On LSI-11, RT-11, Megabytes of Memory and Modula-2/VRS

This article was published in DEC® PROFESSIONAL, The Magazine for DEC Users, Professional Press, Spring House, PA. U.S.A., December 1986. This is the version of the article restored in 05-Jun-2008 which differs in the following points: 1) The graphical illustrations of the original article are here: http://www.modulaware.com/history/vrsmop.pdf
2) The four appendices with related Modula-2 interface and implementation modules as well as VN.SYS driver were added.

Development history

Back in 1981, during the DECUS München Symposium in Konstanz, I got the information, that the programming language Modula-2 [Wirth 83] was available for PDP-11 running under RT-11. Mr. Georg Maier from ETH-Zürich reported, that this compiler directly generates native code, supporting both, FIS and FPU (floating instruction set and floating point unit) hardware options. He pointed out, that even a symbolic debugger was available for Modula-2. Best of all: The compiler, linker, debugger and utilities were written in Modula-2 itself and distributed with its source code by the Institut für Informatik at ETH-Zürich.

It was at the same symposium, where the 22 bit addressing capability of LSI-11/23's MMU (memory management unit) was announced.

Introduction to a perfect marriage

Programs always have a trend to grow until they won't fit into low-memory. Sure, you can use overlay techniques, but most programmers dislike this method and it doesn't always help. In late 1983, the Lehrstuhl für Prozeßrechner (LPR) at Technical University of Munich (TUM) came out with a Modula-2 implementation which makes use of the MMU. Now Modula-2 and the MMU have married. Let's call the marriage VRS (virtual run-time system). VRS meets the challenge of developing and running very large programs under the unmapped single job operating system RT-11.

Sure PDP-11 is a 16 Bit computer

Within four years, the memory prices dropped by a factor of 16. Before VRS became available, the only way to use the extended memory was by means of a mapped operating system, which most programmers disliked or with the VM handler, which emulates a fast, but nothing more than a stupid, solid-state memory disk. But there was no transparent way for the programmer to write large application programs without overlays. Sure, the memory management supports virtual addressing of up to 4M bytes. But virtual means not really. Compare it to virtual money: it simply isn't available, once you really need it.

Now the rest of the story

Memory is inexpensive now. A full megabyte is standard on LSI-11/23 and LSI-11/73 systems. Programmer's primary goal is easily defined: Please let me write very very large programs in Modula-2 on my smart desktop or full blown rack-mounted PDP-11 micro, but without overlays and without the need for tricky constructs.

But how to accomplish this on a 16 bit computer? First we have to tell the RTS (run-time system) something about the MMU and extended memory; let's call it VRS instead of RTS now. Then tell the linker to separate code and data. Furthermore tell the linker, loader and symbolic debugger about programs, which should execute **resident** in virtual memory. TUM did it to the Modula-2 implementation with VRS. Note, that the compiler doesn't know anything about MMU and extended memory. The compiler just wonders about the large heap and stack he's got now. This assures compatibility and transparency of Modula-2 programs. Existing system and application programs only have to be linked by the VRS linker to be executable under VRS. An in depth look at PDP-11's Modula-2 programming environment can be found in [Dotz84].

The solution: VRS Real-memory System

VRS is Modula-2 with a virtual run-time system implemented on a real-time computer under a real-time operating system RT-11. Best of all: large programs run really fast under VRS on PDP-11 without memory constraints. And really fast means: you will not notice the difference between so-called unmapped programs and programs running under VRS. Note, that you

can still access the I/O-page directly. Results of the performance test are published in [Dotz85].

Incredible ! How is VRS implemented?

All Modula-2 programmers I know, are very happy with VRS. They are using VRS since 1984 and because it really works fine, they just don't ask how it is implemented. Try it also!

Comments to the VRS implementation

First I'll illustrate the situation for normal unmapped programs running under RT-11. Operating system, device handlers and **user program code, heap, stack, constant and variable data** share the low memory up to 56K byte or 160000B (B means octal). An address between 160000B and 177776B is virtual and is translated by the CPU to a physical address between 17760000B to 17777776B, where the I/O-page is located.

[Picture of Typical unmapped memory layout, see p1 in http://www.modulaware.com/history/vrsmop.pdf]

In VRS, all Modula-2 program code is virtual and resides in extended memory starting at 160000B, freeing the low memory and leaving more space for data. For those who have a good knowledge of PDP-11's architecture, I'll try to explain the memory layout of a Modula-2/VRS program at load time and during execution. Program code is linked in portions of 8K bytes always to the base address 140000B. Depending on the size of the procedures there are one or more procedures in the same area which is called **code segment**. The linker output file consists of descriptor packets for constant and variable data which are directly addressed. This data resides in low memory up to the address 140000B at execution time. Furthermore the linker output file contains virtual code packets (code segments), determined to reside in extended memory during execution. Remember, virtual code means, that all addresses are relative to the addresss 140000B. At load time, the loader fills the extended memory with these code segments. First the memory area from 160000B is filled, then the area starting at 200000B, an so on. During execution the PAR6 of the MMU points to one of these areas to make the code of the currently active code segment available.

Translation of virtual to physical address

Now you know the reason why virtual code packets start at 140000B: if the MMU is enabled, the APF (active page field, determined by the bits 13 through 15 of a virtual address) is used to select a base address via PAR. Since the APF value is 6, this selects PAR6 to construct the physical address. Here two restrictions become visible: 1) Procedures longer than 8K bytes can't be handled. But don't worry! The compiler can't produce longer procedures. 2) Direct RMON (resident RT-11 monitor) database access via fixed offset, which is normaly located somewhere between 140000B and 160000B is not accessible, since either the virtual code or the operating system is mapped at the same time. Clean programs use the .GVAL request to get fixed offset values. Another method is using a special procedure XMEM.GetRT11Word. Both methods avoid the direct access conflict.

Why RT-11's code can be mapped away

But how can RT-11 operate, if its own code (or the code of it's device handlers) isn't always mapped? Operating system services are always requested synchronously via trap instructions (EMT) or asynchronously via hardware interrupts. Both is handled by VRS. VRS intercepts all these requests and loads the appropriate value into PAR6, before passing the request to the operating system. Remember, that memory below 140000B, where possibly some handler code or the USR resides, is always mapped directly.

[Picture of Virtual code mapping, see p2 in http://www.modulaware.com/history/vrsmop.pdf]

Besides separating code and data, the linker changes all subroutine calls (JSR) generated by the compiler to trap instructions (TRAP). The VRS trap handler interprets these traps at execution time and maps another code segment, if the procedure to be called is not in the currently mapped area. This is accomplished through saving and simply loading PAR6 before the call and restoring it, if the called procedure returns. The new PAR6 value is determined by the so-called code segment table, which is part of VRS. This table is filled at load time. Even calling a procedure which is in a new code segment, doesn't produce any remarkable overhead, since this mechanism is implemented efficiently.

What about virtual data

Until now, we were talking about virtual program code. What about virtual data (large data structures such as large arrays and records)? Assuming, that your pure program code is mostly smaller than 72K byte and you have more than 128K (56K low memory + 72K) byte memory, you can even use the memory above 128K up to 4M byte to define virtual data and operate on it via a procedural interface. The module which is responsible for this task is called XMEM (its interface module is listed in the appendix 1). XMEM is part of the resident VRS, since it is used by the program loader, to place your program code into extended memory (since for the loader, your code is data). There are only four procedures to deal with virtual data: DefArray, RemArray, GetElement, and PutElement. Notes: 1) If you are using virtual data via the XMEM procedures, your Modula-2 program is no longer portable across platforms, although the four procedures could easily be emulated on any other platform. 2) Your constants and variables, heap and stack are always located in low memory up to 140000B. The stack pointer for the main process is always on top of this area (i.e. it must be smaller than 140000B). The stack start address is defined by the RT-11 SYSCOM location 42B and 50B in block 0 of the file VRS.SAV (resident Modula-2/VRS system).

[Picture of Virtual data transfer, see p3 in http://www.modulaware.com/history/vrsmop.pdf]

VRS and the VM handler

Even if you are using VRS for virtual code and data, there may yet be a large amount of memory unused. What about RT-11's VM handler, the stupid RAM-disk emulator? VM doesn't know anything about VRS! Clearly VM can't be used with VRS. There is a special version called XS.SYS (eXtended Storage) which has been written from scratch by ModulaWare in 1985 to be compatible with VRS. XS has configurable memory start and stop addresses. The start address is usually at the 18/22 bit boundary at 256K byte or 1000000B. XS can be used as system device (i.e. XS is bootable) and supports 22 bit addressing mode. Furthermore XS operates on non-standard RT-11/Star-eleven systems with short (4K byte) I/O-page also. Note, that VM and VRS need standard 8K byte I/O-page. [See Appendix 4.]

[Picture of VRS and VM sharing extended memory, see p4 in http://www.modulaware.com/history/vrsmop.pdf]

Experiences with Modula-2/VRS

Since you don't care about restrictions concerning program size and overlays, Modula-2/VRS is now the standard programming environment to develop Modula-2 programs on PDP-11 under RT-11. In 1983, for example, a very large program RDS-11 (relational database system) which is part of LIDAS (Lilith database system) was successfully ported to PDP-11/RT-11 using Modula-2/VRS by ModulaWare. The illustrations in this article were created with the graphics editor called SILcad running under RT-11 also using Modula-2/VRS. In 1984 SILcad was ported to PDP-11 and it's functionality was extended by ModulaWare in 1985. To operate SILcad you need the NEC7220A based GDC-11 Q-Bus raster-scan graphics display controller. The interactive design of the drawings are controlled by a three-button mouse connected (via a small adapter) to DRV-11. The size of SILcad's program code is about 50K byte. The stack and heap which resides in low-memory is about 30K byte in size. The large heap is used to store information about SILcad's graphical elements. The hardcopy from the display (17" monitor, 1024 * 832 pixels resolution, 55Hz refresh, non-interlaced) was produced with an inexpensive matrix-printer.
LIDAS [Zehn83] and SILcad are written in Modula-2 and have been originally developed and implemented for the Lilith Modula-2 computer at the Institut für Informatik, ETH-Zürich, Switzerland. For more information on RDS-11 see [Dotz84b].

Availability of the components mentioned

The Modula-2/VRS programming environment, XS.SYS, RDS-11, GDC-11, and SILcad are available from Modulaware.

[The following notes were added in 05-Jun-2008]

Since there is a free SIMH PDP/LSI-11 hardware emulator available on the internet, the Modula-2[/VRS] distribution kit for PDP/LSI-11 might still be of interest for certain users and might still be available on special request; contact gd at modulaware.com. Concerning XS.SYS, see Appendix 4. RDS is part of the VAX and Alpha OpenVMS Modula-2 compiler kit. GDC-11 requires special hardware which is no longer available.

Acknowledgements

Credit is due to Gerhard Koller from TUM/LPR. He developed VRS under direction of Dr.-Ing. Demmelmaier in 1983 using the 1981's M2RT11 distribution kit of Modula-2 from ETH-Zürich, Institut für Informatik, where Modula-2 has first been designed and implemented (on PDP-11).

References

[Dotz84] Dotzel, Günter and Moritzen, Klaus: **Modula-2 and its Environment**. In: Journal of Pascal, Ada & Modula-2. Vol. 3, Nr. 4, West Publishing Co., Orem UT, 1984, pp. 42-46.

[Dotz84a] Dotzel, Günter and Moritzen, Klaus: **LIDAS: A Relational Database System**. In: Journal of Pascal, Ada & Modula-2. Vol. 3, Nr. 5, West Publishing Co., Orem UT, 1984, pp. 32-37, 52.

[Dotz85] Dotzel, Günter: **Benchmark Test Results for Modula-2 Compiler**. In: Journal of Pascal, Ada & Modula-2. Vol. 4, Nr. 2, John Wiley & Sons, Inc., New York (now publisher of the journal), 1985, pp. 28-32.

[Wirth 83] Wirth, Niklaus: **Programming in Modula-2**. Springer Verlag, New York, 1983 (2nd ed.).

[Zehn 83] Zehnder, C.A.: Database Techniques for Professional Workstations, ETH-Report No. 55. ETH Zürich, Institut für Informatik, CH-8092 Zürich/Switzerland.

Appendix 1: XMEM

Since module XMEM is build into the resident Modula-2/VRS run-time system, it does not have a corresponding implementation module written in Modula-2. The modula VIRDATA mentioned in the interface is listed in Appendix 2. Furnished as is; use at your own risk.

```
DEFINITION MODULE XMEM;

  FROM SYSTEM IMPORT WORD, ADDRESS;
(* extended-memory configuration and definitions for Modula-2/VRS

*)
  CONST
    MinCodeSegm = 0;     (* Minimal codesegment number *)
    MaxCodeSegm = 127; (* Maxinal codesegment number *)

    MinPage = 0;     (* Minimal virtual page in system *)
    MaxPage = 7;     (* Maximal virtual page in system *)

    ResCodePage = 6;     (* Resident, significant code page *)
    IOPage = 7;     (* IO page *)

    MinDataPage = 0;     (* Minimal page for data *)
    MaxDataPage = 4;     (* Maximal page for data *)

    BytesPerBlock = 100B;
    BytesPerPage = 20000B;
    BlocksPerPage = BytesPerPage DIV BytesPerBlock;

  TYPE
    XMADDRESS = CARDINAL; (* Physical blocknumber in 64 byte units *)
    ArrayId; (* Array-identification *)
    Pages = [MinPage..MaxPage];
    DataPages = [MinDataPage..MaxDataPage];
    CodeSegmNumber = [MinCodeSegm..MaxCodeSegm];
    CodeSegmTable = ARRAY CodeSegmNumber OF XMADDRESS;

  VAR
    CodeSegmBase [172354B]: XMADDRESS;
    (* only for RT11SJ (kernel PAR 6); user PAR 6 used under SHAREplus *)

    ErrorCodeSegmBase [410B]: XMADDRESS;
    ErrorSPLimit [412B]: CARDINAL;

    CodeSegmTabInfo [540B]: RECORD
      Address: POINTER TO CodeSegmTable;
      MaxCodeSegm: CARDINAL;
    END;

    IOPagePhysBlock [546B]: XMADDRESS;
```

```
    (* --- special return addresses used by RTS to switch codesegments --- *)
    PrevCodeSegm [550B]: PROC;
    BackToModula [552B]: PROC;
    SkipContext  [554B]: PROC;

    (* --- user VRS-procedures: *)
    GetFromPhysMemory [560B]: PROCEDURE (
      (*ElementAddr:*) ADDRESS,
      (*ElementSize in bytes:*) CARDINAL,
      (*BlockBase address in 32-word units:*) XMADDRESS,
      (*Offset within a 32-word unit in bytes:*) CARDINAL);

    PutToPhysMemory [562B]: PROCEDURE (
      (*ElementAddr:*) ADDRESS,
      (*ElementSize:*) CARDINAL,
      (*BlockBase:*) XMADDRESS,
      (*Offset:*) CARDINAL);
    (* for GetFrom/PutToPhysmemory, a maximum ElementSize
       (transfer count) of 8K byte is allowed, except for
       (ElementSize MOD 512 = 0) & (Offset = 0) *)

    GetRT11Word [564B]: PROCEDURE (
      (*RT11Address:*) ADDRESS, VAR (*Content:*) WORD);

    (* GetRT11Word is used to access the RMON database.
       RT11Address is RMON base address of location 54B +
       fixed offset RT11Address is any address between
       0B and 157776B of the user's space.
       Values between 160000B and 177776B address the IOpage.
       RT-11's programmed request .gval can be used alternatively
       issued via SYSTEM.RT11CALL *)

    RTSSelfTest [566B]: PROCEDURE (
      VAR (*CheckSum:*) WORD); (* not implemented in Modula-2/VRS *)
(*


  The procedures DefArray and Get-/PutElement are available
  from module VIRDATA
*)
END XMEM.
```

Appendix 2: VIRDATA

The Modula-2/VRS virtual data support module mentioned in Appendix 1. Furnished as is; use at your own risk.

```
DEFINITION MODULE VIRDATA; (* GK 1984, GD Dec-1986 *)
(*    Virtual data interface for Modula-2/VRS. Example: UseVIR.MOD
*)
  FROM SYSTEM IMPORT WORD;

  CONST
    BytesPerBlock = 100B;
    BytesPerPage  = 20000B;
    BlocksPerPage = BytesPerPage DIV BytesPerBlock;

  TYPE ArrayId; (* Array identification *)

  PROCEDURE DefArray (VAR Identification: ArrayId;
    MinIndex, MaxIndex: CARDINAL; VAR Element: ARRAY OF WORD);
  (*
    Element defines the element size which can be transfered by a single call of Get-/PutElement. The element's
    size can be 8032 Byte (BytesPerPage-BytesPerBlock) in maximum. The total virtual data size is (MaxIndex-
    MinIndex+1) * (2 + 2*HIGH(Element)) in Bytes and may be larger than 64K Byte (up-to the size of extended
    memory installed minus the space used for resident virtual code). The data is located from the top of the
    current module XMloader's MemoryLayout.Limit.PhysBlock and grows down to the end of the resident virtual
    code. If no more extended memory is available or DefArray is called more than 127 times, a "storage-error" is
    raised.

    MemoryLayout.FirstFree.PhysBlock is modified when a program is loaded and shows the first free physical
    block used by resident virtual code of the currently active program layer and may also be determined by
    looking at the programs's .MAP file generated by LINK.XML. Code is loaded from the bottom (160000B or
    .VRS file) and grows upwards.

    The amount of physical memory installed (or the size of the .VRS file under SHAREplus) is assumed in
    location 43300B of the XM*.SAV file in 64 Byte units, e.g.: a value 4000B is for 128KB (or 256 file blocks of the
    .VRS file), a value 10000B is for 256KB main memory installed, 40000B for 1MB, 100000 for 2MB and
    177600B for the full 4MB of main memory.
    The value located at 43300B is initially loaded into XMLoader's memory descriptor and is decremented in
    DefArray by the data size requested.

*)
  PROCEDURE RemArray (VAR Identification: ArrayId;
    MinIndex, MaxIndex: CARDINAL; VAR Element: ARRAY OF WORD);
  (* garbage collection not implemented *)

  PROCEDURE GetElement (Identification: ArrayId; Index: CARDINAL;
    VAR Element: ARRAY OF WORD);
  (* get the Identification[Index] data into Element *)

  PROCEDURE PutElement (Identification: ArrayId; Index: CARDINAL;
    VAR Element: ARRAY OF WORD);
```

```
  (* put the Element data into Identification[Index] *)
  (*
    Get-/PutElement: if an illegal Identification or an Element with other size as declared in DefArray is
    substituted, a "storage-error". If Index is outside the range declared with DefArray, an "index-out-of-range"
    error is raised.

  *)
  PROCEDURE RemoveArrays;
  (* removes all arrays declared and restores MemoryLayout.Limit.PhysBlock *)
END VIRDATA.
```

```
IMPLEMENTATION MODULE VIRDATA; (* Gerhard Koller, 1984 *)
(* GD Dec-1986 index arithmetic with LONGINT operations *)

  FROM SYSTEM IMPORT WORD, ADR, LMUL, LDIV, LMOD;
  FROM XMEM IMPORT PutToPhysMemory, GetFromPhysMemory, XMADDRESS;
  FROM XMLoader IMPORT LayerPositionDesc, MemoryLayout, OverlayCount;
  FROM Exceptions IMPORT Raise;
  FROM SystemTypes IMPORT ErrorType;

  CONST
    MinArrayId = 1;
    MaxArrayId = 128;
    MaxOverlay = 4;     (* assumed value! *)

  TYPE
    ArrayId = [MinArrayId..MaxArrayId];

    ArrayManagRec = RECORD
      Size,
      MinIndex,
      MaxIndex: CARDINAL;
      BlockBase: XMADDRESS;
    END;

    OverlayRange = [0..MaxOverlay];

  VAR
    ArrayManagement: ARRAY ArrayId OF ArrayManagRec;
    LastDefRec: ARRAY OverlayRange OF ArrayId;
    LastOverlay: CARDINAL;
    PhysBlockBase: XMADDRESS;
    OldLimit: XMADDRESS;

  PROCEDURE DefArray (VAR Identification: ArrayId;
    MinInd, MaxInd: CARDINAL; VAR Element: ARRAY OF WORD);
  VAR BlockLength, IndexRange: CARDINAL;
  BEGIN
    IF MaxInd < MinInd THEN Raise(IndexOutOfRange); END;
    IF OverlayCount > MaxOverlay THEN Raise(StorageError); END;
    WHILE LastOverlay < OverlayCount DO
      LastDefRec[LastOverlay+1] := LastDefRec[LastOverlay];
      INC(LastOverlay);
    END;
    LastOverlay := OverlayCount;
    IF LastDefRec[OverlayCount] >= MaxArrayId THEN Raise(StorageError);
    ELSE
      INC(LastDefRec[OverlayCount]);
      Identification := LastDefRec[OverlayCount];
      WITH ArrayManagement[LastDefRec[OverlayCount]] DO
        MinIndex := MinInd; MaxIndex := MaxInd;
        Size := 2 + 2*HIGH(Element);
        IndexRange := MaxIndex - MinIndex + 1;
        BlockLength := LDIV (LMUL (IndexRange, Size)+LONGINT(BytesPerBlock-1),
```

```
          BytesPerBlock);
        IF (Size > (BytesPerPage - BytesPerBlock)) THEN Raise(StorageError);
        END;
        WITH MemoryLayout DO
          IF BlockLength > (Limit.PhysBlock - FirstFree.PhysBlock) THEN
            DEC(LastDefRec[OverlayCount]);
            Raise(StorageError);
          ELSE
            DEC(Limit.PhysBlock, BlockLength);
            BlockBase := Limit.PhysBlock;
          END;
        END;
      END;
    END;
END DefArray;

(*$S-,$T-*)
PROCEDURE GetElement(Identification: ArrayId; Index: CARDINAL;
  VAR Element: ARRAY OF WORD);
VAR Offset: LONGINT;
BEGIN
  IF (OverlayCount > MaxOverlay) OR (Identification < MinArrayId)
    OR (Identification > LastDefRec[OverlayCount])
  THEN Raise(StorageError);
  ELSE
    WITH ArrayManagement[Identification] DO
      IF (2 + 2*HIGH(Element)) <> Size THEN Raise(StorageError);
      ELSIF (Index < MinIndex) OR (Index > MaxIndex) THEN
        Raise(IndexOutOfRange);
      ELSE
        DEC (Index, MinIndex);
        Offset  := LMUL (Index, Size);
        GetFromPhysMemory (ADR(Element), Size,
          BlockBase + CARDINAL(LDIV (Offset, BytesPerBlock)),
          LMOD (Offset, BytesPerBlock));
      END;
    END;
  END;
END GetElement;

PROCEDURE PutElement(Identification: ArrayId; Index: CARDINAL;
  VAR Element: ARRAY OF WORD);
VAR Offset: LONGINT;
BEGIN
  IF (OverlayCount > MaxOverlay) OR (Identification < MinArrayId)
    OR (Identification > LastDefRec[OverlayCount])
  THEN Raise(StorageError);
  ELSE
    WITH ArrayManagement[Identification] DO
      IF (2 + 2*HIGH(Element)) <> Size THEN Raise(StorageError);
      ELSIF (Index < MinIndex) OR (Index > MaxIndex) THEN
        Raise(IndexOutOfRange);
      ELSE
        DEC (Index,MinIndex);
```

```
            Offset  := LMUL (Index, Size);
            PutToPhysMemory (ADR(Element), Size,
              BlockBase + CARDINAL(LDIV (Offset, BytesPerBlock)),
              LMOD (Offset, BytesPerBlock));
          END;
        END;
      END;
    END PutElement; (* $S=,$T= *)

    PROCEDURE RemArray(VAR Identification: ArrayId;
      MinInd, MaxInd: CARDINAL; VAR Element: ARRAY OF WORD);
    BEGIN
      (* Garbage collection not implemented *)
    END RemArray;

    PROCEDURE RemoveArrays;
    VAR i: CARDINAL;
    BEGIN
      MemoryLayout.Limit.PhysBlock := OldLimit;
      FOR i := 0 TO HIGH(LastDefRec) DO
        LastDefRec [i] := 0;
      END;
      FOR i := MinArrayId TO HIGH(ArrayManagement) DO
        ArrayManagement[i].Size := 0;
      END;
    END RemoveArrays;

BEGIN
    OldLimit := MemoryLayout.Limit.PhysBlock;
    RemoveArrays;
    LastOverlay    := 0;
END VIRDATA.
```

Appendix 3: XMputget

The module XMputget predates Modula-2/VRS and handles only data in the extended adress space; it serves to show machine language level programming in Modula-2. This module was probably not part of the regular Modula-2 distribution kit. Furnished as is; use at your own risk.

```
DEFINITION MODULE XMputget;

(* (c) COPYRIGHT Dotzel, Erlangen, May-1982 *)

FROM SYSTEM IMPORT ADDRESS;

EXPORT QUALIFIED putXM, getXM;

 (*
 Modula-2 XM-Handler, handles up to 4MByte.

 The time to put/get data to/from XM is 35ms for 8 KByte;
 that is 230 KByte/s or 115 KWords/s on a PDP™-11/23.

 Further speed up possible in implementation module, if the
 amount of data to be transfered is fixed or a multiple of
 2 or 3 or 4 ... or x words.


 The 4 MByte address space in this module is
 divided into 8192 Blocks a 512 Bytes;

 BlockNumber range = 0..8191;
 Words range = 1..24576;

 The procedures put/get data of a user program
 under RT11SJ with or without VM.SYS into/from memory
 starting at a block boundary.
 (see also Files.DEF: Procedures ReadBlock, WriteBlock)


 Example:

    VAR WorkSpace: ARRAY [0..511] OF WORD;

    putXM( ADR(WorkSpace), 112, SIZE(WorkSpace) DIV 2);
    (* puts 512 Words of data into block 112 and 113. *)

 Bank 7 of main memory, at location 160000B,
 is normally mapped to the I/O-page starting
 at     760000B (18 bit)
 or 17760000B (22 bit).
```

```
   In the example:
     112 * 512 Bytes --> 57344 (160000B) is the start address;
     the blocks from 112 to 127 (8KBytes) are available
     in a hardware configuration with 64KBytes of main memory.
     The MMU (KT11) is required.)


  The handler also works in a 30K RT-11 environment, where the
  first virtual block number is 120. (Blocks 112 to 119 not available)
  *)


PROCEDURE putXM
   (DataPointer: ADDRESS; BlockNumber, Words: CARDINAL);
PROCEDURE getXM
   (DataPointer: ADDRESS; BlockNumber, Words: CARDINAL);

END XMputget.
```

```
IMPLEMENTATION MODULE XMputget[6] (* $T-, $S+ *);

 (* (C) COPYRIGHT Dotzel, D-8520 Erlangen, Okt 1981, May 1982, 1983.

     This software is furnished under a license for use only on a
     single computer system and may be copied only with the inclusion
     of the above copyright notice. This software, or any copies of
     it, may not be provided or otherwise made available to any other
     person except for use on such system and to one who agrees to
     these license terms. The title and ownership of the software
     remains at all times with G. Dotzel.

     The information in this software is subject to change without
     notice and should not be construed as a committment by
     G. Dotzel.

    XM handler, handles up to 4MByte (22bit mapping)
    via a system dependent Modula-2 program.

    The time to put/get data to/from XM is 35ms for 8 KByte;
    that is 230 KByte/s or 115 KWords/s,
    Data (each word) is moved from user data space to XM using MTPD
    and is moved to user data space from XM using MFPD;
    Execution time of MTPD (R)+ and SOB is
      2.85 + 1.84 + 2.62 = 7.31 ûs; (LSI-11/23)
    Execution time of MFPD -(R) and SOB is
      4.12 + 1.84 + 2.62 = 8.58 ûs; (LSI-11/23)

    (Speed up via more MTPx in the inner loop possible.)
*)

  FROM SYSTEM  IMPORT ADR, ADDRESS, WORD;


  TYPE
    MMURegister= ARRAY[0..7] OF CARDINAL;
    MNE=(mXr3,NumberofWords,mXr5,BlockOffset,mspr0,
          mXsp,DataP, haltnop,mtpd,sob,mr0sp,(*haltnop,*) rtsr7);
  VAR
    Kpdr[172300B],
    Kpar[172340B],
    Updr[177600B],
    Upar[177640B]: MMURegister;

    MMUSR0[177572B],
    MMUSR3[172516B],
    PS[177776B]:    BITSET;

    Copy: ARRAY MNE OF CARDINAL;
    CopyBlock: PROC;
    i: CARDINAL;

   PROCEDURE putXM(
```

```
      DataPointer: ADDRESS;
      BlockNumber,
      Words: CARDINAL);
    BEGIN
      Copy[NumberofWords]:=Words;
      Copy[DataP]:=DataPointer;

      (* set up offset in page: *);
      Copy[BlockOffset]:=512 *(BlockNumber MOD 16);

      (* set up User page address registers: *)
      Upar[0]:=(BlockNumber DIV 16) * 200B;
      FOR i:=1 TO 7 DO Upar[i]:=Upar[i-1] + 200B END;

      INCL(MMUSR3, 4) (* enable 22 Bit addressing mode *);
      (* set previous mode to User mode, current mode is kernel mode *);
      PS:={5,6,7,12,13};

      INCL(MMUSR0,0) (* enable MMU *);
      CopyBlock (* start machine code routine *);
      MMUSR0:={} (* disable MMU *);
    END putXM;


    PROCEDURE getXM(
      DataPointer: ADDRESS;
      BlockNumber,
      Words: CARDINAL);
    BEGIN
      (* modify put routine: *)
      (* MFPD -(R5); Pushes word -(R5) onto current stack *);
      Copy[mtpd]:=106545B;

      Copy[NumberofWords]:=Words;
      Words:=Words * 2 (* get number of bytes *);
      Copy[DataP]:=CARDINAL(DataPointer) + Words ;
      Copy[BlockOffset]:= 512 * (BlockNumber MOD 16) + Words;
      Upar[0]:=(BlockNumber DIV 16) * 200B;
      FOR i:=1 TO 7 DO Upar[i]:=Upar[i-1] + 200B END;
      INCL(MMUSR3,4);
      PS:={5,6,7,12,13};
      INCL(MMUSR0,0);
      CopyBlock;
      MMUSR0:={};

      Copy[mtpd]:=106625B (* restore original value for putXM *)
    END getXM;

BEGIN
  (* start address of the copy routine *)
  CopyBlock:=PROC(ADR(Copy));
  FOR i:=0 TO 7 DO
    Kpdr[i]:=77406B;
    Updr[i]:=77406B;
```

```
    (* set up Kernel page address registers: *)
    Kpar[i]:= i*200B;
  END;
  Kpar[7]:=177600B;

  (* set up copy routine:

     MOV  #NumberofWords,R3;
     MOV  #BlockOffset,R5;
     MOV  SP,R0;
     MOV  #DataP,SP;
     LOOP
        MTPD  (R5)+;
        SOB  R3
     END;
     MOV  R0,SP;
     RTS  PC;                    *)

  (* set up machine code; the X's are inserted by putXM *)
  Copy[mXr3]:=012703B;
  Copy[mXr5]:=012705B;
  Copy[mspr0]:=010600B;
  Copy[mXsp]:=012706B;
  Copy[mtpd]:=106625B;  Copy[sob]:=077302B;
  Copy[mr0sp]:=010006B;
  Copy[haltnop]:=000240B;
  Copy[rtsr7]:=000207B;
END  XMputget.
```

Appendix 4: VN.SYS

If the above mentioned virtual memory driver XS.SYS is not the same as VN.SYS, the source code of XS.SYS is probably lost, since I have no longer access to any 8" floppy drive under RT-11. The only driver I found is another bootable virtual memory driver, called VN.SYS in binary form only, which might or might not be identical to XS.SYS but has the same properties as XS.SYS, except for - if it predates XS.SYS - the configurable virtual start address which would be needed to work in parallel with Modula-2/VRS; its size is only 3 (512 byte) blocks; so if any extension would be necessary, it could be disassembled easily.

Below is the octal dump listing of VN.SYS (© Copyright (1982) by G. Dotzel, Modulaware.com), which replaces RT-11SJ's VM.SYS on systems, where the main memory board is configured in non-standard way for short (4KB) I/O page, but it should also work with 8KB I/O page.

VN.SYS is bootable; it requires a setup for extended memory's start address and/or size as shown in the attached instructions - if the driver corresponds to these instructions. I'm not sure if this VN.SYS works in parallel with Modula-2/VRS.

Requirements: a PDP/LSI-11 processor which implements the MTPD and MFPD instructions (standard on e.g.: LSI-11/23 and LSI-11/73). Furnished as is; use at your own risk.

VN.SYS

VRS-Compatible & Bootable Device Handler

VN.SYS is now **VRS compatible** and bootable. VN.SYS is used just like DEC® VM.SYS, i.e. it can be initialized using .INIT VN and it can be used as system device also. Due to VRS restrictions, data transfer is always performed with high processor priority. This means that a large transfer word count can affect the system time (clock) especially with the slower LSI-11/23. VN.SYS also operates on LSI-11/73. On LSI-11/23 the MMU option (usually default) is required. VN.SYS can't be used with the LSI-11/2 processor.
Note, using a standard file structured device is the only portable way when writing Modula-2 software under RT-11. Modula-2's VRS module XMEM is *not* portable and not much faster than VN.SYS. Furthermore XMEM is limited to 8K byte data buffer size per transfer.
VN has a selectable start and stop address for memory:

Default memory size: 1M byte.

Default VN start address: 256K byte or 1000000B (memory range from 56K byte (160000B) to 256K byte (777776B) is reserved for VRS).

Default VN stop address: 1M byte byte or 2777776B. 22 bit addressing (Q-22 bus) supported for memory up to 4M byte.

Furthermore, VN.SYS provides virtual memory access on RT-11 V04/V05 or STAR-eleven operating systems, not only for standard *28K_Word* Q/Q22-Bus RT-11 systems, but also for *30K_Word* systems (i.e. 4K Byte I/O-page). And it's bootable.

Configuration of stop address (defines device size) and start address:

VN.SYS can be configured applying a simple patch procedure. Type the underlined commands.

256K byte memory / start address at 128K byte: device size 256 blocks
.*PATCH SY:VN.SYS*
**54/* xxxxxx *400* (* device size *)
**1022/* xxxxxx *400*
**1036/* xxxxxx *4000* (* start address offset*)
**2022/* xxxxxx *4000*
**E*
.

1M (1024K) byte memory / start address at 128K byte: device size 1792 blocks
.*PATCH SY:VN.SYS*
**54/* xxxxxx *3400* (* device size 3400B = 1792. *)
**1022/* xxxxxx *3400*
**1036/* xxxxxx *4000* (* start address offset (address 400000B) *)
**2022/* xxxxxx *4000*
**E*
.

1M (1024K) byte memory / start address at 256K byte: device size 1536 blocks
.*PATCH SY:VN.SYS*! or use VN1MB.SYS
**54/* xxxxxx *3000* (* device size 3000B = 1536. *)
**1022/* xxxxxx *3000*
**1036/* xxxxxx *10000* (* start address offset (address 1000000B *)
**2022/* xxxxxx *10000*
**E*
.

2M (2048K) byte memory / start address at 256K byte: device size 3584 blocks
.*PATCH SY:VN.SYS*! or use VN2MB.SYS
**54/* xxxxxx *7000* (* device size 3000B = 3584. *)
**1022/* xxxxxx *7000*
**1036/* xxxxxx *10000* (* start address offset (address 1000000B *)
**2022/* xxxxxx *10000*
**E*
.

```
Dump of file _DUB3:[000000.M2KIT]VN.SYS;1 on  5-JUN-2008 04:06:55.91
File ID (9690,1,0)   End of file block 3 / Allocated 3

Virtual block number 1 (00000001), 512 (0200) bytes

 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000000
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000020
 100377 003400 000570 002566 000000 000000 001000 000001 ........v.x..... 000040
 000000 000000 000000 000000 000214 001000 001570 000000 ..x............. 000060
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000100
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000120
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000140
 177572 000000 000000 000000 000000 000000 000000 000000 ..............z. 000160
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000200
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000220
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000240
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000260
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000300
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000320
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000340
 000000 000000 000000 000000 000000 000000 000000 000340 à............... 000360
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000400
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000420
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000440
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000460
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000500
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000520
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000540
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000560
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000600
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000620
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000640
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000660
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000700
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000720
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000740
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000760
```

```
Dump of file _DUB3:[000000.M2KIT]VN.SYS;1 on  5-JUN-2008 04:06:55.91
File ID (9690,1,0)   End of file block 3 / Allocated 3

Virtual block number 2 (00000002), 512 (0200) bytes

 012302 177772 016703 000000 000000 000340 000556 000000 ..n.à.....Ã.ú.Â. 000000
 004000 062702 006302 006302 006302 003543 003400 022702 Â%..c.Â.Â.Â.Âe.. 000020
 030340 012737 177776 013725 000410 062705 010705 005004 ..Å.Åe..Õ. .ß.à0 000040
 012700 177640 012701 177572 005037 177572 013725 177776  .Õ.z...z.Á...À. 000060
 011125 172500 016125 172440 016125 177740 016125 000010 ..U.à.U. õU.@õU. 000100
 172500 010461 172440 077406 012761 177740 077406 012761 ñ...à.ñ... õ1.@õ 000120
 177656 177600 012737 077024 000200 062704 060221 010411 ...`Äe...~ß... . 000140
 100417 011304 012306 005723 010600 172516 000020 052737 ßU..Nõ..Ó.Æ.Ä... 000160
 005204 177572 000001 052737 060506 006305 010405 001461 1...Å.FaßU..z... 000200
 005404 000000 000444 077403 106545 106545 103001 006204 ....e.e...$..... 000220
 106625 103001 006204 005204 177572 000001 052737 005005 ..ßU..z......... 000240
 177572 106637 177744 062706 010706 010601 077403 106625 ......Æ.Æeä...z. 000260
 000001 052737 177724 062706 010706 001417 111304 010106 F.Ä...Æ.ÆeÔ.ßU.. 000300
 052753 005743 000404 001374 105304 005746 106625 177572 z...æ.Ä.ü...ã.ëU 000320
 106637 177662 062706 010706 010601 010006 000435 000001 ........Æ.Æe ... 000340
 012700 177640 012701 000072 062705 010705 010106 177572 z.F.Å.Åe:.Á...À. 000360
 012521 172500 012561 172440 012561 177740 012561 000010 ..q.à.q. õq.@õQ. 000400
 010704 177776 000020 016737 177572 000030 016737 077010 .~ß...z.ß... .Ä. 000420
 000000 000000 000270 000175 000054 013705 177350 062704 Äeè Å.,.}. ..... 000440
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000460
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000500
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000520
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000540
 000000 000000 000501 000240 000000 000000 000002 000000 ..........A..... 000560
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000600
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000620
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000640
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000660
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000700
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000720
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000740
 000400 116020 041420 000000 000000 000000 000000 000000 ...........C.... 000760
```

```
Dump of file _DUB3:[000000.M2KIT]VN.SYS;1 on  5-JUN-2008 04:06:55.91
File ID (9690,1,0)   End of file block 3 / Allocated 3

Virtual block number 3 (00000003), 512 (0200) bytes

 006302 006302 006302 010105 010002 010203 000410 000137 _.......E.Â.Â.Â. 000000
 012761 000010 012700 177640 012701 005004 004000 062702 Âe....Á...À...ñ. 000020
 010411 172500 010461 172440 077406 012761 177740 077406 ..à.ñ... õ1.@õ.. 000040
 052737 172356 177600 012737 077015 000200 062704 060221 .`Äe...~ß...îôßU 000060
 010504 010306 177776 030340 012737 010600 172516 000020 ..Nõ..ß.à0 .Æ.D. 000100
 012737 177572 000001 052737 060506 006305 010405 001417 ....Å.FaßU..z.ß. 000120
 077403 106545 106545 103001 006204 005204 172356 001600 ..îô......e.e... 000140
 000207 000241 177776 005037 177572 005037 000005 010006 ......z... .¡... 000160
 001000 012702 002000 012701 000002 012700 010000 012706 Æ...À...Á...Â... 000200
 004716 114370 012737 004730 000214 012737 177560 004767 œ.p.ß...Ø.ß.ø.Î. 000220
 000000 000000 000000 000000 001000 000137 004722 005037 ..Ò._.......... 000240
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000260
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000300
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000320
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000340
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000360
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000400
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000420
 000002 004167 000000 000000 000000 000000 000000 000000 ............w... 000440
 000022 004167 012100 000030 004167 000746 012700 000766 ö.À.æ.w...@.w... 000460
 112037 000776 000000 000005 000012 004167 000762 012700 À.ò.w....... ... 000500
 005015 000201 001371 105710 100375 177564 105737 177566 v.ß.t.ÿ.È.ù..... 000520
 027511 000012 005015 000200 026525 026524 047517 041077 ?BOOT-U-......I/ 000540
 000000 000000 000000 000000 000162 067562 071145 020117 O error......... 000560
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000600
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000620
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000640
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000660
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000700
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000720
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000740
 000000 000000 000000 000000 000000 000000 000000 000000 ................ 000760
```