Connecting with the Outside World

by Russell White, IV

An Interim Document, Nothing Is Final

The need for external-data and routine interchange has long been recognized in segments of the M community.[1,2] In 1989, Peter Kuzmak suggested possible interfaces between M and the external world. The new MDC (MUMPS Development Committee) standard enables M programmers to call to non-M routines using a standard-calling syntax.[3] This syntax replaces the nonstandard $ZCALL mechanisms that many vendors use. The syntax uses an introducing character [&], a package name, a period, and the actual routine name, followed by any parameters, for example:

```
S DISPLAY=$&XLIB.XOpenDisplay("M Window")
D &XLIB.XStoreName("Mwindow")
```

External-data typing is not addressed in the standard. In M, there are a large number of implicit data types.[4] In non-M environments, data types are usually explicit. Even when the data typing is explicit, there are complex issues surrounding an interface between any two languages or environments. A number of existing calling standards work with multiple languages (e.g., VAX Calling Standard, 1978 to date, and Digital Equipment Corporation), but these are environment- and vendor-specific. Several American National Standards Institute (ANSI) and International Organization for Standardization (ISO) groups are working on many related topics. One group produced an interim document for Common Language-Independent Data Typing (ISO/IEC/JTC1/SC22/WG11/N319R). The specification is large, complex, and cumbersome. Many people have worked years in this field, and a final standard has yet to emerge. Issues encountered while developing the X Window binding provide a snapshot of these problems.

Three Interfaces

Three types of interface are possible between two languages or environments. The first is a byte- or bit-stream mechanism. This works well for platform-independent network protocols, and is used for both Open M Interconnect (OMI) and X. This type of interface is typically at a lower level than is optimal for application development. Applications on both sides of the byte stream must encode and decode. Performance is not necessarily a feature of these implementations. Portability frequently is a feature, although some byte-stream protocols have failed this as well. To enhance performance, implementations of some protocol-level interfaces include standardized libraries. In either case, both sides of the interface must use compatible interfaces. Examples of standardizing remote procedure calls (RPCs) suggest that this is not uniformly accomplished easily.

The second type of data interchange is that of explicit data typing and conversion. Some of this methodology was used with the X binding. This was only feasible because the X environment is defined in explicit and extensive detail. Detailed mapping of data structures between environments has some negative aspects. The syntax to specify the access can become cumbersome, and may be nonintuitive to some users. While declaring the data translation is feasible in many cases, it also implicitly may impose additional data-typing requirements. This can be a useful feature between two environments wherein there is explicit typing. When the two environments are more disparate, this can impose burdensome restrictions.

Data-type mapping requires three pieces of knowledge. First is the data types supported by the calling language. Second is the data types supported by the called language. Third is data-fit algorithms, best-fit algorithms, and rules for exceptions to the direct mappings. An alternative to this methodology is an intermediate definition language (IDL) and languages that support the first two pieces to interface to the IDL. Now there are no standard IDL/language interfaces, and at least three standard IDLs exist. Some form of an IDL also may be useful in the third method of data interchange.

The third method of data interchange is that of routine-level interface. This is probably the highest-level interface practical between environments. This level allows the calling of routines between environments using defined interfaces. The X binding also made use of this methodology. It is also available in various forms on today's M implementations. RPC forms are based upon this concept, although to date no platform- and environment-independent RPC exists. Many possible uses of technology may be developed at this level. One is the interchangeable use of routines or environments.

What Constitutes a Routine?

Lack of standardization in the definition of what constitutes a routine call also presents a problem. ANSI X11.1 defines a routine. ANSI X3.159-1989, the C language standard, has another definition. Similarly, FORTRAN, COBOL, and other languages each use different calling and parameter mechanisms. In many cases, these are defined in a language-specific manner, and no provision is made for any standardized mechanism outside that environment. Thus, a mapping for calling and interchange of routines is needed.

Looking solely at RPC does not work. Many current RPC implementations are in fact C-based, and presume C calling conventions that are not available on all platforms. Therefore, methods of routine interchange and definition are still mostly platform dependent. Standards work is underway to define standard interfaces, within both the Institute of Electrical and Electronic Engineers (IEEE) (P1003) and ANSI (X3T2).

The X Window binding for M used techniques of two forms of data interchange. Each has its uses, although for general applications-level programming, routine interface seems to be preferred to direct data interchange. Advantages include higher performance and greater flexibility at the applications level. The callback mechanism employed within the binding is a good example of a simple routine interface. The `XMUMPS.SetValue` call is a good example of direct data interchange. The latter does not work well if the external environment is generalized. The techniques can certainly be used, but a well-specified external definition is required.

Reasons for each method vary. The X Window binding has a standard interface to routines typically written in C. Interchange at the routine level allowed simple access to the routines available on platforms supporting X Windows and Motif. These routines are widely documented and texts on them abound. This worked well for routine-level interchange, but data types within X are those of C, and there are many implicit assumptions regarding sizing and structure. Special-purpose routines with the binding to bridge data issues between the two environments now exist. These few routines manipulate the direct data interchange that sometimes becomes necessary. Looking at how various languages interface to X and to other environments provides some interesting insights to the entire problem.

The Bridges to X

C and most other rigidly typed languages use fixed (machine-dependent) sizes for atomic data types. The bit sizes of the atoms may vary across platforms, but the relative sizes and names are constant. Users are warned about making assumptions about size, however. C also includes the nondeterministic zero-terminated character string and composite data types. Composite data types include arrays, structures, and unions. C arrays include a single atomic element, have specified dimensions, and are both rigid and densely packed in their structures. M, by comparison, uses sparse arrays of any data type. In C, structures also can define data aggregates comprised of subelements that are either atomic data elements or composite data types, which eventually can be decomposed to atomic data types. Mapping between these environments raises a number of challenges for implementors. Differences in the scope of life of data within various environments are yet another challenge.

With the X binding as an example again, using the highest-level interface (usually the level of the routine) greatly simplifies programming at the applications level, if data typing is assumed to be implicit or externally defined. This method works for interchange with environments where no explicit structure must be applied to data formats. For those formats where data interchange is needed, transformation at routine invocation time, if feasible, is useful. The issue of where data reside, given two environments, is problematic. Keeping the data on one side only can lead to serious performance problems or programming difficulties. Flexible high-level interfaces solving these issues are highly desirable.

The physical structure of atomic types presents problems for languages using rigid typing. Other issues are less apparent, but more problematic. Many languages use either ad hoc calling structures or language-specific structures. Lack of standardization wreaks havoc in a multilingual programming environment. VAX users have long taken for granted the calling standard that VMS-layered products use. Within the personal computing market, it is different. There have been cases where one vendor's C compiler uses a different calling structure than that of another vendor. If one uses a library compiled by one vendor's compiler, the lack of a standard calling mechanism can prevent using another vendor's compiler with that library.

Avoiding Other Languages' Pitfalls

These issues are greater than that of a single language or environment. While data approximations that work across platforms are feasible, there may be environments where they do not work. Standardizing the physical format of data does not ensure that complete interoperability will occur automatically. M has avoided these pitfalls in the past by taking a higher-level view. The concept of an IDL is useful, but care must be taken to avoid a yet larger set of problems.

Providing data representation as a part of external routine interface appears to give the best hope for interoperability with the rest of the computing world. Implementations of the IDLs exist on some M external-call implementations today. They deal with interoperability on the platform or platforms for which the implementation was designed. Typically, these are implementations and methods that are generally applicable. This method places the data-interface definitions with the routine-access definitions. The implementation of these IDLs across multiple platforms has proven this to be one valid methodology.

Defining absolute mechanisms or atomic hardware types may lead to the same pitfalls encountered with other languages. As anyone who has ported compiled code between 16-bit, 32-bit, and 64-bit processors or interfaces can attest, programmatic or data size assumptions are often overlooked, and cause enormous platform portability issues. Combining relative sizing and characteristic specific typing (numeric, alpha character, Boolean, etc.) is probably the best solution, being simple yet flexible. This approach can be taken consistently with the proposed 1993 ANSI X11.1 specification.

Providing data representation as a part of external routine interface appears to give the best hope for interoperability with the rest of the computing world.

Creating standard interfaces at levels where solutions are either infeasible or unusable on various platforms seems fruitless. Specifying exact physical data element sizes, explicitly or implicitly, is not useful either. No matter what size is chosen, some platform exists for which these are invalid, useless, or possibly unimplementable. Using the highest-feasible level of abstraction lends to programmers the strengths of M coupled with access to the external world. Any IDL notation adopted must be readily understandable and map directly to M data constructs. A notation so abstract and complex that the meaning and mapping are not readily recognizable by both M programmers and external programmers serves little purpose. To date, M has been remarkably free, by design, of these issues. Whatever solution emerges, it needs to prolong this trend. **M**

Endnotes

1. P. Kuzmak, "Interfacing MUMPS to Mainstream Computing," *MUG Quarterly*, 16:2/3 (May 1986), 5.

2. P. Kuzmak, "The Challenge of MUMPS for the 1990's: Interface MUMPS to Mainstream Computing," *MUG Quarterly*, 19:1 (May 1989), 65-69.

3. R. White, IV, "External Routine Calling Syntax," *MUMPS Computing*, 22:2 (April 1992), 37.

4. T. C. Salander, "Data Types: Strong, Weak and Imaginary," *MUG Quarterly*, 20:2 (August 1990), 43-51.

Additional Sources

International Organization for Standardization, [ISO/IEC/JTC1/SC22/WG11/N319R] {also ANSI X3T2/92-097} Information Technology - (Common) Language Independent Datatypes, Working Draft 6.1, 9 September 1992.

D. Marcus, "Bit Manipulation," X11/SC13/TG2/92-2, 1992.

D. Marcus, "Data Structures," X11/SC13/TG9/92-4, 1992.

R. White, IV, "Arbitrary Structures," ANSI X11/SC9/90-13,14,15,16.

R. White, IV, "X Window Binding," *MUMPS Computing*, 22:1 (April 1992), 43-47.

Russell White works in the DSM Product Group of Digital Equipment Corporation in Massachusetts. He is a member of the M Technology Association, the New England MUMPS Users' Group, DECUS, and other professional organizations. In addition, he is the chairman of Subcommittee 13 of the MUMPS Development Committee, the subcommittee overseeing data management and manipulation. Beyond M interests, he writes for publications on horticulture and philatelics.