'\n\n", str);
    if ((retval = putprop(submodno0, str)) < 0)
      exit(_errmsg(errno, "can't write due to "));
    printf("\n\n");
    printf("Function returned %d.\n", retval);

    printf("Terminating...\n");
    retval = term(submodno0);
    printf("Function returned %d.\n", retval);

    printf("Terminating another one...\n");
    retval = term(submodno1);
    printf("Function returned %d.\n", retval);

    printf("Thank you for using SUBMODs.\n");
}
```

Limitation

There are two different ways to use the above shared libraries: one way is to combine them with the standard OS-9 trap handler (*cio* or *csl*). If this is done and both the main program and the subroutine module use the same trap handler, all I/O function including buffered I/O are safe, and there are no known limitations as to what extend buffers and pointers may be shared between functions of the main program and functions of the subroutine module. If, however, library C functions are used, or C functions are used even interchangeably from trap handler and library, the shared library concept has one important limitation that is obvious from the

way C library global data are handled. Information is then only transported from the main program to the shared library function but not vice versa. In addition, the information is not updated after the call of the initialisation procedure. Therefore, buffered I/O must always be flushed before branching between the main program and the subroutine module. In addition, buffered files must not be opened in one module and closed in the other module and vice versa.

Conclusion

The concept for shared libraries based on subroutine modules presented herein, has been thoroughly tested and used in a real-world application. If the above limitation is considered, this concept can be recommended without any other restriction.

Reference

[1] Dayan PS (1992) *The OS-9 Guru, 1 – The Facts*, edition 1, Galactic Industrial Ltd., Durham UK.

The complete development environment for shared libraries based on subroutine modules as described in the current article is available on the OS-9 International code disk.

Carsten Emde can be reached by email at <carsten@effo.ch>.

PCMCIA/JEIDA support under OS-9

- PCMCIA/JEIDA card raw access
- FLASH read/write supported
- EPROM emulation devices available
- MCDISK – SCSI device with full PCMCIA/ JEIDA and ATA support

High-Tech·Made in Switzerland

Täfernstrasse 20
CH-5405 Baden-Dättwil

Tel. ++41 56 83 30 80
Fax ++41 56 83 30 20

GNU *make* for OS-9

Carsten Emde

Introduction

The availability of the GNU C compiler *gcc2*, a Bourne-like shell *sh* and a Unix-like library of C functions *os9lib.l* has made it much easier to port Unix software to OS-9 than ever before. One important tool, however, was still missing: a standard *make*. This article presents GNU make for OS-9 that executes many Unix *makefiles* without requiring more than only marginal adaptations. In order to allow for the coexistence with the original OS-9 *make* tool, GNU make for OS-9 was called *gmake*.

Technical Details of the Port to OS-9

Very few modifications were necessary to adapt *gmake* to OS-9; the main changes relate to the default settings in the file *default.h*. In order to let *gmake* behave similarly to OS-9's original make tool – as far as the search strategies and names for default directories are concerned – the following changes were made to the default variables and implicit rules.

Default Variables

```
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
LINK.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)
LINK.r = $(CC) $(LDFLAGS) $(TARGET_ARCH)

LINK.cc = $(CXX) $(CXXFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)
COMPILE.cc = $(CXX) $(CXXFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
```

Implicit Rules

```
%.r:
%: %.r
        $(LINK.r) $(RDIR)/$^ $(LOADLIBES) $(LDLIBS) -o $(ODIR)/$@

%.a:
%.r: %.a
        $(RC) $(SDIR)/$< $(RFLAGS) -o=$(RDIR)/$@
```

```
%.c:
%: %.c
        $(LINK.c) $(SDIR)/$^ $(LOADLIBES) $(LDLIBS) -o $(ODIR)/$@

%.r: %.c
        $(COMPILE.c) $(SDIR)/$< $(OUTPUT_OPTION)

%.cc:
%: %.cc
        $(LINK.cc) $(SDIR)/$^ $(LOADLIBES) $(LDLIBS) -o $(ODIR)/$@

%.r: %.cc
        $(COMPILE.cc) $(SDIR)/$< $(OUTPUT_OPTION)
```

Similar changes were made for other languages (FORTRAN, Pascal) and other tools (*yacc, lint*).

Using *gmake*

The *gmake* program is easy to use, since most of the frequently required run-time options, for example '-n' and '-d', are the same in the two make tools *gmake* and *make*. But *gmake* has a number of additional features such as the '-p' option. This option lets *gmake* reproduce the current status of the internal data base (implicit, environment-derived and explicit variables, and implicit and explicit rules). These settings are written in form of a *makefile* which is very helpful for debugging purposes.

When *gmake* is started with the '-h' option the following usage information is provided:

```
Usage: gmake [options] [target] ...

Options:
  -b, -m                      Ignored for compatibility.
  -C DIRECTORY, --directory=DIRECTORY
                              Change to DIRECTORY before doing anything.
  -d, --debug                 Print lots of debugging information.
  -e, --environment-overrides
                              Environment variables override makefiles.
  -f FILE, --file=FILE, --makefile=FILE
                              Read FILE as a makefile.
  -h, --help                  Print this message and exit.
  -i, --ignore-errors         Ignore errors from commands.
  -I DIRECTORY, --include-dir=DIRECTORY
                              Search DIRECTORY for included makefiles.
  -j [N], --jobs[=N]          Allow N jobs at once; infinite jobs with no arg.
  -k, --keep-going            Keep going when some targets can't be made.
  -l [N], --load-average[=N], --max-load[=N]
                              Don't start multiple jobs unless load is below N.
  -n, --just-print, --dry-run, --recon
                              Don't actually run any commands; just print them.
  -o FILE, --old-file=FILE, --assume-old=FILE
                              Consider FILE to be very old and don't remake it.
  -p, --print-data-base       Print make's internal database.
  -q, --question              Run no commands; exit status says if up to date.
```

```
-r, --no-builtin-rules      Disable the built-in implicit rules.
-s, --silent, --quiet       Don't echo commands.
-S, --no-keep-going, --stop
                             Turns off -k.
-t, --touch                 Touch targets instead of remaking them.
-v, --version               Print the version number of make and exit.
-w, --print-directory       Print the current directory.
--no-print-directory        Turn off -w, even if it was turned on implicitly.
-W FILE, --what-if=FILE, --new-file=FILE, --assume-new=FILE
                             Consider FILE to be infinitely new.
--warn-undefined-variables  Warn when an undefined variable is referenced.
```

Important Features

GNU make has so many features that it is impossible to describe them as part of this article, but a detailed users' manual in Postscript format (*make.ps*) is part of the software distribution. Only three important features are mentioned here in order to exemplify *gmake*'s versatility.

Make without *makefile*

Actually, *gmake* does not require a *makefile* to be present; if the default rules are acceptable, it is sufficient to simply enter

```
$ gmake program
```

but it is even possible, again without *makefile*, to define specific compiler and linker options, e.g.

```
$ gmake program 'CFLAGS=-O2 -v' ODIR=/dd/MYCMDS
```

Include Directive

System-wide definitions such as compilation rules and options can be held in a separate file and included in the same way as header files can be included into C sources. The following settings may, for example, be written to */dd/SYS/cflags*

```
OPT = -optasm -O2 -fomit-framepointer
CPU = -mc68020
TRP = -uwlibs -ctrap

CF  = $(OPT) $(CPU) $(TRP)
```

and included into every project's *makefile*

```
include /dd/SYS/cflags

CFLAGS  = $(CF)
LDFLAGS = $(TRP) $(CPU)

ODIR    = /dd/MYCMDS

program:  program.r
```

Environment Variables

All environment variables can be evaluated in the *makefile*. This allows, for example, defining environment variables that modify the compilation and linking procedure. A common application is the definition whether a host or a target software version is produced. The make procedure may then also take care that the newly produced target version is sent to the target, e.g. via network. Such a *makefile* could have the following form:

```
program:   program.r

ifeq ($(PLATFORM), HOST)
  @echo Producing host software
  @$(CC) $(LDFLAGS) -D$(PLATFORM) $@.r -o $@
endif

ifeq ($(PLATFORM), TARGET)
  @echo Producing target software
  @$(CC) $(LDFLAGS) -D$(PLATFORM) $@.r -o $@
  @send2target
endif
```

If host or target software is to be compiled, the lines

```
$ setenv PLATFORM HOST
$ gmake
Producing host software
```

or

```
$ setenv PLATFORM TARGET
$ gmake
Producing target software
Sending software to target ... Done.
```

must be entered, respectively.

Conclusion

The compatible make utility for OS-9 fills the remaining gap in the list of tools that are required for porting Unix software to OS-9. Many large software packages such as *Ghostscript* etc. can now be ported much easier to OS-9, since most time was spent in the past to adapt the *makefiles*. The port of small tools to OS-9 may even take no more time than it takes to port these tools to a Unix station, the operating system of which also is not explicitly listed in the makefile. The current version of GNU's stream editor *sed*, for example, could be ported to OS-9 without requiring any major change to the *makefile* or to the sources so that the entire port was done in less than 5 minutes.

The described software is available as PD #115 from EFFO. The printed version of the mentioned manual (more than 160 pages) is also available from EFFO at a nominal handling fee.

Carsten Emde can be reached via email at <carsten@effo.ch>.

OmniRay
AUTOMATION MIT SYSTEM

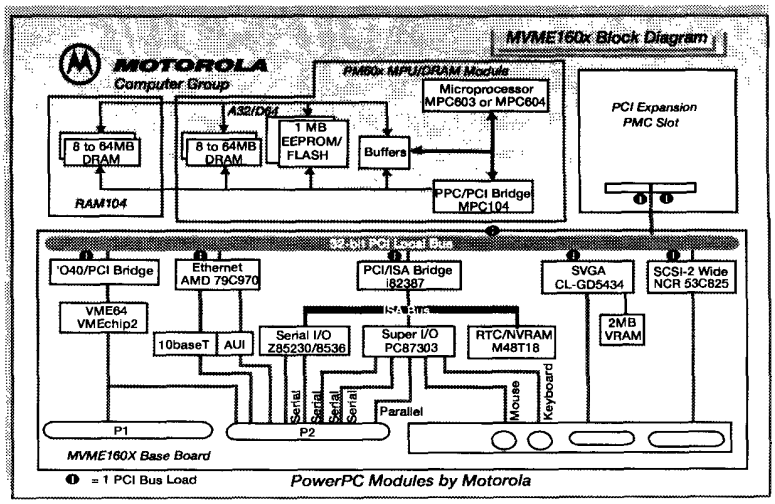
SY 01/95

More power for VME-boards thanks to the PowerPC from Motorola.

The new product family MVME160x establishes new standards concerning price/performance as well as concerning maximum performance. Optimal flexibility is achieved by a modular design: processor-, memory- and mezzanine-modules for I/O are exchangeable.

The user has, for example, the choice between CPU-module with PowerPC603 or -604 (66 resp. 100 MHz clock frequency) and 8 to 128 Mbyte DRAM that are mounted on the base board with the I/O controllers.

The base board offers maximum functionality with Ethernet, 16-bit SCSI-2, superVGA-graphics, mouse-port, keyboard-port and four serial-ports in one single VMEbus-slot. OS-9 is released on these boards.



Omni Ray AG
Industriestrasse 31, CH-8305 Dietlikon/Zurich, Phone 01 835 21 11, Fax 01 833 50 81
A Company of Sonepar Electronique International SEI

Debugger Insights

Carsten Emde

Hard Versus Soft Breakpoints

In principle, two different methods exist to execute a program under the control of a debugger. The first method is called "soft breakpoint", it simply surveys the program counter register: the debugger executes the program step by step, and the address in the program counter is compared with a predefined value (the breakpoint) after every single instruction. The advantage of this method is that it can be used irrespective of whether the code is located in RAM or ROM, since the code does not need to be modified. As a disadvantage, the program executes much slower than under normal conditions; in consequence, a code section with critical timing conditions normally cannot be debugged with soft breakpoints. In addition, debugging can be very time consuming. Therefore, a second method exists that is based on exception processing, e.g. using the "illegal instruction" vector. In a first step, the debugger inserts the address of its own exception handler at the appropriate vector (e.g. vector #3), and then replaces the instruction at the chosen breakpoint by an instruction that causes exception processing, e.g. Motorola's reserved "illegal" instruction 0x4AFC. The debugger may then start the program in the same way as if it had been started without debugger, but execution is stopped and control is returned to the debugger whenever the program counter reaches the inserted illegal instruction. This is called "hard breakpoint"; its only disadvantage is that the code must be modified, i.e. the method cannot work, if the code is located in ROM.

OS-9 Debuggers

The source level debugger *srcdbg* and the system-state debugger *sysdbg* always use hard breakpoints. The user-state debugger *debug* and the ROM-level debugger, however, may be employed in different ways so that they use either hard or soft breakpoints. Unfortunately, these two debuggers have a different user interface so that different commands must be employed for a particular breakpoint method.

The User-state Debugger *debug*

The commonly used commands to prepare a program for execution, to set a breakpoint and to start execution are

```
$ debug

dbg: f program
default symbols belong to 'program'
dn: 00000031 0000002E  00000081 00000003   00000000 000004E2  000017D0 00000000
an: 00000000 001227D0  00000000 01678B00   00000000 001222EC  00129000 001222EC
pc: 01678B50  cc: 00 (—)
<FPCP in Null state>

_cstart             >2D468010         move.l d6,_totmem(a6)

dbg: b main

dbg: g
dn: 00000001 0012271E  00000001 00000003   00000000 000004E2  000017D0 00000000
an: 0000014C 00000000  0012271E 00122714   00122710 00000000  00129000 001222E4
pc: 01678D90  cc: 10 (X—)
<FPCP in Null state>
main                >4E550000         link.w a5,#0
```

If these commands are entered, *debug* uses soft breakpoints, i.e. execution is relatively slow and any real-time behaviour is disabled. Instead of 'g' (go), however, the 'x' (execution) command can be used. This command uses hard breakpoints, but in contrast to 'g' an argument is expected that specifies the maximum number of instructions to be executed, unless a breakpoint is encountered. In order to imitate the 'g' command, the highest number possible must be entered, i.e. 'ffffffff'. Since *debug* automatically transforms negative numbers to the 2's complement representation, '-1' can be entered instead. In conclusion, the command

```
dbg: g
```

starts execution with soft breakpoints, the command

```
dbg: x-1
```

starts execution with hard breakpoints.

The ROM Level Debugger

By default, the ROM level debugger is set to use hard breakpoints. The command 'o' allows to modify various program settings that can be inspected with the 'o?' command:

```
RomBug: o?
a                    - toggle control register display
                       (68010/68020/68030/68040/683XX)
b<n>                 - numeric input base radix
c<n>[:f]             - set MPU type to <n> (68000/etc), FPCP type to 6888<f>
d                    - toggle FPCP decimal register display
e <addr>             - display exception frame (default <addr> is .a7)
f                    - toggle FPCP register display
m                    - toggle MMU register display
r                    - toggle rom type (soft) or ram type (hard) breakpoints
x                    - toggle disassembly hex output format
```

```
v                       - display vectors being monitored
v[-][s|u][d]<n> [<m>] - monitor exception vector ('-' to restore vector)
                          's' system state only, 'u' user state only
                          'd' display only, <n> vector number in decimal,
                          'm' upper limit vector number in decimal
v?                      - display all exception vector values
```

As can be seen, the 'or' command toggles the breakpoint method. In conclusion, the ROM level debugger uses hard breakpoints by default; the command

```
RomBug: or
```

entered once, causes the ROM level debugger to use soft breakpoints.

Carsten Emde can be reached via email at <carsten@effo.ch>.

microware

The Only Real-Time Total Solution Supplier

MORE CHOICE - MORE OPTIONS - TOTAL SUPPORT

Microware's **OS-9** Real-Time Operating System is available for the Motorola 68k, Intel X86 and PowerPC processor families, with off-the-shelf I/O to support virtually any demanding real-time applications.

TIGHTLY INTEGRATED TOOLS

To accelerate your project, Microware puts easy-to-use development tools at your fingertips. **FasTrak** is Microware's development environment built around **Ultra C**, our ANSI C compiler. **Ultra C** brings true interprocedural and global optimization. **FasTrak** is available for Unix and now for Windows 3.1.

The tight integration of OS-9 and development tools boots your productivity and reduces your time-to-market.

PROVEN QUALITY

In over 5000 products, designers have relied on Microware's quality solutions for their demanding applications. Microware's recent **ISO 9001 certification** - the first such certification in the system software industry - reflects our total commitment to quality and reliability in our products.

Learn how Microware can handle your real-time design challenges. Call us at *(33) 42 58 63 00*

MICROWARE SYSTEMS FRANCE

Château de la Saurine, Pont de Bayeux - 13 590 MEYREUIL FRANCE - Tel : (33) 42 58 63 00 / Fax : (33) 42 58 62 28

All trade or product names are trademarks of registered trademarks of their respective holders

Letters to the Editor

Big Hard Disks Under OS-9

OS-9 International 1/95, p. 21

We read with interest the article about big hard disks. There is an important addition to make that supports the recommendation to limit the size of an OS-9 hard disk partition to 2 GByte: NFS for OS-9 is unable to address a file, if it is located at a higher logical address than 2 GByte. According to Microware, this limitation is already part of Sun's original NFS software and was not removed in the OS-9 port. The problem is still present in OS-9 V3.0, and Microware has not announced any plans to release a new version that allows to mount larger hard disks than 2 GByte via NFS.

Gerald Nimmrich, EKF Elektronik Messtechnik GmbH , <gn@ekf.werries.de>

OS-9 3.0 - What is New?

OS-9 International 1/95, p. 9

There is a comment to be made to Beat Forster's article "OS-9 3.0 - What is New?", since, in the meantime, Microware has released the 3.0.1 drop-in upgrade accompanied by a bug report. In general, the article reflects quite well the situation as reported by Microware; the *tsleep(1)* problem has, in fact, been solved. The kernel's *P$PModul* (primary module) field that pointed to the kernel's name instead of its address, however, was also fixed and now behaves identically to all other modules. Retrospectively, the irregular behaviour in 3.0.0 was a mistake and not, as the article suggests, a feature.

Wolfgang Ocker, reccoware systems, <weo@recco.de>

_getsys();

Reto Peter

Monthly EFFO Meetings

The winds of change have again visited EFFO. Starting in June, the monthly EFFO meeting takes place in the Restaurant "Zunfthaus am Neumarkt" in Zurich. Its exact address is Zunfthaus am Neumarkt, Neumarkt 57, CH-8001 Zurich, phone +41 1 252 79 39. It is located in the heart of Zurich and can easily be reached from the main railway station using tram 3 or bus 31 (stop "Neumarkt").

As usual, the meeting starts at 8 PM, but most participants meet at 7 PM in the Restaurant to have supper together.

Everybody interested in OS-9 is kindly invited to join the meeting.

Imprint

Published by European Forum For OS-9 (EFFO)
President Werner Stehling
Vice President Reto Peter
Director of Finance Stephan Paschedag
Editor-in-Chief Carsten Emde
Design Marc Balmer, Werner Stehling (layout)

Address
European Forum For OS-9
P.O. Box
8606 Greifensee
Switzerland

OS-9 International

FAX +41 1 940 38 90
email os9int@effo.ch

Copyright © 1995 by European Forum For OS-9 (EFFO).
Copyright © (design) 1994 by Marc Balmer.
All rights reserved. No part of this journal may be reproduced without the prior written permission of the publisher. All source code is provided without any warranty. Trademarks are not marked as such.

Printed directly from disk by Fotoplast, Zurich, Switzerland
ISSN: 1019-6714

Subscriptions

OS-9 International is the official organ of the European Forum For OS-9 (**EFFO**). The subscription is included with the annual **EFFO** membership fee. In addition, it is available by separate subscription for non-**EFFO** members, single issues are also available. All following prices are given in Swiss Francs, shipping included:

	Switzerland	*Europe*	*Overseas*
One year (3 issues)	25.00	30.00	35.00
Single issue	10.00	12.00	14.00

To subscribe to **OS-9 International** or to order a single issue send a letter, postcard, fax or email to **EFFO**.

Advertisements

OS-9 International is not only an ideal platform for discussing OS-9 related topics, it is also the ideal place to advertise. **OS-9 International** reaches end-users, system-software developers and, nevertheless, decision-makers.
Please contact **EFFO** for detailed information on how to place an ad in **OS-9 International**.