RJ 10338 (A0502-006) February 4, 2005
Computer Science

IBM Research Report

UVC: A Universal Virtual Computer for Long-term Preservation of Digital Information

Raymond A. Lorie
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

Raymond J. van Diessen
IBM Business Consulting Services
Transistorstraat 7
Almere, 1322 CJ, Netherlands

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

UVC: A Universal Virtual Computer for Long-Term Preservation of Digital Information

Raymond A. Lorie

IBM Research Division
Almaden Research Center
San Jose, CA 95120-6099

Raymond J. van Diessen

IBM Business Consulting Services
Transistorstraat 7
Almere, 1322 CJ, Netherlands

ABSTRACT

The preservation of digital data for the long term presents a variety of challenges. But one of the most difficult of these challenges is to maintain the interpretability of files created long ago. A sequence of bits is meaningless if it cannot be decoded and transformed into an intelligible representation. Since the decoding and transformation are generally specified through a computer program, the loss of that program translates into a loss of valuable information.

In 2000, a project in IBM Research proposed the use a Universal Virtual Computer (UVC) to specify today a process that will be executed on a – still unknown – machine of the future. The UVC is a general purpose computer, complete yet basic enough as to remain relevant for a very long time. The method consists of archiving, with any file of a specific data format, a program P which can decode the data and return the information to a future client, according to a logical view (a la XML). The novelty is that P is written for the UVC. In the future, the only thing required is an emulator of the UVC, which will be able to run the program P and return all data according to an easy to understand, logical view.

The initial specifications of the UVC were published in 2001 [1]. This report describes a more advanced – and, in some respects, simpler – version of the UVC, that has been fully implemented and is now considered to be the UVC Convention, version 0.

1. Introduction: the preservation problem

Preserving bit strings (or files) is a basic requirement of any data processing system. The problem has existed since the beginning of the computer era. The classical solution consists of copying the information periodically from an old medium to a new one. This solves both medium and device obsolescence problems at the same time. In addition, it takes advantage of the new technologies providing higher data densities at lower costs. Today, the tremendous increase in data volume has triggered many developments in storage hardware and storage management systems. New approaches are being introduced at the research and product levels. But being able to read the bits correctly in the future is only one part of the solution.

Another challenge concerns the interpretability of files created long ago. A sequence of bits is meaningless if it cannot be decoded and transformed into an intelligible representation. Since, the decoding and transformation are generally specified through a computer program, the loss of that program translates into the loss of valuable information. In the past - and present - the industry managed to avoid disasters (in most cases) by periodically converting data to newer formats and rewriting programs. But such conversion methods come with many dangers and they will not be able to support the huge increase in multimedia data that applications will want in the future.

Alternatives have been proposed but none has been implemented in full-fledged systems.

For example, a lot of attention has been given to the use of XML as a technology-independent format. The idea is to extract from the original format all data elements and to tag them appropriately. The complete information is then encoded in an XML string. The argument is that the XML string will remain readable in the future. It must be noted that identifying the data elements is not sufficient. The semantic information on what to do with each element must still be specified as metadata[1]. However, the presence of tags facilitates the association of semantic information with the data elements. A drawback of the technique is that XML may not be the most appropriate storage format for certain document types, since it brings with it a potentially huge increase in file size.

On the other hand of the spectrum, the emulation of old machines makes it possible to run old software on new machines. This may be both good and bad: good because it will run the old program and present the data as seen in the past; bad because if forces the user to go through old interfaces and does not allow for reusing the data in different ways, making re-purposing all but impossible.

This paper refers to a different approach, using a Universal Virtual Computer (UVC) to specify processes that have to run in the future. The method was initially proposed in [1] and reference [2] described an initial version of the UVC architecture. This paper describes the version that has been fully implemented as part of a joint study between IBM and the KB (Koninklijke Bibliotheek, the National Library of the Netherlands) [3].

[1] Many of the current applications provide some form of functionality to translate their internal proprietary formats into an XML format. However, the tags and data elements are often cryptic and still heavily dependent on the semantics encoded in the application.

2. The Universal Virtual Computer (UVC) solution.

A virtual computer can be used to specify today processes that will need to be executed on a future, still unknown, real machine. The only requirement is that a UVC emulator be always available (a relatively simple task compared to an emulator of a real machine.)

When a file of a certain format F is archived, a program P is also archived which can decode the data and return the information to a future client, according to a logical view (a la XML). The definition of the logical view can be archived in a similar way. The novelty is that P is written for the UVC. In the future, the only thing required is an emulator of the UVC, which will be able to run the program P and return all data in an easy to understand, logical view. The UVC is a general purpose computer, complete yet basic enough as to remain relevant for a very long time. Given the simplicity of the UVC, writing a program that emulates it is reasonably easy.

This method provides all the advantages of XML. But it has the additional – and very significant - advantage of keeping the data in its original form instead of forcing an XML representation for the stored document. In principle, keeping the original format should always be possible. For example, for PDF it is possible to write a decoder of the PDF format. But it may sometimes be simpler to immediately convert the original format, once, into another one, and then preserving that representation as if it were the original. The choice of an "intermediate format" must be guided by a good balance between efficiency of storage and decoding complexity. The same logical view can often be used for a multitude of internal formats (for example, one logical view would support a color image independently of its original format). This greatly simplifies the writing of Restore Applications in the future.

3. The system components

The design goal for the UVC was not to define a minimal general-purpose computer. Instead, the idea was to develop an intuitive computer with powerful and flexible instructions for handling bit strings, and to take advantage of the fact that it is virtual and that performance is of secondary importance. The architecture relies on concepts that have existed since the beginning of the computer era: memory, registers, basic instructions, without secondary features often introduced for improving the execution performance and the memory usage. It also tries to be natural; for example, a negative number is a positive number with a sign (no 2-complement); there is no notion of byte, simply bits that can be used at will.

For ease of explanation, we consider the various components shown in Fig. 1. The emulator and the new Restore Application are both written in the future. The UVC being emulated is composed of a CPU that executes instructions, a status (conditional flag and error indicator), an instruction counter, and a memory. All of this is part of the UVC Convention and will not change in time[2]. The UVC program is a sequence of UVC instructions in machine language, with very little glue that holds the various pieces together (this is also part of the Convention). The instruction set includes an Input/Output facility to exchange data between the UVC emulator and the Restore Application (through explicit I/O instructions in the UVC program).

[2] It would be unrealistic to claim that the proposed UVC will never change. The version being described here is Version 0. Any archived document will always contain such a UVC version number to accommodate for rare modifications.

The next sections go through these components in detail.

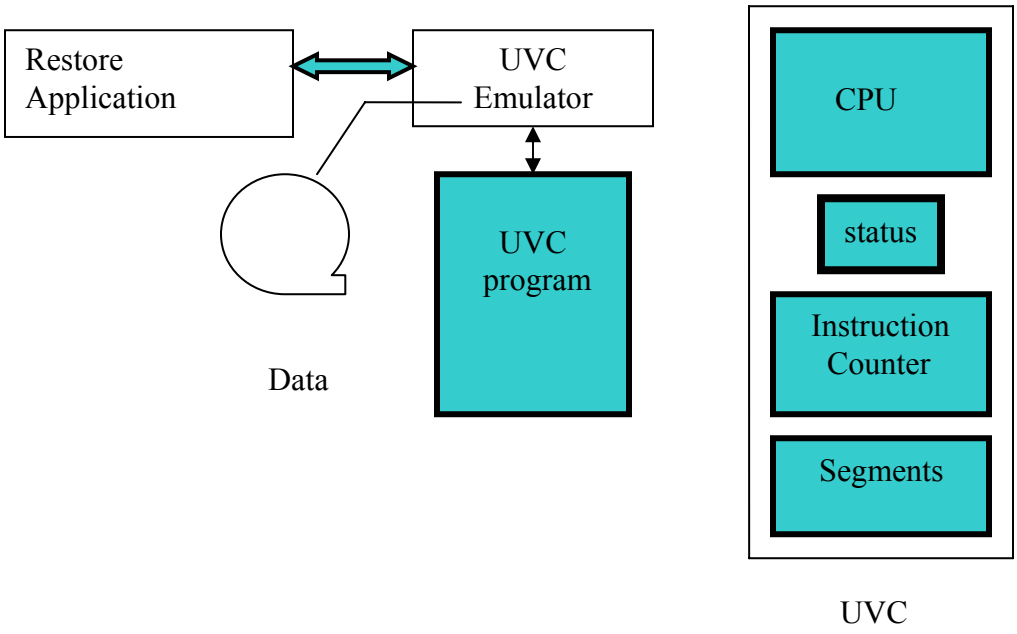

Figure 1: Global architecture

3.1 The memory model

The memory model is that of a segmented store. A segment contains an arbitrarily large set of *registers* and a bit-addressable *sequential memory*. A register may contain a value or a pointer to a particular bit of the sequential memory (actually the displacement from the beginning of the segment). There is an unlimited number of registers and registers are of unlimited length. An integer value occupies as many bits as necessary at the right of the register. The sign is a separate bit (0 for plus, 1 for minus). The UVC itself maintains the length of the value internally. The only operations on the sequential memory of a segment are designed to move information from the memory to registers and vice-versa or to communicate with the external world. Note the bit – rather than byte or word – orientation; this is very convenient for handling bit streams and it increases independence from real machine architectures. A segment is uniquely identified by a *Physical Segment Number*.

A UVC program is composed of sections that interact between themselves. Each section is stored in a segment. An individual section can address all segments that are in its *address space* (conversely, the segments *belong* to the section). Each segment is uniquely identified by a *Logical Segment Number*. During an execution, the UVC emulator will assign physical segments to logical segments and will keep track of the mapping. The address space of a section always contains segments 0, 1, and 2, plus a segment that contains the section code, plus an arbitrary number of segments containing variables and data. If a section is called recursively, the code is stored only once, but each instantiation will have its own address space.
The different types of segments are now explained (Note that, in the remainder of this publication, all segment numbers are logical segment numbers).

Segment 0
Segment 0 is accessible by any section (it belongs to all address spaces). It contains a collection of shared constants and variables. There is a mechanism to load the constants initially (see *archive module*, below).

Segment 1
The segment is accessible by the section to which it belongs. If the section is invoked recursively, segment 1 can be addressed by all instantiations; it acts as a shared memory to communicate between multiple activations of the same section.

Segment 2
There is one such segment per address space; its function is to handle data exchange during invocations (calls) among sections. If a section A invokes a section B, A will see the results of B in A's segment 2. (The UVC will adjust the mapping, avoiding a need to copy the results from one segment to another).

Segment 3 to 999
Segments in that range are shared.
They belong to the address spaces of all sections. For example, if two sections refer to a segment 4, both see the same segment 4.

Segments >= 1000
Segments in that range are private. If two sections refer to segment 1000, they see different segments; the emulator maps them onto different physical segments. When the section is invoked recursively, each invocation will see its own instance of a segment in that range.

In the remainder of this paper, Reg1 stands for a pair (s1, r1), denoting the content of register r1 in segment s1. As mentioned above, s1 is a logical segment. The content of a register may be a value or an address (a displacement) in the sequential memory, if that is what the instruction expects.

In any instruction, the specification of a register R may include an indirection flag (R*). When the emulator encounters a register R*, the content of R* is a register number which identifies the register containing the operand. The various cases are illustrated in Fig. 2.

3.2 Status indication

The status of the UVC after an operation is conveyed by the following indicators:

1. The *condition flag* reflects the result of the last comparison operation. It can only be changed by a comparison instruction.

2. The *error indicator* is a 32-bit number identifying the error that occurred. The possible values are determined by the specific implementation of the UVC emulator and are useful for debugging that implementation.

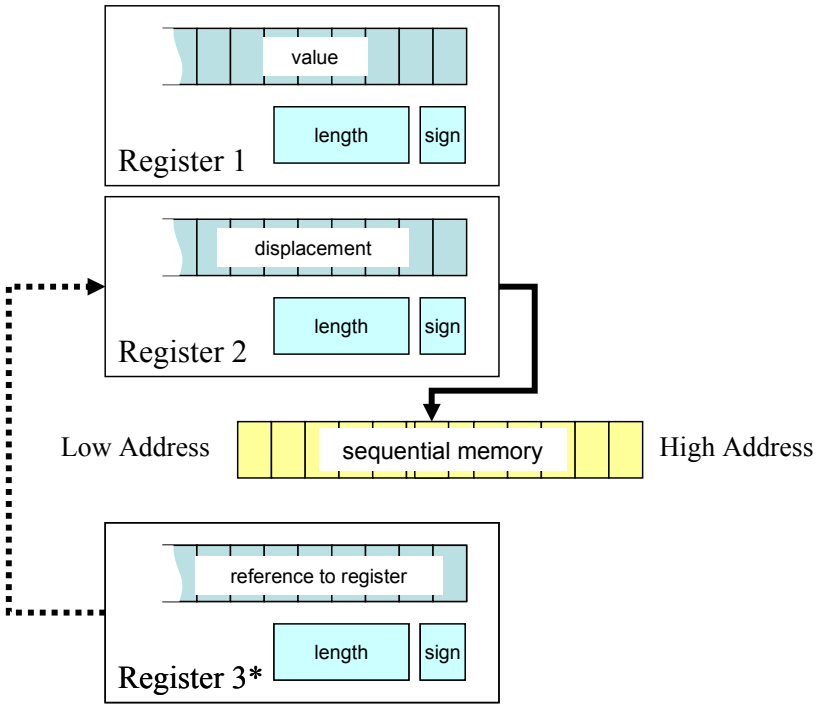

Figure 2: The possible contents of registers in UVC instructions

3.3 Instruction Counter

The instruction counter contains the address of the next instruction. This address is updated by the CPU each time an instruction completes. The address is composed of two integers (32 bits): the segment number of the segment containing the section, and the bit-displacement of the instruction inside that segment. Most instructions update the counter to the address of the bit that immediately follows it in the sequential memory. A few instructions modify the counter in a different way, altering the sequential flow (see individual instructions, below).

3.4 Instruction formats

The majority of instructions have the same format: an 8-bit operation code (*opcode*) followed by zero, one, two, or three 64-bit strings. Each 64-bit string designates a register: the first 32 bits identify a segment; the next 32 bits are decomposed into an indirection flag and a 31-bit value that identifies a register in that segment.

In the example that follows, the opcode expects two operands: Reg1 and Reg2. The specification of an operand is a 64-bit string, composed of a segment number (seg) and a register number (reg) with its indirection flag. If that flag is 0, the register contains the operand; if it is 1, the register contains the number of the register (in the same segment) that contains the operand (the lengths are shown in parentheses):

op (8 bits)	segment (32 bits)	flag (1 bit)	register (31 bits)	segment (32 bits)	flag (1 bit)	register (31 bits)

Only a very few instructions have formats that differ from the one shown above. A list of all instructions and their operands is provided in the next section

3.5 Instruction set

The UVC instructions are shown in the various following sections, arranged in some meaningful groups. The binary values corresponding to each opcode are given in the summary table in Appendix B.

A general note on bit order semantics and bit order transfer:

In a register, the rightmost bit is the least significant. When the register length is automatically updated as the result of an operation, leftmost bits appear or disappear (the length is updated accordingly). When a register is transferred to memory, the leftmost bit of the register is copied at the bit position indicated in the instruction; the next bit in the register is copied to the next bit (higher bit address) in the memory, etc. The inverse operation is clearly defined. The process is graphically depicted in Fig 3.

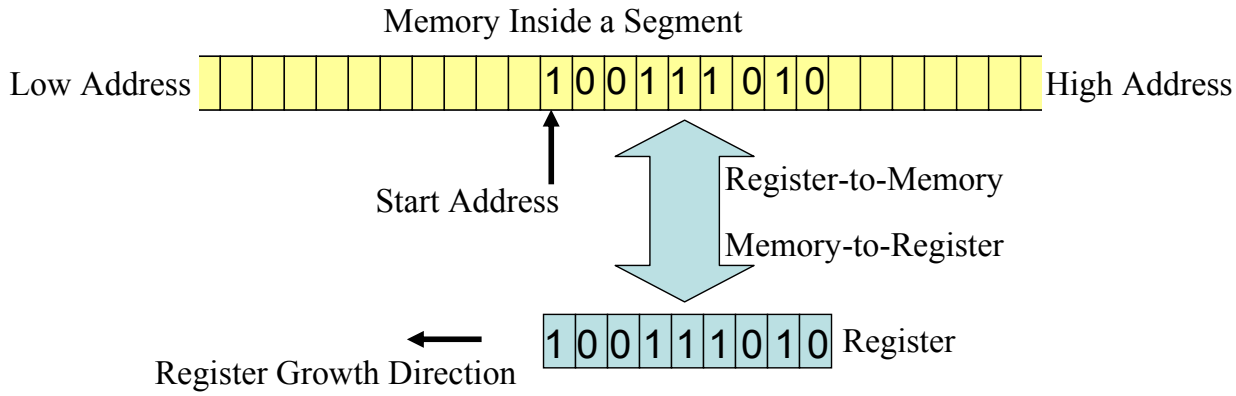

Figure 3: Bit Order Semantics

When data is transferred from memory to the communication channel (details below), the bit at the address specified in the instruction is sent first, then the bit at that address+1, +2, etc. The inverse operation stores the first bit received on the channel at the address supplied in the instruction, the next bit at the address+1, etc.

Instructions to move data between the memory and the registers.

load Reg1, Reg2, Reg3,
Load from memory to register.
Insert into Reg1 a k-bit string from memory, starting at address in Reg2; the length k is in Reg3.
The length of Reg1 is set to k.

store Reg1, Reg2, Reg3
Store from register into memory.
Store the rightmost k bits from Reg1 into memory at address in Reg2.
The length k is given in Reg3.

lsign Reg1, Reg2

Load sign.
Set the sign of Reg1 to 0 (or 1) if the single memory bit at address in Reg2 is 0 (or 1).

ssign Reg1, Reg2

Save sign.
Set the memory bit at address in Reg2 to 0 (or 1) if the sign of Reg1 is 0 (or 1).

Operations on registers only

loadr Reg1, Reg2

Load register.
Copy the content of Reg2 into Reg1 (including the sign).
After the operation, the lengths of Reg1 and Reg2 are identical.

psign Reg1

Set sign to positive.
Set sign of Reg1 to 0.

nsign Reg1

Set sign to negative.
Set sign of Reg1 to 1.

loadc Reg1, k, string

Inserts in Reg1, right justified, the k bits of the given string;
k is a 32-bit integer denoting the length of the string.
The length of the register is set to k. The sign of Reg1 is unaffected.

Since some of instructions change the length of a register, the following instruction provides access to the length.

rlen Reg1, Reg2

Get register length.
Store the length of Reg2 into Reg1. [3]

Numeric instructions

These instructions may cause the lengths of registers to change to accommodate the result. The sign is set to the result sign; if the result is zero, the sign is set to 0 (positive). All numeric instructions are performed according to the laws of binary arithmetic.

add Reg1, Reg2

Addition
The sum of the values in Reg1 and Reg2 is computed and stored in Reg1.
The size of the register r1 is set to the order of the leftmost 1 bit in Reg1.

---

[3] If an operand is expected to be a 32-bit integer but is actually shorter, it is padded with zero's on the left; if it is actually larger, an error condition is raised.

subt Reg1, Reg2

Subtraction
The value in Reg2 is subtracted from the value in Reg1 and the result is stored in Reg1.
The size of the register r1 is set to the order of the leftmost 1 bit in Reg1.

mult Reg1, Reg2

Multiply
The product of the values in Reg1 and Reg2 are computed and stored in Reg1.
The sign of Reg1 is also updated.
The size of Reg1 is set to the order of the leftmost 1 bit in Reg1.

div Reg1, Reg2, Reg3

Divide
The division of the value in Reg1 by the value in Reg2 is computed;
the quotient is stored in Reg1.
The size of Reg1 is set to the order of the leftmost 1 bit in Reg1.
The remainder is stored in Reg3. The sign of Reg1 is also updated,
and the sign of Reg3 is set to the original sign of Reg1.

Comparison instructions

grt Reg1, Reg2

Greater than
Set the condition flag to 1 if the value in Reg1 is larger than the value in Reg2.
(the signs are taken into account).

equ Reg1, Reg2

Equal
Set the condition flag to 1 if the value in Reg1 is equal to the value in Reg2;
(the signs are taken into account).

Logical instructions

not Reg1

Negation
All bits are inverted in Reg1.
The post-execution length of Reg1 is the same as the pre-execution one.

or Reg1, Reg2

Or
The bits in Reg1 and Reg2 are or'ed bit by bit and the result is stored in Reg1.

- If the pre-execution length of Reg2 is less than that of Reg1, Reg2 is virtually left-padded with 0's (the post-execution length of Reg2 remains unchanged).
- If the pre-execution length of Reg2 is larger than that of Reg1, Reg1 is left-padded with 0's (the post-execution length of Reg1 becomes equal to the length of Reg2).

and Reg1, Reg2

And

The bits in Reg1 and Reg2 are and'ed bit by bit and the result is stored in Reg1.

- If the pre-execution length of Reg2 is less than that of Reg1, Reg2 is virtually left-padded with 1's (the post-execution length of Reg2 remains unchanged).
- If the pre-execution length of Reg2 is larger than that of Reg1, Reg1 is left-padded with 1's (the post-execution length of Reg1 becomes equal to the length of Reg2).

Instructions that alter the flow of execution

br Reg1

Branch to a given address,

Set the instruction pointer to the displacement of the target instruction in the same section.

brc Reg1

Branch conditionally.

The instruction acts as the previous one (br) if the condition flag is on.

Otherwise, the execution proceeds sequentially.

call Reg1, Reg2, Reg3

Call a subroutine (another section).

Invoke code section identified by Reg1;

Reg2 identifies the starting address of the first instruction to be executed;

Reg3 identifies the segment used to submit parameter(s).

break (no operand)

Return control to the calling section at instruction following the call.

stop (no operand)

Stop execution and return the emulator to its initial state, waiting for a new input.

The sequential memory can be initialized by first loading a value in a register and then storing the register value in the memory through a store instruction. Of course, a register can also be initialized by loading an already initialized value from the sequential memory.

Communication with the outside world (I/O)

The communication makes use of a simple channel abstraction. The abstract channel behaves as a half-duplex communication channel. Any "message" of data traveling on the channel is composed of three components:

1) *Message Type (a 32-bit integer)*
   The types and their semantics are chosen by the application. They identify the different roles of the data being transferred. It may be a tag for a piece of data or simply a code that is used for synchronization between the UVC program and the application.
2) *Message Length (a 32-bit integer)*
   It is the length of the data being transferred (in bits).

3) *Message Body*
It is the actual bit string to be transferred.

There are two UVC instructions to interact with the communication channel: *In* and *Out.*

In Reg1, Reg2, Reg3
The contents of Reg1 and Reg2 before the operation are irrelevant.
Reg1 will be set to the message type received.
Reg2 will be set to the length of the message
Reg3 specifies the starting memory address where the data will be stored.

Out Reg1, Reg2, Reg3
Reg1 contains the message type.
Reg2 contains the length of the data to be transferred.
Reg3 contains the starting memory address where the data resides.

The UVC convention does not impose any additional requirement on the use and/or specific implementation of the channel. For instance, if the piece of data to be transferred is larger than the maximum allowed, it can be split into multiple messages; the exchange can be controlled by introducing message types such as message start, message continuation and message end. Similarly, synchronization between the UVC emulator and the outside application can be established by sending specific user defined messages types with no data (When the message length is zero the memory address specified in Reg3 is ignored).

Clearly, the UVC convention requires a half-duplex abstract channel that must be enforced by all specific implementations. Figure 4 illustrates the only two valid communication patterns.

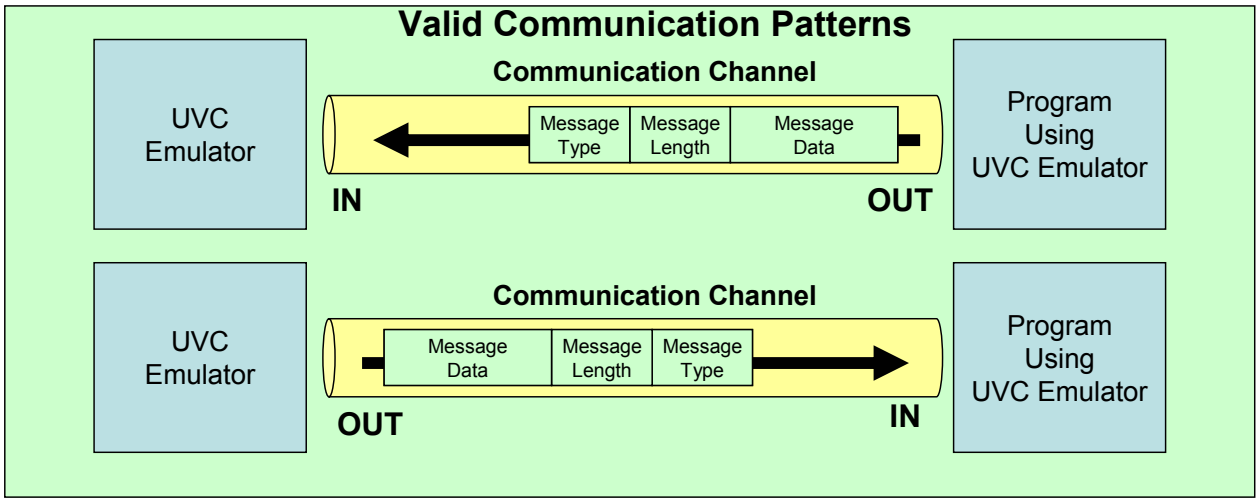

Figure 4: Valid communication patterns

An invalid communication patterns should raise an error flag in the UVC emulator; the emulator will interrupt the execution and notify the application. As mentioned before, the UVC Convention does not enforce any specific implementation of the channel: actual transfer can occur when both sides are executing their respective IN or OUT operations or the message can be temporarily buffered.

3.6 Organization of the archived module

A program is built out of multiple sections that call each other. Sections can be written independently, using symbolic names for segments. A classical compile and link process transforms these symbolic references into segment numbers that obey the numbering conventions described above. The module is then obtained by concatenating several items as governed by the following hierarchical structure:

Stream:	Nconstants, Constant*, Nsections, Section*
Constant:	Reg#, Sign, Length, string
Section:	Segment#, Length, Code

Nconstants (a 32-bit integer) is the number of constants that have to be loaded in segment 0; it is followed by Nconstants structures of type Constant, followed by Nsections (a 32-bit integer), the number of program sections to be loaded, followed by Nsections structures of type Section .

The structure of type Constant is the concatenation of a register number (a 32-bit integer), a sign bit, a string length L (a 32-bit integer), and the L-bit string itself.

A structure of type Section is the concatenation of a segment number (a 32-bit integer), the length of the code in bits (a 32-bit integer), and the code itself. The segment number indicates in which segment the section code must be loaded. By definition, the first section in the archived module is the starting section of the program.

3.7 Remarks

The UVC described in this report has been fully implemented. The implementation is being used at the National Library of the Netherlands (KB) [5] and by a publicly available demo on the IBM AlphaWorks site [6].

An important question is “how can we be sure that an archived program is bug free?”. There are two correctness issues. The first one is making sure that a given UVC implementation is correct. To that effect, an extensive UVC test program has been developed that covers the whole functionality of the machine:

- Initial loading
  Testing whether sections and global constants are loaded correctly in the UVC
- Activation of sections
  Invoking a section, passing arguments and results, and returning control to the invoking section.
- Bit addressability
  Extensive testing of different bit oriented and segment/memory mechanism memory.
- Instruction Set
  Extensive testing of all instructions, making sure that no unwanted side effect (such as overflow) occurs.

- Communication
  Testing the behavior of the communication channel through the usa of the IN and OUT instructions.

The second correctness issue concerns the UVC program itself. Here, more than an absolute correctness, we need to "proof" that the program handles correctly all instances actually ingested in the system. An exhaustive approach consists of applying the program to all instances, at ingest time or at least before obsolescence of the original viewer or application. In some cases, the results can be compared automatically with those obtained originally. If the UVC program is interactive, exhaustive checking is not possible; then the best verification techniques available at that time should be used, together with checking the results for a reasonably large sample of instances.

Finally, this paper was supposed to be exclusively a "Principle of Operation" manual. To learn more about the preservation issues in general and more particularly about the UVC approach, the reader should consider the references [1], [2] and [4], and for particular applications, [3] and [5].

3.8 Acknowledgments

To Henry Gladney for his useful comments on a draft of this paper.

To the technical staff from IBM who developed the "proof of concept" and the supporting tools: Sidney Huiskamp, Paul Brunckhorst, Jeffrey van der Hoeven, and Nanda Kol.

To Johan Steenbakker, Hilde van Wijngaarden and Erik Oltmans from the KB, and Jacqueline Slats and Remco Verdegem from the ICTU – "Testbed Digitale Bewaring", for providing concrete testing environments and feedback.

3.9 References

[1] Lorie, R. A.: *Long term preservation of digital information*. Presented at JCDL 2001, Roanoke, VA, 2001.

[2] Lorie, R. A.: *A Methodology and System for Preserving Digital Data*. Presented at JCDL 2002, Portland, Oregon, 2002.

[3] Lorie, R. A.: *The UVC: A Method for Preserving Digital Documents: Proof of Concept*. The Hague, IBM and Koninklijke Bibliotheek (KB), 2002. www.kb.nl/hrd/dd/dd_onderzoek/reports/4-uvc.pdf

[4] Gladney, H. M. and Lorie, R. A. *Evidence after Every Witness is Dead*, ACM Trans. Office Information Systems 22(3), 2004.

[5] van Wijngaarden, H. and Oltmans, E.: *Digital Preservation and Permanent Access: the UVC for images.* Proceedings of IS&T Archiving Conference, San Antonio, TX., 2004.

[6] IBM Alphaworks emerging technologies, Digital Asset Preservation tool. Available at: www.alphaworks.ibm.com/tech/uvc

Appendix: An example

In many applications of the UVC technology, the function of the UVC program will be to decode the internal format of a file and return the results according to a predefined logical view. In more general cases, the UVC program can also implement any arbitrary logic, using some input parameters and/or some input file(s). The following program is very simple but still illustrates the more general case. Instead of processing a file, it generates the results by (a recursive) computation.

```
int a = 10;  // introduced to illustrate sharing

void main ( )
     {
      int x, y, w;
      scanf ("%d", &x);
      y = factorial(x);
      w = a * y;
      printf ("%d\n", w);
     }

int factorial (int x)
     {
      int z;
      if (x == 1) return (1);
      z = x * factorial(x-1);
      printf ("%d %d\n", a, z);
      return (z);
     }

}
```

The execution of the C program produces the following results:

```
10    2
10    6
10    24
240
```

For a simple output, the documentation may easily explain to the future user what the output represents. But, in general, the future user will want to process the data and it is therefore preferable to return the data elements, one by one, and tagged. If tag=1 identifies the output for a, tag=2 identifies the output for z, and tag=3 identifies the output for w, the results would be:

```
1     10
2     2
1     10
2     6
1     10
2     24
3     240
```

This is actually what is implemented in the UVC program below. The documentation may easily explain the format by using a simple specification (such as a DTD in XML):

```
Result:    Line*, W
Line: A, Z
A (1)
Z (2)
W (3)
```

where the values in parentheses indicate the tag values.

Constants.asm

```
# Constants to be defined in segment 0
# Entry format:
# register   sign (plus: 0, minus: 1)   length (in bits)   value (in hex)

# Constants used for communication
0 0 1 0x0          # default entry address into a section
1 0 1 0x1          # constant 1 = the message tag for a
2 0 2 0x2          # constant 2 = the message tag for z
3 0 2 0x3          # constant 3 = the message tag for w
4 0 16 0xFFFF      # constant is memory address of message to be output

# Global variables
5 0 4 0xA          # a = 10
```

Main.asm

```
Main
1001                # segment number for Main section
0,1002,1003         # segments this section references (for assembler only)

# Program: Main, to compute the factorial of a given number
# This program computes the factorial of a value received over the
# communication channel. It outputs the result as binary values.
# These values are tagged as mentioned in the simple specification above;
# the tags themselves are communicated as message types.
#
# By convention, the argument to the factorial section is loaded in register
# 1 of the segment containing the argument. The result is communicated back in
# register 2 of the same segment.

# section uses 1002 as working segment
# Set input address in register (1002,12): 255
LOADC 1002 1 8 0xFF

# Get argument and load it in argument section (1003)
IN 1002 2 1002 3 1002 1            # only (1002,1) is an input argument
LOAD 1003 1 1002 1 1002 3          # save input value x in (1003,1)

# Set up arguments to call Factorial (seg 1010) with arguments (seg 1003)
LOADC 1002 4 12 0x3F2              # set (1002, 4) to value 1010
LOADC 1002 5 12 0x3EB              # set (1002, 5) to value 1003

# Call the factorial section
CALL 1002 4 0 0 1002 5

# z = a * factorial(x)
LOADR 1002 6 0 5                    # copy value of a in (1002,6)
MULT 1002 6 1003 2                  # multiply a by the result of factorial
RLEN 1002 3 1002 6                  # store length of result in (1002,3)
STORE 1002 6 0 4 1002 3             # store result in output area - at address
                                    # specified in (0,4)
OUT 0 3 1002 3 0 4                  # message type = 3 (0, 3)

STOP
```

Factorial.asm

```
Factorial
1010                    # segment number for Main section
0,1,2,1002,1003         # segments this section references (for assembler only)


# Computes the factorial of the value given in register (2,1)
# The result is placed in register (2,2)

# Set (1002,2) equal to 1
LOADC 1002 2 1 0x1

# Move the arguments in segment 2 to local segment 1002
LOADR 1002 1 2 1

# if (1002,1) > 1, jump to recursion
GRT 1002 1 1002 2               # compare argument to 1
LOADC 1 1 RECURSION             # constant (address) computed by Assembler
BRC 1 1                         # greater than 1, needs more

# else jump to base
LOADC 1 1 BASECASE
BR 1 1                          # equal to 1, we have got everything

label: RECURSION
# call argument segment (1003) with segment new argument
LOADR 1003 1 1002 1

# subtract one (1002,2) from argument in (1003,1)
SUBT 1003 1 1002 2


# call section factorial (1010 = 0x03F2) with segment (1003=0x3EB)
LOADC 1002 3 16 0x3F2               # set (1002, 3) to value 1010
LOADC 1002 4 16 0x3EB               # set (1002, 4) to value 1003

CALL 1002 3 0 0 1002 4

# compute factorial
# multiply factorial(x-1) by x.
MULT 1002 1 1003 2                   # multiply result by (1002,1)
LOADR 2 2 1002 1                     # and store in (2,2)

# Return value of global var a
RLEN 1002 5 0 5
STORE 0 5 0 4 1002 5
OUT 0 1 1002 5 0 4                   # message type = 1 (0,1)

# Return result of this factorial call
RLEN 1002 5 2 2
STORE 2 2 0 4 1002 5
OUT 0 2 1002 5 0 4                   # message type = 2 (0,2)
BREAK

label: BASECASE
# base case of recursive routine n=1
LOADR 2 2 1002 2
BREAK
```

Appendix B: UVC opcodes

Opcode	Dec.	Hex	Operands	Function
			Move information between registers and memory	
Load	10	0A	Reg1 (dest), Reg2 (address), Reg3 (length)	Load from memory to register
Store	11	0B	Reg1 (src), Reg2 (address), Reg3 (length)	Store from register into memory
Lsign	12	0C	Reg1 (dest), Reg2 (address)	Load sign
Ssign	13	0D	Reg1 (src), Reg2 (address)	Save sign
			Operations on registers	
Loadr	20	14	Reg1 (dest), Reg2 (src)	Load register
Psign	21	15	Reg1 (positive)	Set sign to positive
Nsign	22	16	Reg1 (negative)	Set sign to negative
Loadc	23	17	Reg1 (dest), Reg2 (length), bit string	Load constant
Rlen	24	18	Reg1 (length), Reg2 (src)	Get register length
			Numeric instructions	
Add	30	1E	Reg1 (dest), Reg2 (arg)	Add
Subt	31	1F	Reg1 (dest), Reg2 (arg)	Subtract
Mult	32	20	Reg1 (dest), Reg2 (arg)	Multiply
Div	33	21	Reg1 (quotient), Reg2 (arg), Reg3 (remainder)	Divide
			Comparison instructions	
Grt	40	28	Reg1 (arg1), Reg2 (arg2)	Greater than (arg1 > arg2)
Equ	41	29	Reg1 (arg1), Reg2 (arg2)	Equal (arg1 = arg2)
			Logical instructions	
Not	50	32	Reg1 (dest),	Negation
Or	51	33	Reg1 (dest), Reg2 (arg1)	Or
And	52	34	Reg1 (dest), Reg2 (arg1)	And
			Instructions that alter the flow of execution	
Br	60	3C	Reg1 (address)	Branch
Brc	61	3D	Reg1 (address)	Branch on condition
Break	62	3E		Return to calling section
Call	63	3F	Reg1 (section), Reg2 (address), Reg3 (argument)	Call another segment
Stop	64	40		Stop execution
			Communication with the outside world (I/O)	
In	70	46	Reg1 (msg type), Reg2 (length), Reg3 (address)	Input
Out	71	47	Reg1 (msg type), Reg2 (length), Reg3 (address)	Output