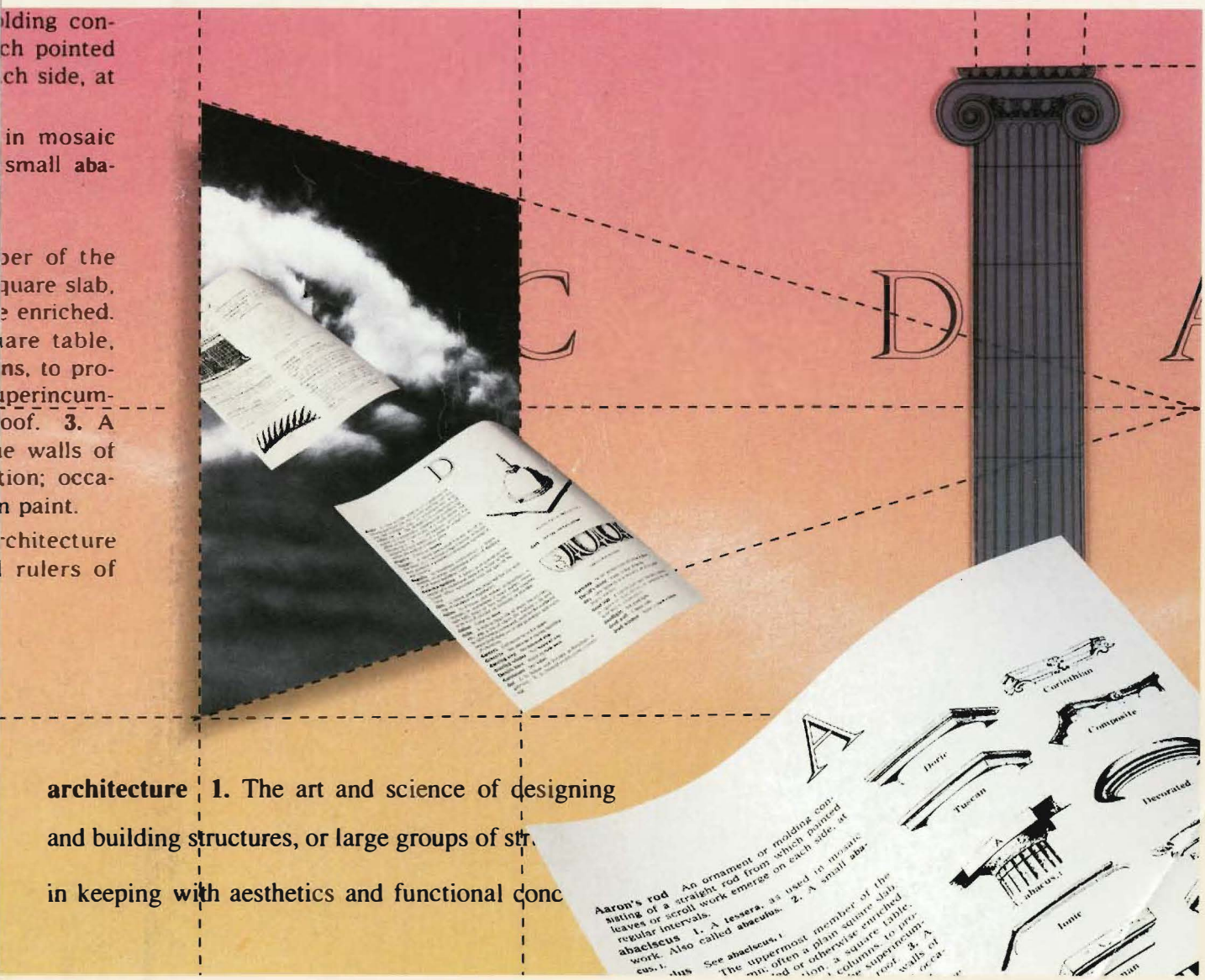
Compound Document Architecture

Digital Technical Journal

Digital Equipment Corporation



Volume 2 Number 1

Winter 1990

The *Digital Technical Journal* is published quarterly by Digital Equipment Corporation, 146 Main Street MLO 1-3/B68, Maynard, Massachusetts 01754-2571. Subscriptions to the journal are $40.00 for four issues and must be prepaid in U.S. funds. University and college professors and Ph.D. students in the electrical engineering and computer science fields receive complimentary subscriptions upon request. Orders, inquiries, and address changes should be sent to The *Digital Technical Journal* at the published-by address. Inquiries can also be sent electronically on NEARNET to DTJ@CRL.DEC.COM. Single copies and back issues are available for $16.00 each from Digital Press of Digital Equipment Corporation, 12 Crosby Drive, Bedford, MA 01730-1493.

Digital employees may send subscription orders on the ENET to RDVAX::JOURNAL or by interoffice mail to mailstop MLO 1-3/B68. Orders should include badge number, cost center, site location code, and group name. U.S. engineers in Engineering and Manufacturing receive complimentary subscriptions; engineers in these organizations in countries outside the U.S. should contact the journal office to receive their complimentary subscriptions. All employees must advise of changes of address.

Comments on the content of any paper are welcomed and may be sent to the editor at the published-by or network address.

ISSN 0898-901X

Documentation Number EY-C196E-DP

Book production was done by Digital's Educational Services Media Communications Group in Bedford, MA.

Cover Design

This issue features products specified in Digital's compound document architecture, the CDA architecture. The design on our cover takes a page from classical architecture to evoke the concepts of structured architecture and creation of compound documents. Like the blueprint shown in the background, the CDA architecture provides a structure for applications to seamlessly integrate text and graphics into pages such as those displayed in the foreground.

The cover was designed by David Comberg and David Shepherd of the Corporate Design Group.

Contents

Editor's Introduction

Jane C. Blake
Editor

Compound document architecture is the theme of this issue of the *Digital Technical Journal.* Digital's architecture for compound documents, CDA, addresses a need all of us have if we work with text files, tabular data, graphics, or images. We want the flexibility to easily exchange that data, revise it, or combine it into one document, regardless of the applications, operating systems, and hardware involved.

In the paper that opens the issue, Bob Travis recalls Digital Engineering's recognition of this need for an overall data interchange environment. Bob's paper is an overview of the CDA architecture which resulted from this recognition. He introduces the syntax and standards that formed the basis for development and shows the relationships of these to the CDA components and to the goals for the architecture.

Central to the architecture is the revisable form of a compound document and its data cross-linkages. In the paper following the overview, Bill Laurune and Bob Travis discuss the DDIF document interchange format which specifies the revisable form for text, graphics, and images. The authors describe the DDIF design, its relationship to standards, and practical aspects of data interchange.

A second format, DTIF, is the subject of Carol Young's and Neal Jacobson's paper. They open their discussion with a description of the types of problems designers had to address to allow for the interchange of revisable tabular data between applications. They then describe how the DTIF format meets the goals they set, among which were architecture neutrality, application independence, and extensibility.

For processing DDIF, DTIF, and non-CDA formats, the CDA architecture includes a set of services, specifically, the CDA Toolkit and the converter architecture. Dick Gumbel and Marty Jack relate how the toolkit was designed to provide application programs access to CDA documents through a procedural interface. They also describe the converter architecture which imports and exports documents to and from non-CDA formats.

Two more services are included in the CDA architecture for efficient interapplication integration. In their paper, Baldwin Cheung and Neal Jacobson give details of the AIL application interface library and DECdecision Builder. With AIL, developers can build tightly integrated levels of applications. DECdecision Builder allows applications to be integrated at the user level.

Al Sung, Neal Jacobson, and Carol Young then describe the DECdecision product. Designed for end-user decision support, the DECdecision application's five components provide a high level of data integration through database access, spreadsheet, charting, flow control, and management functions. DECdecision serves as a model for the development of applications that use the features of the DECwindows and CDA architectures.

The DECwrite editor, developed in conjunction with the DECwindows and CDA architectures, is the topic of our next paper. Seth Cohen and Gene Morgan focus on the relationship between this DECwindows-based compound document editor and the DDIF interchange format. By resolving issues relative to this relationship, designers were able to draw on the benefits of the interchange format without sacrificing formatting speed and ease of editing.

The closing paper for this issue, by Neal Appel and Ron Olson, features an application that extends the CDA architecture to meet scientific and engineering requirements. Neal and Ron describe the development of the DECview3D application which supports interactive two- and three-dimensional viewing and annotation of scientific and engineering data.

I thank Bob Travis for not only contributing papers to this issue but for his help in developing its general content. I would also like to take this opportunity to welcome Barbara Lindmark and Cathy Phillips to the DTJ office. Barbara is the associate editor who has helped launch the journal on its new quarterly schedule. Cathy is the subscriptions administrator who has made journal subscriptions available to our readership, beginning with this issue.

Jane Blake

Biographies

Neal B. Appel As a principal software engineer in the CAD/CAM Technology Center, Neal Appel is the architect for the DECview3D product and consultant for the design of other CAD/CAM products. Before joining Digital in 1987, Neal worked at Computervision, where he was a member of the Technical Staff Council. In this role, he provided technical direction and consulting for Computervision's graphics development. He received a B.Sc. (Honors, 1977) in mathematics from Northeastern University, an M.Sc. (1979) in applied mathematics, and an M.Sc. (1980) in computer science from Michigan State University. He is a member of ACM and the SIGGRAPH Special Interest Group.

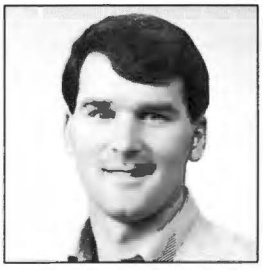
Baldwin K. Cheung Baldwin Cheung is a principal software engineer in the Core Applications Group. He is a developer of the DECwrite editor, the architect for the AIL application interface library, and responsible for developing the LiveLink functions. With co-developer Neal Jacobson, he has applied for a patent for the Application Interface Library. Since joining Digital in 1982, Baldwin has participated in the VT240, PRO/GIDIS, PRO/NAPLPS, PRO/VENIX, and VS31 graphics development projects. He holds a B.S. (1977, highest honors) in mechanical engineering from Northeastern University. Baldwin has taken graduate studies at the Massachusetts Institute of Technology.

Seth S. Cohen A consultant software engineer in the Workgroup Publishing Group, Seth Cohen designed the DECwrite compound document editor and led its development effort. He also coordinated the use of DECwrite technology across different Digital platforms and projects. Since joining Digital in 1973, Seth has designed the hard-copy support of UIS workstations, the PRO document architecture, and a package to validate RMS files. He has led the development projects for his designs, as well as the development teams for the PRO graphics arts, RMS-10/RMS-20, DBMS-10/DBMS-20 projects. Seth has an M.S. (1973) in computer science from the Massachusetts Institute of Technology.

Richard T. Gumbel Richard Gumbel joined Digital in 1979. He is currently a principal software engineer in the Core Applications Group. Dick is one of the developers of the CDA Toolkit and CDA converter architecture. Together with his co-developer, Martin Jack, he has applied for a patent for the CDA converter architecture. He is currently designing more features for the toolkit. Prior to his work on CDA projects, Dick was part of the DECforms and RT-11 development efforts, and the leader for the VAX FMS project. He has a B.S. (1974) in physics and an M.S. (1979) in computer science from West Virginia University. Dick is a member of ACM and the SIGSOFT Special Interest Group.

Martin L. Jack One of the developers of the CDA Toolkit and CDA converter architecture, Martin Jack is now designing new approaches to simplify software product installation and management. He has applied for a patent for the CDA converter architecture with his co-developer, Richard Gumbel, and for a patent for the DOTS multiple object transport system. Marty is a consultant software engineer in the VMS Base Systems Platform Engineering Group. He worked on the VAX COBOL compiler project and was a member of the VAX architecture review committee. He joined Digital in 1973. Marty has an S.B. (1971) in chemistry from the Massachusetts Institute of Technology.

Neal F. Jacobson As the project leader for the DECdecision Builder project, Neal Jacobson led the design and development of this component. In his position as consulting software engineer in the Core Applications Group, he is currently the architect for the Application Control Services project. Neal joined Digital in 1980. He received a B.S. (1978) in computer science and mathematics from SUNY at Albany. Neal has also worked on VAX DBMS and led the VAX TEAMDATA project. He has patent applications pending for the Builder Application Integration Services, the DTIF table interchange format, and the Application Interface Library.



William R. Laurune At the time of his work on the DDIF document interchange format, William Laurune was a principal software engineer in the Core Applications Group. He is currently a member of the Digital team that is developing factory automation software for Boeing Commercial Airline's new sheet metal center. Bill joined Digital in 1980. He has a degree in English writing, with a minor in computer science, from the University of Pittsburgh. A combination of graduate classes in computer graphics and a writing background led to a natural interest in Digital's first laser printer support projects and advanced document processing technologies.

Wm. Eugene Morgan Eugene Morgan is the co-developer of the DECwrite version 1.0 product and developed the DDIF document interchange format's read and write code for that version. As a principal engineer in the Workgroup Publishing Group, he is now leading the DECwrite version 1.1 and version 2.0 projects. Since joining Digital in 1978, Gene has worked on many projects, including support of DECnet-20 and IBM 2780/3780 communications for DECSYSTEM-10 and DECsystem-20, and the Printserver 40 Communications Software, for which he has a patent application pending. Gene studied mathematics and computer science at Old Dominion University, and is a member of ACM and SIGGRAPH.

Ronald M. Olson As a marketing consultant for the LDP/Science Product Marketing Group, Ronald Olson is the project manager for the Scientific Document Processing program. One of his major responsibilities is the development of CDA architecture support among science market application vendors and customers. In 1989, Ron received Digital's Marketing Leadership award. Prior to joining Digital in 1974, Ron was employed by Control Data Corporation. He has a B.A. in Mathematics from St. John's University. In addition to the *Digital Technical Journal,* Ron has also published in the *CODASYL Systems Committee Technical Report*.

Alan Sung Alan Sung is a principal software engineer in the Core Applications Group. In this position he is responsible for the development of decision support products. He was the senior technical contributor to the DECdecision Calc project and continues to act as a consultant to the project. He was also the principal designer and project leader for the VAX Xway project. Prior to that project, Alan worked on the VAX DECalc project in the Technical Systems Group. He joined Digital in 1982. He has a B.S. (1982) in computer science from Cornell University and is a member of ACM.

Robert L. Travis Jr. Since joining Digital in 1978, Robert Travis has been a member of the development teams for many of Digital's major document processing products, including WPS-8. He represented Digital on the ANSI and ISO standards committees for the development of mail/messaging, SGML, and ODA standards. Most recently, as a senior consultant for Software Engineering in the Core Applications Group, Bob developed the CDA architecture and started the CDA program. Before joining Digital, he managed his own software consulting firm. He holds a B.A. (1963) and an M.A. (1965) in mathematics from Wesleyan University.

Carol A. Young Carol Young is the architect of the DTIF table interchange format and leader of the DECdecision Calc version 1.0 project. She and her co-developers have applied for patents for the DTIF format and for asynchronous DECalc-PLUS. Carol is currently a principal software engineer in the Decision Support Group. Prior to this, as a senior software engineer, she was the project leader for the development of DECalc versions 3.0a, 2.2, and 2.1. She was a technical contributor to DECalc-PLUS version 1.0. Carol joined Digital in 1982 after receiving a B.A. in computer science and French language and literature from the University of Michigan.

Foreword

Jeffrey H. Rudy
Group Manager
Core Applications Group

In the 1970s, word processing systems were the most advanced method of document processing. With this technology, users could create and revise documents. However, there were limitations. Only one user could access data at a time. Graphics were not supported, and the finished product had the appearance of "computer-generated" material.

In the 1980s, higher performance electronic publishing systems began replacing word processing systems. Users could now create and revise text *and* graphics, and could produce professional, high-quality printed documents. Electronic publishing systems also met a growing demand for more information-sharing capability. Multiple users could now share the data. However, there were still limitations. Information could only be shared among users of the same operating systems and the same applications. Electronic publishing systems did not address the need to integrate and interchange data.

As we approach the 1990s, the need is growing to interchange data across different operating system platforms, different applications, and different standards. A principal barrier to this type of information sharing is the fact that companies will continue to use different systems. Therefore, vendors must develop document processing systems that both meet these companies needs and accommodate existing and future multivendor environments.

In the early 1980s, Digital recognized that future document processing systems would need to support data interchange and integration through a compound document architecture. To this end, the diverse document processing development projects then underway were brought together into one body to reach technical agreement on a common architecture and standards for data interchange. From this effort, an overall design structure for data interchange among all developing applications and a meta-syntax for that design were defined.

In 1987, Digital announced its CDA design. The first version of the CDA toolkit that incorporates this design was distributed in 1988. The design comprises three parts: the CDA architecture, CDA-compliant products, and the CDA program. The interchange formats in the CDA architecture are, in part, an outgrowth of the work done in the 1980s by U.S. and international standards bodies.

The CDA architecture is a comprehensive and open interchange standard which enables users to easily interchange numerous types of revisable information among applications and with each other. We refer to this revisable information as "compound documents" that can encompass not only text, but graphics, images, and numerical data, as well as the table-oriented data usually associated with spreadsheets or database queries. The CDA architecture has been further enhanced with an Applications Control service that dynamically links the files in a CDA document to give users access to "live" data in the network.

Moreover, the CDA architecture is operating system-independent and removes the barrier to information sharing that has limited previous document processing systems. In fact, Digital has made the technical specifications for the CDA architecture public to further support the open interchange of information.

At the present time, the CDA architecture contains four formats that support data interchange. A fifth is planned for the near future. The DDIS document interchange syntax, which is the meta-syntax, is the base encoding system for the interchange formats, DDIF and DTIF. DDIF is the CDA document interchange format for revisable text, graphics, and images. DTIF is the CDA table interchange format for revisable data tables and spreadsheets. The DOTS object transport syntax supports the electronic transfer of multiple data elements, in multiple interchange formats.

The CDA architecture goes beyond other data interchange formats by providing an integrated conversion architecture that supports software development for translating existing formats to and from CDA documents. In addition, the CDA conversion architecture acts as an interchange hub among multiple formats. Data can be moved from

one format to a CDA format and then translated to another format. As additional converters become available, the conversion architecture will simplify data interchange and support of multiple standards and application formats for the application developer because separate converters and applications will not be necessary.

CDA-compliant products give users the power to implement the CDA architecture and actively exchange and share data.

In the past year, Digital has announced several major CDA-compliant products. The DECwrite editor is a "what-you-see-is-what-you-get" (WYSIWYG) compound document editor that combines word processing, desktop publishing, drawing, and business graphics into one product. The DECdecision product is a workstation-based, advanced decision support solution for data-dependent professionals. Three VAXimage products permit users to include images, i.e., nonelectronic, existing objects such as photographs, in documents and exchange them as if they were text or graphics.

The scope of CDA-compliant products has been extended to meet the unique document processing needs of specific industries. In the scientific area, the DECview3D product provides graphics translations, two-dimensional and three-dimensional graphics manipulation, and annotation of engineering and scientific data.

Digital is as firmly committed to supporting third-party development of CDA-compliant products as it is to developing these products itself. Many CDA applications have been and will continue to be produced by organizations external to Digital. Third parties were actively involved in the design of the CDA architecture to ensure that the design included what they needed to develop CDA-compliant products in the future. The result of that involvement was very successful. We now have over 50 key software companies developing CDA applications.

The CDA program is a broad set of activities that support the implementation of the CDA architecture. These activities include:

- Providing tools, training, and support for CDA-compliant products by independent software vendors
- Making CDA toolkits available for multiple platforms and extensions to the CDA architecture
- Making the *CDA Manual* publicly available
- Ensuring that all future Digital products support the CDA architecture

Together, the CDA architecture, the CDA-compliant products, and the CDA program answer the data interchange limitation of past document processing systems.

For the future, the CDA design is open and extensible. Already, Digital and third parties are developing more applications to extend the revisability capabilities for the data and add more interchange methods.

Although many vendors have recognized the need for a compound document architecture, Digital is the first to develop and release one. Our success is based on many factors. One is our ability to build system architectures. Our network, VAX system, and DECwindows architectures are successful examples of that point. Digital is also a recognized supporter of open standards, as proven by its founding membership role in the Open Software Foundation. Many of the CDA applications, such as the DDIF and DTIF interchange formats, are based in part on existing ISO standards. Finally, Digital has probably the broadest and most in-depth expertise and experience in building distributed network systems. From all of this, we were able to build a single system architecture that provides an integrated, seamless computing environment across multivendor operating platforms.

In this issue of the *Digital Technical Journal,* you will learn in more detail about how the many pieces of the CDA design were developed. As you read these papers and come to see the CDA design as an integrated whole, you will understand how the CDA design makes a truly distributed network possible.

Robert L. Travis Jr.

CDA Overview

The CDA family of architectures, services, and applications is designed to support the creation, interchange, and processing of compound documents in a heterogeneous network environment. This family emerged as the result of a fundamental goal: to develop a coherent set of standards and capabilities for data interchange across the Digital computing environment. Of the four stages identified by the CDA document processing model, the central focus is the revisable compound document and its logical structures and data cross-linkages. Key design decisions for each of the major CDA components were made with reference to Digital, industry, and international standards. The major CDA component development efforts are described in more detail in this issue's succeeding papers.

Background of the CDA Program

As our society has made the transition from an industrial economy to one based on information, the role of documents in business has changed. Documents are no longer merely for record-keeping — slips of paper that record the transfer or status of real goods. Today, documents are often the real goods.

Documents are a fundamental part of the overall business information flow. Information flows from sensors, such as laboratory instruments, financial data acquisition, and processing programs, and is incorporated into summaries, reports, and presentations. The recipients of the information use it to make decisions that, in turn, will affect the information available to other decision makers.

In light of the growth of information sharing, document preparation and presentation technology is no longer the exclusive province of document processing professionals. Workstations, color graphics terminals and personal computers, laser printers, computer digital-font technology, and document preparation software make high-quality document production accessible to all.

As the amount of data available on-line increases, so does the need to create links among the various application programs that support the data. Multiple workstation and central processing graphics and data processing applications need to be able to interchange information in a revisable format to produce business and technical documents.

As early as 1983, Digital's Systems Architecture and Review Authority (SARA) recognized the need for an overall data interchange architecture for Digital products and sponsored a task force to design it. (SARA was a Digital Central Engineering cross-group technical committee that has since been superseded by other structures.)

At that time, a number of separate text and graphics projects existed. These projects were related to workstation developments and other platforms, and each had its own emerging architecture and standards. In addition, several database and spreadsheet-oriented data table processing efforts were underway, all of which clearly needed effective data interchange; without it, all the products would be "standalone" and unable to work cooperatively. The SARA effort brought together the representatives of these diverse efforts to reach technical agreement on common interchange.

The first thing the SARA task force did was to agree on an overall design structure for data interchange among all of the developing applications. This design called for a single, common meta-syntax for the data and a number of domains; each domain would have its own data syntax based on the common model. Some of the domains envisioned were compound documents, database tables, scientific arrays, laboratory data, and CAD/CAM product information. The efforts of the SARA task force culminated in agreement on a common meta-syntax, DDIS, on which each of the separate data domains would be based.

In addition to the definition of the DDIS meta-syntax, small work groups were formed with members from existing development efforts to address the domain-specific problems. The CDA interchange formats are, in part, an outgrowth of that work.

Certain additional adjustments were made to DDIF to enhance support for SGML. Bit-for-bit compatibility with ODA was not a goal for the DDIF format, since converters would be needed anyway to deal with the required extensions. However, all important compatibility functions were considered. Because of its positioning in the CDA model, the DDIF format only deals with the elements of ODA's Processible Document Architecture.

The text content of the DDIF format is derived directly from ISO character coding standards. The added graphics primitives were designed to be compatible with the ANSI/ISO Graphical Kernel System (GKS) standards because of their wide use. The document imaging model is designed to be compatible with PostScript. Image content types include the standard CCITT bitonal (FAX) encodings, as well as significant extensions to handle multispectral images.

Several non-Digital research groups helped to align the DDIF structures and semantics to key existing or evolving standards. (The Austrian Research Center at Seibersdorf and the AGD/FhG group at Darmstadt were among the groups that helped.)

For table data, there are a number of de facto industry standards or pseudo-standards. All are different, and all are proprietary to one or more products. The DTIF format defines a single standard encoding for all the functions that appear in all the important industry standards. DTIF may not assign a standard treatment for a function or feature which is unique to a single industry product and not apparently suitable for interchange with others. However, the DTIF format includes a general escape mechanism for private encodings in such cases.

No existing standards are applicable to the problem the DOTS syntax set out to address. The CCITT X.400 mail and messaging standard, however, is closest to the kind of multiple object transport function DOTS provides. Therefore, our decision was to design the DOTS system encoding to be as compatible as possible with the X.400 standard's treatment of multiple body parts in message interchange.

Unrelated to standards issues, but nonetheless a significant decision, was how to couple CDA products with the technical and scheduling constraints of Digital's base systems and with DECwindows software.

Since the CDA architecture is meant to be an enhancement to the overall system infrastructure, it affects a number of base system components. There is a logical coupling with DECwindows software, as well, because much of the benefit from the CDA architecture is only realized in a multi-application graphics context. As design projects, these components had different technical requirements and schedules. The impact of these factors was felt particularly in the development of the CDA Toolkit, because its consistency as a bundled product across multiple operating system environments had to be maintained.

Multiple product development schedules also complicated the debugging environment, because multiple layers of dependent products were being developed and tested at the same time. Fortunately, the Digital Easynet provided some very effective communications tools to support these multiple development groups. These included the VAX Notes product and multiple interconnected mail systems. We also benefited from an enthusiastic group of early users, who frequently accepted code updates on-line. As a result, the test-fix-retest cycle was accomplished very rapidly, even though many levels of products had to be tested.

Summary

The CDA family of architectures, services, and applications form an integrated support environment for compound documents. This support is designed and implemented to operate consistently on multiple hardware and software platforms. Tools for software development, as well as end-user access, are included within the CDA architecture. Significant efforts have been undertaken to ensure that the CDA benefits are not restricted to Digital-produced products.

There is still much work to do, however, as the applications and requirements for compound documents continue to expand into new media types — voice, music, and video — and into more dynamic relationships, such as hypertext. These areas, and others, are now active areas of CDA architecture research and development.

Acknowledgments

Many people in addition to those recognized elsewhere in this issue of the *Digital Technical Journal* have contributed significantly to the development of CDA, and it is only possible to name a few of them here. Tom Hastings and Vijay Trehan drove the SARA data interchange architecture and vision, and developed the first drafts of the DDIS specification

services were also designed in detail and developed. Fortunately, much of the early architecture model development and many of the key decisions had already been approved. Thus, the development teams were able to work with a reasonably stable base. Even so, a great deal of productive feedback from the development process into the architecture process still occurred.

Architecture and Development Trade-offs

Designers looked at a number of alternative technical paths during the CDA development — choices that would have led to quite different results. Perhaps the most significant of these was the selection of the revisable and final interchange formats.

The basic question in each case was whether we should develop a new Digital standard or adapt either an existing internal or an international industry standard.

As discussed in the earlier section Background of the CDA Program, DDIS was chosen as the basis for the CDA architecture interchange syntaxes, based on ISO ASN.1. Other alternatives had been considered. These included a variant of TLV binary encoding used in the VAX workstation (VWS) facility, as well as some plain text encodings. None of these alternatives had the combined advantages of the chosen DDIS encoding.

- It was a good match with ISO and Consultative Committee on Telephony and Telegraphy (CCITT) standards.
- It could use the same encoding support as in other network and data protocols.
- It was compact and easy to decode.
- It was extensible.
- It could easily handle arbitrary embedded data syntaxes, including binary.

The DDIS syntax provides some Digital standard private identifiers for convenient exchange of popular data types, such as floating point and standard character string types. It also defines domain identifiers that are used to uniquely identify data streams, which are defined as DDIS syntaxes. Some of the more complex features of ASN.1 are not included in DDIS — macros, for instance — because they are not yet needed in the specific data syntaxes. This may change in the future, as the DDIS specification use continues to broaden.

The DDIF format had to be highly compatible with the existing ISO Standard Generalized Markup Language (SGML), ISO 8879, and the emerging ISO Office Document Architecture and Interchange Format (ODA/ODIF) standard, ISO 8613. Digital's customers today require the implementation of computing standards that extend beyond the boundaries of any single vendor. Even standards that are merely national in scope are no longer adequate, since electronic communications today span the globe to support international commerce. Even within the bounds of a single multinational corporation, the equipment of many different vendors and countries of origin may be represented. For these reasons, Digital is committed to the international standards process.

One option was simply to choose either SGML or ODA/ODIF and define the DDIF data format as its extension or specialization. The apparent advantages of this option faded on careful analysis.

- SGML and ODA are not compatible. Therefore, choosing one makes compatibility with the other somewhat problematic.
- SGML is delimiter based. Moreover, no techniques were available for mixing SGML elements with ASN.1 elements.
- SGML's standard technique for incorporation of nontext data is by external reference only.
- The DDIF format required semantics that the SGML standards did not contain.
- ODA was very incomplete in several areas critical to CDA. The required extensions would make the result nonconforming, since CDA could not consider the extensions optional. One example of this is the need to handle linkages between multiple documents and applications.
- ODA does not integrate text and nontext (e.g., graphics and image) data types very well. It simply combines the existing, nonintegrated standards by layering structuring primitives on top.

We decided to follow the ODA document model very closely, including the treatment of generic and specific structure, but to adopt a different treatment for combining structure with content. By following the ODA model closely, we are able to track ODA evolution compatibility, even though our philosophy maintains that the relationship between structure and content should be more highly integrated.

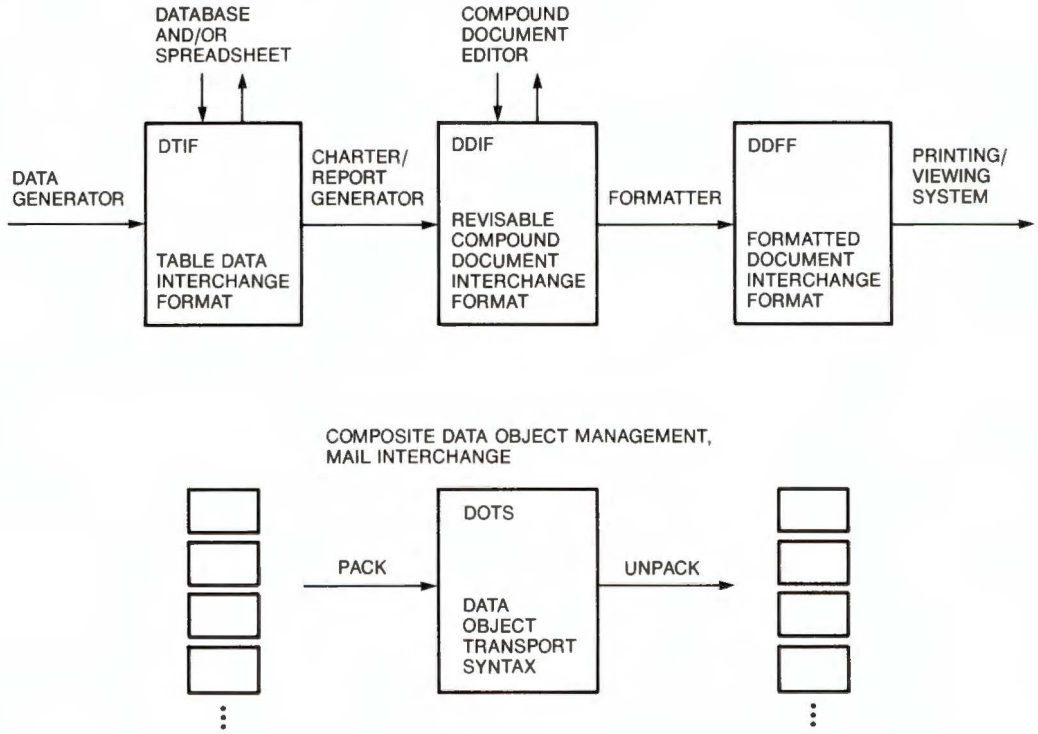


Figure 2 CDA Component Architectures

or implied to the creating application(s) can be changed, updated, or recalculated more easily than in final form documents. Applications that involve processing and data interchange are more easily developed and operate more efficiently by dealing with revisable rather than final form document data.

The revisable form for spreadsheet and data tables is specified by the DTIF table interchange format. The revisable form for structured text, graphics, and image is specified by the DDIF document interchange format. Both DTIF and DDIF provide for the inclusion of related or underlying data in other formats. The CDA Toolkit implements a data interchange hub using these two interchange formats. The toolkit also provides general processing access to in-memory representations of the data.

The final form of a document represents the abstract document component relationships as resolved display attributes. These attributes include text fonts, character positions, positioned and sized graphics frames, and final page layout. Final form is produced from revisable form by a formatting process. A final form document is specifically formatted for a particular class of display. The final form for compound documents in the CDA architecture is the DDFF format described earlier; the current version of this is based on PostScript.

Figure 2 indicates how the CDA architectures relate to each other, and which types of applications and services are most concerned with each architecture.

The Interchange Formats and Services

Within the model, two things could be clearly seen. These were the required interchange formats and their supporting services, and the key applications that must rely on those services and interchange formats. Task groups were formed to define the interchange formats and their supporting services.

Each task group first abstracted a design model, based on the application requirements. The priorities for elements within the model were based on the actual needs of the specific applications. Each format or interface had a responsible architect to arbitrate and make final decisions regarding the resulting structures.

As the CDA services such as the CDA Toolkit were being defined and built, the applications and bundled system components that needed to use the

The key Digital-produced applications that form the hub of end-user access to the CDA architecture facilities are the DECwrite application, a powerful compound document editor, and the DECdecision application, an integrated table data-handling and analysis package. Both are capable of tight integration with other applications, and with each other, by means of LiveLink connections.

DECwrite and DECdecision can be used cooperatively to build decision support applications and include the results in reports and other kinds of compound documents. The DECchart application is included with both DECwrite and DECdecision, and its features for dynamic plotting of data tables are available through LiveLink connections. The other components of DECdecision — Builder, Calc, and Access — are all linked together and dynamically share data by using the LiveLink facilities.

The CDA architecture and program comprehensively support and encourage non-Digital development groups to support the CDA architecture in their products. The focus of the CDA Toolkit is to provide easy access from application code to CDA interchange formats and converters. Training is available for software developers in the use of the toolkit and other CDA components.

Development of a Compound Document Architecture

The first concrete step in the development of the CDA architecture was to establish a model. This model would accurately describe the desired result and identify the various architectures and other necessary components.

The CDA Architecture Model

The CDA architecture model identifies a four-stage applications pipeline which culminates in the display or printing of a document. (See Figure 1.) The first stage encompasses applications that are the source of information (tables or graphs) for inclusion in documents. The second stage is a revisable compound document, where content is added and manipulated. This stage is dominated by logical structure and data cross-linkages. In the third stage, a final compound document results from applying formatting rules and layout characteristics to a revisable compound document. The fourth stage is the transformation of a final document into device-specific language for display or printing.

The processing steps and data formats of the first stage are diverse and very domain specific, since they are determined by the needs of many different processing environments. The first stage is important to the CDA model because it is the foundation for the LiveLink function's access to other applications and data. Application-specific data viewing modules (e.g., DECview3D software) provide the link between the stage one and stage two functions.

Data interchange can take place in all four stages, but the CDA architecture primarily focuses on the second one. Revisable compound documents, their processing and interchange, are at the heart of the CDA architecture.

The revisable form of a document contains abstract relationships between components of the document. Because these relationships are abstract (logical) rather than concrete (representational), any aspect of the document that the user has stated

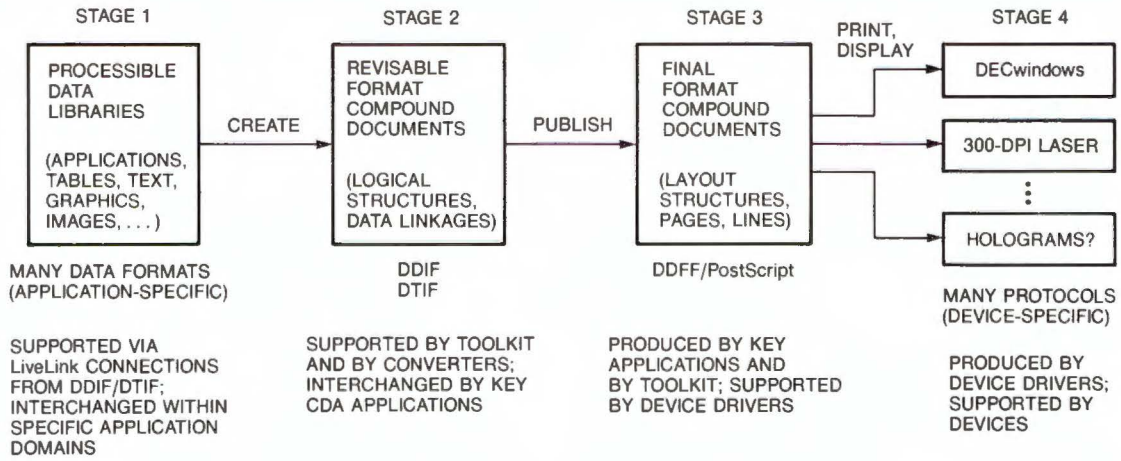


Figure 1 Compound Document Processing Model

There are currently three interchange formats and two related encoding standards that are managed as part of CDA.

- As noted above, the DDIS data interchange syntax is the base encoding standard for interchange formats. These formats include the DOTS data object transport syntax, the DDIF document interchange format, and the DTIF table interchange format.
- DOTS is a general encoding for multiple data elements, in multiple interchange formats, and the network of cross-relationships that might exist among them. It is used by the mail program to interchange multifile documents, maintaining the links among them.
- DDIF is the CDA interchange format for revisable compound documents. It is supported by the CDA Toolkit, read and created by multiple products, and edited by the DECwrite editor.
- DTIF is the CDA interchange format for revisable data tables and spreadsheets. It is supported by the CDA Toolkit and edited by DECdecision components.
- DDFF is the name given to the CDA interchange format for final form compound documents, but DDFF itself is not yet defined. We have hopes that the ISO SPDL (Standard Page Description Language) effort will be able to meet the CDA requirements in this area, when its specification is available. In the meantime, CDA applications use Structured PostScript as the canonical final form for compound documents. It is created by document formatting applications and by converters within the CDA Toolkit and consumed by output handlers. It can become a part of other revisable or final compound documents by means of links.

The DDIF and DTIF formats share many important characteristics, as can be seen from the associated descriptions elsewhere in this issue. Perhaps chief among these are a common approach to external file linkage to support application integration and to aid cross-system transport, and common features to support future extensions of the standards as well as ad hoc extensions to meet user- and application-specific needs.

Services and Application Program Interfaces

Services and their associated application program interfaces (APIs) are used by application developers to include CDA functions within applications. Standard APIs for the CDA bundled services are provided on all system types, which enhances the portability of applications by removing any potential system dependency.

The revisable interchange formats (DDIF and DTIF) both have associated in-memory formats that can be manipulated by using the CDA Toolkit. The toolkit also serves as the hub for the CDA converter architecture, which provides some converters that are bundled with the base operating systems and others that are optionally available with applications or in the CDA converter library. The CDA Toolkit also has a core DDIF layout capability, which forms the basis for the CDA viewers that are bundled with the operating systems.

The LiveLink services support cross-application integration within the CDA model. Currently, these services are bundled with the DECwrite editor and DECdecision tool applications.

On the VMS operating system, the record management services (RMS) have been enhanced to support automatic filtering of the CDA architecture data types into plain text streams. This enhancement allows existing non-CDA applications to have appropriate read access to CDA data types. CDA data streams that exist as RMS files are all specially marked as to data type by the new RMS semantics tag feature. This tag drives the selection of an appropriate filter when needed.

The corresponding facilities on the ULTRIX operating system are available as standard filters and utilities. Since semantics tagging is not possible on the ULTRIX operating system, the ULTRIX file utility has been enhanced to recognize the CDA architecture file types.

Mail and print utilities on the operating systems automatically deal with the CDA architecture file formats. The DECwindows mail interface can invoke the appropriate CDA architecture viewer on receipt of a mail message containing CDA architecture data types.

Applications

Digital has produced a few key applications as part of the CDA program. These applications provide a showcase for the CDA facilities and a focus for application integration, and also deliver some key end-user services. The bulk of the applications supporting CDA, however, will certainly come from Digital's customers and from other software producers.

DDIS Overview

The DDIS meta-syntax is based on the International Standards Organization (ISO) standards for the Abstract Syntax Notation; ISO standards 8824 and 8825 are collectively called ASN.1. ISO 8824 describes a notation for describing a data syntax, and ISO 8825 describes the method of physically encoding a given element of the notation. ISO 8825 prescribes a method of defining private data elements. We used this method in DDIS to define a floating point data type, which was missing from ASN.1. Also missing from ASN.1 was a means of representing the ISO 8859 character sets, such as Latin 1, now in standard use in Digital products.

The DDIS syntax is Digital's standard data definition and encoding method for the central CDA data formats. From the outset, we wanted a common data access method for CDA data formats so we could build a common data access layer, use a common notation, and build a common set of development tools such as analyzer utilities.

The DDIS standard notation can be compiled to create a parse table, symbol table, and symbolic definitions in the form of include files. These tools make software development faster and simpler.

The DDIS encoding has a number of desirable characteristics for encoding CDA data.

- DDIS is a tag-length-value (TLV) encoding. This method allows unused data elements to be omitted, thus producing a more compact encoding. Such elements are described in the notation as being "optional." Alternatively, the notation may specify a "default" value, which the DDIS data access layer used by the receiver supplies to the receiving process.

- The DDIS access layer converts the DDIS encoding to and from the native format of the CPU on which the process is running. The layer accomplishes this conversion by locating a given tag in a parse table that contains the data type of each tag. Thus, for example, DDIS-encoded mail delivered on a node is converted by low-level access routines to the system's native format at the time the mail is read.

- The DDIS encoding has no special delimiting bit sequences; it makes use of a full eight bits per byte. Thus the encoding can represent scanned images, arrays of floating point numbers, and integers more efficiently than an ASCII encoding, which would reserve certain bit sequences.

CDA Program Goals

Consequent to the efforts of the SARA task force to define the overall data interchange environment, Digital created the CDA program. The program would pull together the development and architecture resources needed for the full support of a robust compound document environment.

It soon became clear this effort would require a system-wide infrastructure upgrade. We realized that an architecture was needed to cover the data and function integration requirements of compound documents. As a result of that need, all relevant parts of the software system environment potentially would need enhancement to handle compound documents. In particular, those standard system utilities on which users depend for normal everyday tasks had to be enhanced. Such utilities as mail and printing would need to deal with complex data types as gracefully as they did simple ASCII text.

The technical goals adopted for the CDA architecture were to

- Provide seamless handling of text, tables, graphics, and image document content
- Be extensible to new media/data types
- Incorporate support for linked applications
- Deal with data and documents at the end-user (application) level
- Provide layered services for document handling
- Incorporate support for key industry and international standards
- Support heterogeneous systems

Elements of Digital's Compound Document Architecture

The CDA architecture is a composite architecture, comprising multiple interchange formats, services, and applications. All of the key components of the CDA architecture are described in detail in other papers in this issue of the *Digital Technical Journal*. These components are also described briefly here, with particular emphasis on their relationships to each other and to the CDA architecture as a whole.

Interchange Formats and Related Standards

Interchange formats and encoding standards enable CDA applications and services to interoperate in a distributed heterogeneous environment.

and carried it along until the CDA program was able to take it over. Mahendra Patel saw the necessity of a corporate CDA architecture and provided the senior technical guidance to give it the appropriate urgency. Andy Wilson and Alan Hasham were key members of the original technical team that worked with me to define the key elements of the CDA architecture, and helped to educate other engineering groups about CDA requirements. Butler Lampson provided critical help in the early days of CDA to set DDIF development on the right course. Bob Ayers helped to clarify our thinking about the proper relationship of CDA with the ODA/ODIF standard under development. Leszek Kotsch provided us with his own valuable technical insights as well as the contacts with external research and standards people we needed for early external technical review. Finally, there are the literally dozens of immensely bright and committed engineers who reviewed all the early drafts of specifications, found problems and suggested solutions, built baselevels of their products to use early CDA support implementations, and suffered all the pain of having to redo things that already worked, in order to track the evolving specifications and make interchange a reality; thank you!

General References

International Standard ISO 7942 for Information Processing Systems, Graphical Kernel System, International Standards Organization (1985).

International Standard ISO 8879-1986(E) Information Processing, Text and Office Systems, Standard Generalized Markup Language (SGML).

International Standard ISO 8613 Information Processing, Text and Office Systems, Office Document Architecture (ODA) and Interchange Format (1989).

CCITT X.400 standards for mail and messaging (1988).

CDA Reference Manual, vols. 1 and 2 (Maynard: Digital Equipment Corporation, Order Nos. AA-PABUA-TE and AA-PABVA-TE, 1989).

William R. Laurune
Robert L. Travis Jr.

The Digital Document Interchange Format

The DDIF document interchange format is one of the central data formats of the CDA architecture. The design of the format was driven by the user demand for increased data portability and system support for more sophisticated document processing capabilities. The DDIF format supports highly integrated text, graphics, images, and application data. A major goal was to design the DDIF format for acceptance as a standard document format. The design includes easy and speedy data access, minimal storage size, high-quality data representation, revisability, and format extensibility. The extensibility of the format makes it easy for users to accommodate individual and future needs.

This paper presents an overview of the CDA architecture data format, the DDIF document interchange format. Since the DDIF format is described in great detail in the CDA manual, this paper provides the *Digital Technical Journal* audience with insight into the concepts and motivations behind the development and implementation of the DDIF format.[1] It also presents an overview of the design of the DDIF format in a less formal manner than either the DDIF standard or the CDA manual.

Motivation for DDIF Development

Looking back to the mid-1980s, we can readily see that the computer hardware being produced by Digital was changing in a number of ways. The simultaneous increase in processor speed and decrease in the size and cost of memory made a computer of a given capacity less expensive and more widely available, especially in the form of personal workstations. Equally significant for document processing, cheaper memory and microprocessors made it economical to build display devices in which each bit of memory was mapped to a pixel on a video display or to a dot on a printed page.

A display device that is driven from such a mapping of bits to pixels has a tremendous advantage over a device that can only display a limited set of characters on a fixed grid. A pixel-addressable device, like a modern laser printer, can display complex graphics, text of any size and style, and, of course, scanned images. The utility of bitmapped displays to convey information is a quantum leap from character-cell devices like the video terminals and impact printers of the early 1980s.

Still, the potential power of a bitmapped display was not immediately available to computer users, who cannot reasonably assemble bitmapped pixel displays bit by bit. What was clearly needed was a host of applications that allowed users to transform their ideas into pixel images that could be displayed on the new devices.

As bitmapped devices began to grow in popularity, several groups within Digital began to plan the next generation of office software to take advantage of these new workstations and display devices. To the authors and their management, it was clear that the new hardware and software would gradually replace the previous generation. It was equally clear that the forthcoming software would create a host of new problems if no standardization of data representations was in place.

In particular, data portability, one great advantage of the existing software, was threatened by the explosion in the number of office applications. At the time, hardware and software alike generally accepted ASCII. A user could create a document in ASCII, distribute it by means of electronic mail, and print it without special conversion or processing. ASCII text files could also be used as data files for software applications, including operating system utilities.

The software applications in existence for more advanced devices already suffered noticeably from an inability to interchange data and from a lack of operating system support. Users could not change editors, mail their documents, or submit them to conventional print queues. The foreseeable boom in graphics, image processing, and advanced text

applications threatened to compound the data portability problem by introducing a host of new data formats.

The solution to many of the data portability and system support problems was to find or develop a standard data format for high-quality text, graphics, and scanned images. The Digital standard for document interchange ultimately became known as the DDIF document interchange format. This paper presents some of our goals for the DDIF format, describes the basic design of the DDIF format, and highlights some of the design decisions we made along the way. Other papers in this journal describe some of the facilities provided by the CDA architecture that allow much of the ease of use and data portability formerly provided only by simple ASCII text files.

Goals

From the outset, our goal for the DDIF format was that it eventually serve as the common currency of document processing, just as the ASCII text file does on Digital systems today. Moreover, the DDIF format would have expanded capabilities that included graphics, sophisticated text, and relationships to external data.

As a common coin of the realm, the DDIF format would offer users tremendous power and flexibility. The 1980s computer user could create, edit, mail, print, view, search, and compile text files. We wanted the 1990s user to be able to create complex documents for which the same set of basic services was available.

For the DDIF format to be acceptable as a standard document format, a number of challenging goals had to be met.

Speedy access — Reading a DDIF file could not require so much memory or CPU time that users would not accept the DDIF format as a natural storage format for their data. Stated otherwise, after allowing for expected near-term increase in processor speed, we wanted processing of DDIF documents to take about the same amount of time (or less) as processing a simple text format on the early 1980s generation of computer hardware.

Minimal size — Relative to the kind of data being stored, the storage overhead of using the DDIF format could not rule out its use by many of the products used on Digital platforms.

Ease-of-access — Unless application developers could call a set of toolkit routines to read and write the DDIF files, it was not likely that the format would become popular. The design of such software layers required a clear division of function between application and toolkit.

Highly representative — To serve adequately as a document storage format, the DDIF format had to be able to represent the data each application would need.

To some extent, the vast number of features that users of high-end document processing systems expect pushes the DDIF format away from its goals for easy and speedy access and more toward complexity. Many of these features are based on the traditions of the publishing industry and do not necessarily fall into the boundaries of any formal scheme or theory of document processing. The way to keep the DDIF format representation small was to seek the smallest basic units of document representation that could be combined to build the more complex units. Later in this paper, we explore how the various elements of the DDIF format are combined like atoms to create the complex molecules of documents.

Another set of goals for the DDIF format is related to the success of products that rely on the format for representing documents. For products based on DDIF to be competitive with similar products in the market, the format had to

- Be highly revisable. The native format for documents must support document editor features that make revising a document as easy as possible. A revisable document contains many interelement relationships that allow revision. For example, a printer protocol might describe text as being fixed on specific pages, but a revisable document format describes the rules for the flow of text.

- Provide a modern imaging model to display complex pictures. The conceptual basis for describing a picture is an imaging model, which describes the operations that can be performed on a video display or laser printer page under formation. Applications use the basic operations provided in the imaging model to build complex pictures. Imaging models vary in the number of operations required to build complex pictures. Imaging models can also be deficient. For example, some older imaging models failed to allow for the compact description of curves.

As a revisable document format, the DDIF format is necessarily influenced by the prevailing capability of printers and video displays. It would hardly be a service to users to allow them to create documents they cannot accurately view or

print. On the other hand, a document storage format that does not provide the user with access to the most useful features of modern laser printers can hardly hope to form the basis for competitive products in today's market, much less in tomorrow's market.

- Provide an application integration capability. For a document interchange format, application integration support encompasses the ability to represent references to external data sources, and applications to invoke to process such external data. For example, a chart in a document might be produced by a charting application that converts user-supplied parameters and a designated spreadsheet into a chart that is ultimately represented in the document interchange format's native graphics primitives.
- Provide extensibility for future needs. Two forms of extensibility are needed in a format used for both data interchange and storage of user data. The first form is the ability to extend a format in a standard way to represent new semantics for interchange. The second form of extensibility is the ability to allow individual applications to represent private semantics that must be retained from one session of the application to the next.

DDIF Design

Each DDIF encoded document consists of three variable-length elements: the document descriptor, the document header, and the document content. (See Figure 1.)

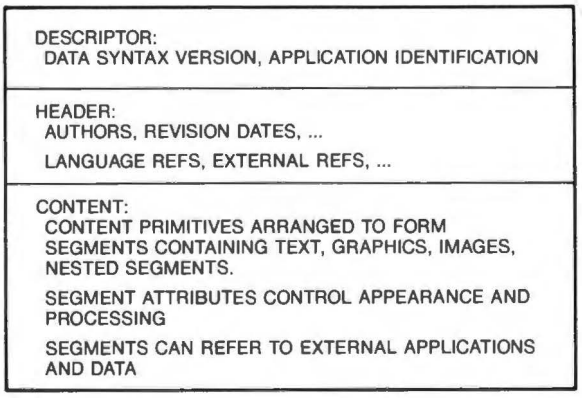


Figure 1 Top-level Structure of a DDIF Document

The descriptor contains information about the revision level of the data syntax and the name of the application that created the document. The document header contains global document-specific information such as time and date of creation and a list of authors. The document content generally contains the largest quantity of data, and describes the part of the document that is displayed, printed, and edited.

As shown in Figure 2, the content of a document is organized in the DDIF format as a hierarchy of bounded document content portions, called segments.[2] Each DDIF segment may be text, graphics, image, and/or nested segments. Additionally, a segment's content may be computed by a standard or application-defined function.

Each segment represents document content that is distinguished from its surrounding content by a difference in its presentation or processing. As a result, each segment has a unique set of attributes that apply to the presentation and processing of the content. Attributes not declared for a given segment are generally inherited from the segment that encompasses that segment. The values of attributes bound to a segment override the value of the same attribute bound to an outer-level segment.

The DDIF standard prescribes initial values for each presentation attribute. These are supplied by each receiving application or by the CDA Toolkit on behalf of the application. The fact that each segment declares only the difference between its attributes and its parent segment's attributes keeps document size minimal.

The nesting of segments and the declaration of segment attributes can be compared to the use of begin-end program segments in a structured programming language. In a structured language, variables can be declared for each segment. When a programming language allows nested declaration of variables of a given name, then variables of innermost scope take precedence when that name is referenced.

The scoped method of structuring document content was chosen for the DDIF format in preference to the alternative "modal" form, where an attribute remains in effect until explicitly changed. For revisable documents, the scoped approach allows a segment to be skipped entirely without rendering the attribute state ambiguous. Conversely, a segment can be inserted into the document without resetting the previous state.

An application can maintain the current state of the segment attributes by pushing all the changed

```
┌─ begin-segment { segment-type "Section" }
│┌ begin-segment { segment-type "Header" }
│└ latin1-content "Introduction"
│└ end-segment NULL
│
│┌ begin-segment { segment-type "Paragraph" }
│└ latin1-content "Italics indicates that the feature is demon"
│└ soft-directive hyphen-new-line
│└ latin1-content "strated in this document."
│└ hard-directive new-line
│└ end-segment NULL
│
│┌ begin-segment { segment-type "Paragraph" }
│└ latin1-content "Many other benefits of document structure"
│└ soft-directive new-line
│└ latin1-content "are not immediately apparent."
│└ hard-directive new-line
│└ end-segment NULL
│
└─ end-segment NULL
```

Figure 2 A DDIF Document Segment Hierarchy

attributes on a stack at the beginning of the segment and popping the stack at the end of the segment. The CDA Toolkit performs this service on behalf of the application when such a processing mode is selected. Such a service is useful when converting to modal form, as is required when preparing a DDIF format document for printing or display.

Individual units of document content — text strings, curves, images, and so on — are represented by content primitives. Content primitives are generally displayable, but the primitives themselves do not entirely describe the presentation of the content. Each content primitive is presented according to the applicable presentation attributes declared for, or inherited by, the segment that contains the content. The importance of separating the content's shape from the content's processing and presentation attributes will become apparent in the discussion of the more advanced attribute inheritance mechanisms later in this paper.

Content primitives are displayed in the order in which they are stored in the DDIF format, with the caveat that complicated layout scenarios like footnotes might force the content to be displayed on other parts of the current page, or even on another page. However, if all the pages of a document were simultaneously displayed, content would appear to be written in the order in which it is stored in the DDIF document. In general, the order of storage of content in a DDIF document is the same as the order in which the document would be read.

The DDIF format's imaging model was formulated based on the results of a survey of modern page description languages, graphics metafiles, and image transmission formats. Most of the DDIF imaging operators, and all of those being used at present, are at least compatible with the more capable page description languages processed by laser printers. The general model is that of writing on a display surface through a mask. The mask blocks writing where it is solid and allows writing where it is open. The use of the mask permits text and graphics operators to appear in various patterns. The mask model does not provide for complementing an existing pattern, which is common in video display protocols. But complement mode is unnecessary when creating a series of pages with static displays. Editing applications can, of course, use complement mode during the interactive editing process.

Text and Layout

The DDIF text-content primitives can represent text in such alphabets as Cyrillic, Greek, Arabic, Hebrew, and others defined by the International Standards Organization (ISO) in ISO Standard 8859. Ideographic languages such as Chinese are also supported. Digital's worldwide engineering organizations contributed to the internationalization of DDIF text support.

Like all displayable DDIF content primitives, text-content primitives are displayed using the applicable presentation attributes. These include typeface, size, color, and rendition, among others. Text rendition includes such familiar variations on text presentation as underlining and highlighting.

Traditionally, graphics applications and document processing programs have taken different approaches to the display of text. To unify the DDIF format's treatment of text as much as possible, formatting is determined by a special set of attributes, called the layout attributes. Text that behaves as an object in a graphics context differs from conventional document text primarily in the way it is positioned, rather than in the way it is represented or in its rendering attributes. Graphics text is positioned along a finite path — defined in terms of one or more lines, arcs, or curves — along which the characters of the text are arranged. Document text is laid out along a series of paths determined by a formatting function built into the receiving application.

Formatting columnar text can best be envisioned as a process of pouring relatively flexible strings of text into molds formed by the columns of the pages. The creator of a document may provide a set of templates for formatting the content of the document. Formatting templates are described page by page. Each page template contains a set of galleys into which the content of the document is formatted. Galleys are usually rectangular and form the columns, footnote regions, and running header regions of the page. In addition to galleys, graphics may be placed on page templates to serve as logos or page background.

A document's creator may provide a template for each page in the document. Alternatively, one or more generic templates may be specified to be used under described circumstances. Interactive document editing programs often store generic templates for use in creating new pages during editing.

Graphics

The DDIF format provides a set of graphics content primitives that are similar to those commonly used in modern graphics metafiles and page description languages. The basic set of graphics primitives includes polylines, arcs, and curves. Each graphic primitive defines, or contributes to, a path that can be drawn, filled with a pattern, or both. Paths composed of lines, arcs, and curves can also be defined for the layout of text.

The presentation attributes of graphic primitives, such as line-width and fill pattern, are determined by the line attributes in effect for that segment. Attributes that apply to graphics objects include color, line width, line pattern (dash-dot, etc.), as well as parameters for the line-drawing algorithm that affects line-end shape and corner clipping.

Graphics primitives are constrained to occur in a frame. A frame is described as a segment that has frame attributes, which include a coordinate system and a background color. The natural nesting ability of segments allows frames to be nested within other frames. Frame attributes may also describe a "clipping" region that allows frames to provide a window on a graphic picture that hides graphic objects outside the window. Clipping regions need not be rectangular. They are defined in terms of arcs, curves, and lines. A frame's attributes may also define a shape to be used for formatting when text flows around the frame.

A frame may be positioned relative to the text content in which it is embedded. For example, this can occur in a margin at the same vertical position, or below the current line. Alternatively, a frame may be placed at an absolute position on a page. A frame can also behave like a character in text and take part in the flow of text during formatting.

Because video displays and printed pages are two dimensional, the DDIF format's graphics primitives are two dimensional. An application that processes three-dimensional graphics, such as a CAD/CAM system, may create a document that includes a two-dimensional rendering of the three-dimensional object from a given perspective. The DDIF application integration capabilities may be used to store the name of the application or function, along with a pointer to the original three-dimensional data.

DDIF Images

The trends toward faster processors and lower storage costs that have made bitmapped output devices economical have also made image processing technology available to more users. As a result, the use of images is rapidly becoming a normal part of office document processing.

The DDIF format provides means to store many types of images, including a variety of storage and compression techniques, bit-plane organizations, and color mappings. The more commonly used image storage mechanisms are very much like and, in fact, derived from, the various image compression techniques used for FAX transmissions.

The image services library (ISL), a shareable run-time library of image processing routines, allows otherwise limited applications to display and process most of the variations on image storage. For example, an eight-bit-per-pixel image can be displayed on a four-bit-per-pixel workstation. ISL routines also support the creation of DDIF format-encoded images from in-memory image formats.

Images are stored in frames, which provide a reference frame, clipping region, and positioning information. Like graphics, scanned images may flow with text or assume an absolute position on the page.

Revisability Support

This section discusses some of the major ways the DDIF format supports the creation of highly revisable documents.

Catalogs of element definitions are one of the most significant ways to provide powerful labor-saving features in document processing applications. Two major definitional elements provided by the DDIF format are the segment-type definition and the content definition. Respectively, these provide for the definition of collections of attributes and collections of content primitives. These definition methods are described in the subsections that follow.

Both segment-type and content definitions are declared as segment attributes and are available for reference from nested segments. Segment-type and content definitions can be stored in DDIF documents that serve as libraries of such definitions. This allows any number of users to share document styles and/or commonly used illustrations, logos, copyright notices, and so on. When used as a style, the definitions must be declared on the document's root segment that serves as a style.

The CDA Toolkit will, on request, resolve references to segment-type definitions and content definitions. The toolkit applies the attributes of segment types and expands content definitions on behalf of the calling application.

Segment-type Definitions

A DDIF segment-type definition is a set of segment attributes that is assigned a label for reference from a segment. A segment that references a segment type acquires the attributes defined for the segment type. If a segment declares different values for attributes defined for a referenced segment type, the local attributes override those of the definition.

Segment-type definitions are declared at the beginning of a segment. In fact, they are one of the attributes of a segment. As such, they go out of scope when a segment is ended.

A set of segment-type definitions bound to the root segment of a document can serve as the "style" for other documents. Such a definition set provides a means for all the elements of a document to share a common appearance, and for a set of documents to share appearances.

A segment-type definition may also reference a previously declared segment-type definition, which allows users to create local variations on a common style.

Content Definitions

A content definition plays much the same role as the macro facility supported by most programming languages in the sense that the definition is copied directly into the point of reference.

A content definition can be a single content primitive, such as a text string or curve. Or it can consist of a segment that contains multiple content primitives and segments to any level of nesting.

Using content definitions to define and share common data elements reduces data entry. In addition, content definitions can provide a single point of maintenance for data that may change.

External References

An external reference represents data that is required for complete processing of the document in hand. A common use of external references in a document is to include text and other data from another document, either for the purpose of acquiring data from a common source or as a means of breaking a large document into manageable pieces.

The DDIF format's basic external reference mechanism supports several different forms of data sharing. This permits high revisability and information sharing among documents, as well as referencing nondocument data formats for parts of a document.

Externally referenced data may be stored in a file, a library, or other storage mechanisms. Files are the most common storage mechanism today, but document databases are a likely future storage means.

Externally referenced data may be either in DDIF format or in another format. When the externally referenced data is not in DDIF format, a function is usually applied that produces results in DDIF format. A document may be included as a whole, or a single segment may be selected.

All externally referenced document content sources are listed in the document header portion of a DDIF format-encoded document. By listing the external references in the header, all external references can be found without processing the entire document. For example, the DOTS data object transport syntax uses this feature to create a mailing envelope that includes the mailed document and the content of all the designated external references. Each external reference description in the document header specifies the format of the data, the storage system that holds the data (such as RMS), and whether the externally referenced data should be included as the document is transmitted.

External references provide users with a number of very powerful document processing features. These include the ability to share a set of graphics, present a chart generated by a function on a spreadsheet, and break a large document into a number of files while retaining the option to process it as one document.

Computed Content

As was mentioned earlier, the content of a segment may be computed. That is, the content primitives that make up the displayable and processable elements

```
SegTypeDefn {
 type-label "Paragraph"
 type-attributes {
  segment-tags { SegmentTag "$P" }
  layout-attributes {
   galley-based-layout {
    wrap-attributes {
     WrapAttributes {
      wrap-format flush-path-both
      quad-format flush-path-begin
      maximum-orphan-size 2
      maximum-widow-size 2
      }
     }
    galley-layout {
     LayoutAttributes {
      space-before { integer-constant 6 }
      space-after { integer-constant 18 }
      leading { escapement-constant { integer-constant 2 } }
      }
     }
    }
   }
  text-attributes {
   text-font 6
   text-rendition { RenditionCode default }
   text-height { integer-constant 12 }
  }
 }
}
```

Figure 3 Example of Segment-type Definition

```
begin-segment { segment-type "Paragraph" }
latin1-content "Italics indicates that the feature is demon"
soft-directive hyphen-new-line
latin1-content "strated in this document."
hard-directive new-line
end-segment NULL
```

Figure 4 Actual Paragraph Example

of a segment can be derived by executing a function that accepts defined parameters and returns the DDIF format content primitives.

The computed content parameters are an attribute of a segment. When a document processor receives a document with computed-content segments, it may recompute the content of the segment or use the current content primitives.

The computed content capability is used for many situations in which document processors perform content calculation and insertion on behalf of the user. Simple examples of computed content include the numbering of pages, list elements, section numbers, and footnote markers. More complex examples of computed content are cross-references, tables of contents, and indexes.

Computed content is most often text content. However, graphic and image frames can result from computing content, especially if the computing function is an external application.

Examples of Definitions and Content

The examples presented in this section show how complex document elements are built up from very fundamental document-processing concepts, and how a high degree of revisability is introduced into a document.

Figure 3 shows the definition of a segment type labeled for reference as "Paragraph." The subelements of the segment-type attributes data element serve as a style for any segment that references the Paragraph type, as follows:

- The $P segment tag declares that the segment type obeys a set of rules defined for paragraphs. For example, paragraphs cannot contain chapters. The DDIF standard defines the allowable structure, in terms of other segment tags, for tags like $P. Receiving applications can use these tags to allocate data structures during internalization of the document.
- The layout attributes describe how this particular paragraph is to be fitted into the galleys (columns) into which it is formatted. Wrap attributes describe how text is broken into lines. The galley/layout attributes describe the spacing of the paragraph as a whole.
- The text attributes describe the size, rendering, and typeface to be used. Many text attributes, including text color, are acquired by default from the parent of the segment referencing the Paragraph segment type. The ability to acquire default attributes from the environment is a useful feature because it allows the type to fit the surroundings.

Figure 4 is an example of how an actual paragraph is formed. It references the Paragraph segment type defined in Figure 3.

The various data elements of the paragraph serve to define the paragraph as follows:

- The segment type is a reference to a segment-type definition, such as the Paragraph defined in Figure 3.
- The latin1-content data elements contain the text of the paragraph. The character set is defined by the ISO Latin 1 standard, and contains ASCII as a subset.
- Soft directives are formatting and processing commands that represent the line breaks and hyphenations that resulted when the document was formatted. Soft directives can obviate reformatting the document when they are known to be valid, e.g., if the application reading the document was the one that created the document. However, the receiver of the document can ignore soft directives at its discretion. For example, the document may need to be reformatted for display on the intended display device.

- Hard directives convey the same commands as soft directives but are considered mandatory for receivers. The hard new-line directive at the end of this paragraph indicates that the last line of text is to be isolated rather than merged with the first line of the next paragraph. Hard directives usually represent commands inserted at the direction of the human user, such as absolute line or page breaks.

Related Standards

Prior to discussing standards that are related to DDIF, we must establish that standards exist for different aspects of document processing. We can identify the following as playing a role in document processing and information display:

- A markup language permits the insertion of commands or tags into a document that delineate the intended formatting and/or structuring of the document. With markup languages, users can create complex documents at a character-cell terminal. Examples of markup languages include Runoff and various standard generalized markup language (SGML)-based document types.
- A page description language is used to describe the final-form appearance of the pages of a document in a manner suited for transmission to a printer or other display device. PostScript is an example of a page description language.
- A device independent, or DVI, format describes a video or hardcopy display that is not dependent on, or directly supported by, a specific device. Subsequent processing is required to produce a device-specific protocol or a page description language.
- A metafile is a log of calls to a subroutine interface, or a set of commands that can be interpreted to drive calls to a set of subroutines that creates a display. CGM, the Computer Graphics Metafile, is an example of a graphics metafile format.
- A device-independent interface is a subroutine package that allows applications to create displays without depending on the device protocols. The Graphical Kernel System (GKS) is an ISO standard for a device-independent subroutine interface to create graphical displays.
- An interchange format is a data format supported by multiple applications as opposed to a private format supported by only one.
- A conversion hub is a highly representative data format used as an intermediate during conversion from one format to another. Use of a conversion hub allows conversion between N formats using $2 \times N$ converters rather than N-squared converters.

Not all document formats fit neatly into one category. For example, the DDIF format is used both as a conversion hub and as an interchange format.

Two document formats deserve special mention, since they are international standards that will become increasingly important for document processing in the years just ahead. These are the ISO standards Office Document Architecture and Interchange Format (ODA/ODIF) and SGML.[3,4]

Office Document Architecture and Interchange Format

ODA/ODIF is an ISO standard intended for much the same purpose as the CDA/DDIF format. It serves as a conversion hub and interchange format. From a distance, the ISO ODIF standard and the DDIF format appear to be very similar. The basic encoding is almost identical. In fact, the DDIS data interchange syntax is based on the ISO ASN.1 standard used to encode the ODIF standard. Both the DDIF format and the ISO ODIF standard provide a number of revisability features.

To the application developer, and ultimately to the application user, important differences exist between the ISO ODIF standard and the DDIF format. Many of the goals we set for the DDIF format were not set for ISO's ODIF standard, especially application integration support. The ISO ODIF standard also makes literal use of existing ISO standards for text and graphics content. ISO is only now starting to adopt many of the recent advances in imaging models. Also, ODIF defines a means of describing the final format of a document in conjunction with its revisable form. We chose to rely on page description languages and other existing final-form document formats to make the DDIF format revisable form more robust.

Digital's use of the DDIF format does not prevent our support of the ODIF standard in any way. ODIF will eventually be accessible through the DDIF format converter library.

Standard Generalized Markup Language

SGML is primarily a syntax for describing document markup languages. It does not define attributes or

layout mechanisms for text, nor does it provide a standard for graphics or images.

The recommended use of the SGML syntax is to define a document type, which describes the hierarchy of markup tags in conforming documents. Users of the document type then use tags, typically of the form <TAG>, to delineate the structure of a document in terms of chapters, sections, paragraphs, or other elements appropriate to the document type. To display a marked-up document, an application is invoked that parses the markup, applies uniform style to tagged elements, and displays a formatted document. Of course, a structured document can be processed for purposes other than formatting, such as extraction of specially tagged elements.

A document encoded using SGML can be converted to the DDIF format; and, indeed, such a converter is already available in the converter library.

Pragmatics of Document Interchange

This section describes some practical matters of interchanging documents, including how the DDIF documents are represented and interpreted.

Data Encoding

When stored as files on a disk, DDIF documents are encoded according to the rules defined by the DDIS data interchange syntax. The DDIS syntax has a number of advantages over other encoding mechanisms in storing and transmitting compound documents.

- The DDIS syntax encoding takes advantage of a full eight bits-per-byte which allows compact representation of scanned images and other nontextual data. The DDIS syntax encoding method uses tags to identify data elements.
- The encoding method itself does not use relative pointers or offsets. Data elements are located by parsing the tagged data elements. Each element that contains data has a length, so there is no need to reserve special characters as delimiters. DDIS-encoded data is stored in files as a bytestream without record boundaries. The length fields of individual data elements eliminate the need for file-level record lengths.
- Because each data element is tagged, an application parsing the data encoding can detect missing elements, and unused fields can be omitted from the encoding. By carefully using such optional data fields in the design of DDIF, we made the fundamental overhead of encoding document data in DDIF very small — normally about 50 bytes. The overhead occurs primarily in the document descriptor field and takes the form of the name of the application that created the document and the DDIF version being used. After the overhead of describing the document, the DDIF encoding size for simple text would compare favorably with text file stored as variable-length records. Thus the DDIF format can serve as the standard encoding format for documents of a wide range of complexity.
- The data type of each data element is known to the receiving application by virtue of its tag. Low-level data access services used by the CDA Toolkit perform conversion of data types like integers and floating point numbers. Thus, the DDIS syntax serves as a computer-architecture-independent encoding.

The DDIS syntax's data encoding is not appropriate to in-memory manipulation by document processing applications or the CDA Toolkit. When the toolkit reads a DDIF document from a file, the DDIS syntax's encoding is efficiently changed to an in-memory representation that contains the same information. An application that retrieves DDIF document data from the CDA Toolkit may further change the representation of the data to an internal format that is optimized for the kind of processing it performs.

Document Interchange and Conversion

If the DDIF format only provided a common format for document storage, it would still be useful. Document processing applications that use the DDIF format could share access routines and documentations. And, system support and development tools would be widely available. However, the DDIF format does more. It is intended for interchange between document processors of various types, including viewers, formatters, and document editors.

Given a mutually supported document format, a document processing application can accept a document produced by another application. Such a document format is called an interchange format. Alternatively, a document format can be converted to a common representation and converted again to the format required by the intended receiver. The intermediate format is called a conversion hub. Since both formats need a high ability to represent documents, a single format can serve both

uses, as long as it meets the higher efficiency requirements of interchange. From a user's perspective, a commonly supported interchange format is more convenient because the conversion time and resource expense are absent. Native CDA program applications support the DDIF format as the interchange format. To allow users to access formats created by applications that do not directly support the CDA architecture, the architecture provides converters.

Document interchange can take place by a variety of mechanisms, including shared memory, file interchange, and even subroutine calls. Anytime one document processing application reads a document that another has written, interchange occurs.

The mutual support of an interchange format, however, does not magically make every application consistent with another. Some applications will be unable to represent the full semantics of the interchange format internally. Even those applications that can, may be unable to process the information in a meaningful way. The interchange format does provide, however, a single, consistent way for the applications to exchange those elements that they do share. For example, an application that processes text but not graphics can exchange text with a text and graphics application. However, the application will obviously not become a processor of graphics merely by reading a DDIF file.

One generic feature that an application can provide is to store DDIF segments that it cannot process and later re-insert them into the document at the same relative position. For example, a document processing application that supports text but not graphics can set aside graphics frames, either in memory or in a temporary file. Similarly, a DDIF encoded document can be converted to many different formats. However, most existing document processors will probably not be modified for the maximum support of the DDIF format, because it is impractical to change software that is becoming obsolete.

The probable future of document interchange is that Digital and many third parties will evolve new applications to support the DDIF format. For some, the DDIF format will be an output format. For others, such as viewers, the DDIF format will be the native input format. Document editing tools will either support most of the DDIF format or adapt to editing only certain segments, while preserving those they cannot. Over a period of many years, users will convert their existing documents to the DDIF format in order to migrate to newer applications.

Extensibility

The DDIF data format is extensible at three levels: User, Application, and New Versions. Users extend the format by defining new styles (segment types) that combine existing features in new ways, creating new document types. Applications extend the format by adding new processing tags denoting segments with special properties and by supporting computed-content expressions to incorporate new external data types. New versions extend the specification itself, by adding new content primitives (to accommodate such things as sound) or by extending the layout model, for example. Applications also have access to opportunities for encoding private data elements at key points in the DDIF syntax, to accommodate special needs, or which anticipate expected enhancements to the basic syntax. These extension capabilities were designed in from the start, to not conflict with each other, and to ensure the capability for upward-compatible enhancement over time.

Foreseeable Extensions

The expression "a picture is worth a thousand words" is commonly quoted because some ways of conveying information are more powerful than others. Similarly, various means of expression — voice, pictures, motion pictures, printed words, diagrams, and even touch — are more appropriate to convey certain types of information.

The DDIF format is a means of storing and interchanging data intended for presentation to people. Hence, the work of extending the DDIF format will probably go on for years. It may continue until software and hardware technologies have changed so completely that an entirely new concept in document processing is needed. Advances in optical fibers, optical disks, optical switches, and microminiature lasers may very well result in a technology in which color holographic displays with voice input and output are the norm.

In the meantime, CDA applications are already pushing the DDIF format to display color pictures on paper, store synchronized voice, and represent images and graphics that can be set in motion. We should expect to see (and hear) these new technologies in the near future.

Acknowledgments

The individuals who reviewed and contributed to the DDIF format constitute too large a group to be listed here. However, we should especially note the

work of other organizations that worked with the Core Applications Group to make the DDIF format a success. The Image Program Office not only provided the image content and attributes for the DDIF format, but wrote software that made everyday image processing a reality. And the International Products Group defined clearly the requirements for the support of the world's major markets for document processing.

References

1. *CDA Reference Manual,* vols. 1 and 2 (Maynard: Digital Equipment Corporation, Order Nos. AA-PABUA-TE and AA-PABVA-TE, 1989).

2. The DDIF format segments should not be confused with the construct called a segment in the graphical kernel system (GKS). A GKS segment is roughly equivalent to the DDIF content definition, as described in this paper.

3. *International Standard ISO 8613 Information Processing, Text and Office Systems, Office Document Architecture (ODA) and Interchange Format* (1989).

4. *International Standard ISO 8879-1986(E), Information Processing, Text and Office Systems, Standard Generalized Markup Language (SGML).*

Carol A. Young
Neal F. Jacobson

The Digital Table Interchange Format

The recent information explosion has created a multitude of end-user data table processing applications, including database access tools, spreadsheets, charting packages, laboratory automation systems, and electronic business documents. As the amount and popularity of tabular data increases, so does the need to share or interchange tabular data between applications. Within the CDA architecture, the DTIF table interchange format defines an application-independent and architecture-neutral format for the interchange and storage of revisable data tables. The DTIF format uses the DDIS data interchange syntax as the basis for a three-part architecture that defines the syntax and encoding for documents containing revisable data tables, the formula for expressions defining relationships between table elements, and the presentation and other processing characteristics of a data table.

Users have benefited from the recent information explosion in many ways.

Advances in database storage and retrieval technology provide access to increasingly larger and more up-to-date pieces of information. Spreadsheets, charting packages, and decision-support systems help users visualize and analyze this information. Laboratory automation equipment helps scientists capture and analyze data to repeat experiments reliably. Even the "business of business" is becoming automated with the advent of electronic document interchange (EDI) and electronic business documents. Graphical compound document editors enable users to create presentation-quality documents that contain input from these diverse information sources.

However, these specialized data access, analysis, and presentation tools make data sharing between applications increasingly important and more complex. The CDA architecture, the DDIF document interchange format, and the DTIF table interchange format help solve data-sharing and interchange problems.

The CDA architecture facilitates the interchange of revisable compound documents. These documents may contain text, graphics, and data — both internally stored or externally generated. The DDIF format is the storage and interchange format of compound documents. The DTIF format is the storage and interchange format of tabular data (i.e., rows and columns). The CDA architecture also includes a set of services and applications for processing these interchange formats, such as the CDA Toolkit and CDA converter architecture.

The CDA architecture addresses data sharing as a function broader than simply accessing data in a common way. Data sharing in this larger sense is the

- Exchange of data between different applications (e.g., a spreadsheet and a charting package)
- Use of data by different users and across different organizations
- Data representation and exchange between different operating systems and across heterogeneous networks

This paper focuses on the design goals used in the development of the DTIF format, and the structure and type of data contained in DTIF documents and data tables.

Background of DTIF Development

Historically, individual data storage formats have been developed that are specifically tailored to the needs of each application. As such, these formats may not be easily adapted to the requirements of another application. Further, rather than adapt an existing format, application developers often prefer to define another data format. As a result, these specialized formats complicate data interchange. To share data, an application must explicitly program support for another application's format.

Therefore, as the number of applications that need to share data increases, the code needed also increases.

A more efficient approach is to have each application use a common data format. Common data formats are becoming more available, but there are still some problems.

For example, in the spreadsheet application domain, various data formats are commonly used and thus become de facto standards. These include applications that provide direct support for reading or writing to the standardized format. Another common approach provides conversion utilities to translate the application's native format to the standardized format. Conversion can be costly, however.

Moreover, a de facto standard is originally developed for a particular application. If a feature is not present in the original standardized format, it cannot be represented there. Such features can become lost in the translation from one format to another, and the intent of the original data may be lost.

Finally, these more commonly used applications have no single encoding format, or methodology, that addresses database applications as well as spreadsheet applications.

We wanted the DTIF format to resolve these types of problems, not perpetuate them. Therefore, we built the format from the basic elements common to all data tables and processors. Tables contain columns, rows, and cells. Cells contain values, and values are usually numeric or textual. These common elements form the least common denominator for all data tables. This basic information describes a data table for inclusion within a report or mail message. However, this approach does not address the issue of revisability. Additional issues had to be addressed for the DTIF format to support a high level of revisable data interchange.

- Are certain values in the table computed?
- Which formulas were used in the calculations?
- Are table values constrained to a certain type (e.g., the value must be an integer)?
- Which section(s) of the table is to be displayed or printed?
- Do table values represent a particular kind of value (e.g., currency or percentages)?
- How should these values be displayed?
- Which currency symbol or radix point is appropriate?
- Are certain values hidden or unmodifiable?
- What is the default value to be used in place of a blank cell?

By addressing these issues during the design, we developed the DTIF format to specifically facilitate the interchange of revisable tabular data between applications. Typical tabular data examples include results from database queries, spreadsheets, and input to charting packages. Other possible sources include laboratory equipment, EDI documents, or almost any application generating data structured in tabular form.

Relationship to the DDIS Syntax

Data interchange can take various forms. It can occur between applications, across operating systems, and across hardware architectures. To facilitate all three forms required defining a format that was independent of any particular application, file system, or hardware architecture. To meet these needs, the DDIS data interchange syntax was developed.

The DDIS syntax defines the set of conventions for the use of ASN.1, a standardized method for defining and encoding syntaxes used for data interchange.[1] The DDIS syntax provides an architecture-neutral method to define data syntaxes and encode the syntactic elements as a self-describing byte stream.

The DDIS syntax uses tag, length, and value fields to describe the values within an encoding. Each element is uniquely tagged within its context and is self-describing. Both primitive and constructed element types can be combined to create a grammar that describes the order and type of information that may be present in an encoding. Therefore, a grammar describes the syntax and encoding for a particular data class, which is called a domain. Since a grammar is defined as a set of self-describing elements, generic processors can be constructed to handle all grammars without being tied to any one in particular. The CDA Toolkit is an example of a generic processor that handles encodings from both the document and data table domains.

The DTIF format uses the DDIS syntax to define a syntax and the encoding for revisable data tables, including the order and type of information which can appear in a data table. The encoding defines the method for writing this information to an output stream. An instance of a syntactically valid DTIF encoding is called a DTIF document.

The following information may be included in a DTIF document:

- The logical structure of one or more tables (dimensions and definitions for the table columns, rows, and cells)
- The formulas defining the relationships between table elements
- The last computed cell values when the table was encoded
- The presentation characteristics, including support for international applications and for multiple views of the table data
- The constraints that apply to various table values, such as data typing or size restrictions
- The generic columns, or column attribute type definitions. Generic columns may be referenced by table columns to provide the default attributes for a column.

The DTIF format supports efficient encoding for both densely populated and sparsely populated data tables. Sparsely populated tables contain blank or missing values.

Attributes applicable to certain table elements, such as all cells in the table, may be defined and inherited hierarchically. Attributes defined at one level of the hierarchy provide the defaults for attributes at lower levels. Redefining an attribute at a lower level overrides the default. Generic columns and the inheritance mechanism allow efficient encoding of commonly used data table attributes.

Interrelated Architecture

The DTIF format is composed of three interrelated architectures.

- Part I defines the syntax and encoding of revisable data tables.
- Part II defines the syntax and encoding of expressions.
- Part III defines the syntax and encoding of edit strings.

Part I references Parts II and III. Cell and column expressions are defined using Part II, and format information is defined using Part III. Parts II and III were also designed to be used independently. All three parts are discussed in more detail in the section Tables.

Implementation

Digital's precursor of the DTIF format was generic table encoding in the 1985 release of the VAX TEAMDATA product. Generic table encoding supported data exchange between the VAX TEAMDATA spreadsheets and data table editors. This implementation included many of the same features or features similar to those in the DTIF format, such as metadata support, formulas, inheritance, and sparse encoding. The implementation was even based on a precursor of the DDIS syntax.

A more robust generic table encoding design was developed in 1987 and used in VAX DECalc version 3.0a, a Digital spreadsheet processor. This implementation became the native storage format for VAX DECalc version 3.0a spreadsheets.

The DTIF format offers major improvements over both of the previous generic table encoding designs. It permits data sharing among the spreadsheet, database access, and business graphics components of DECdecision software.[2]

Design Goals

In this section, we discuss the design goals achieved in the development of the DTIF format. These goals included:

- Application-independent interchange
- Revisability
- Internationalization
- Modularity
- Compact data representation
- Sequential processing
- Extensibility
- Support for private extensions
- A robust feature set

First, a DTIF document is application-independent. An application reading a DTIF document need not know anything about the application that created the document. The converse is also true: an application creating a DTIF document need not know anything about the application that reads the document.

A DTIF document is self-contained. The tabular data and the semantics needed to process it are included in, or defined by, the DTIF document itself. A DTIF document is also self-describing. Each element describes itself in terms of its DDIS syntax

domain. An application can, therefore, process DTIF documents created by a multitude of applications without having any special programming knowledge about any one.

To support revisability, the DTIF design allows revisable data to be encoded as well as the final form data. Formulas and relationships, as well as the data values, are captured in the encoding. Applications that understand these relationships can re-derive the data values and allow the user to explore the implications of changing certain values. This revisability is especially beneficial in supporting document interchange among scientific and laboratory users, where experimental, test, and formulaic data may need to be revalued.

The DTIF format also facilitates interchange between international applications. The document header contains a list of languages used in the document. Languages are identified by a language symbol and country code from Annex B and Annex D of the ISO standards 639 and 3166. In addition, a DTIF document may contain one or more language preference tables that define information particular to a country, language, or user. A single document or table can contain multiple country or language support. Language preference tables are described in more detail in the Document Structure section later in this paper.

In addition to language preference tables, the DTIF format supports strings that contain text from multiple character sets (8-bit, 16-bit, or 32-bit). Such strings can be stored in table cells as the title of a document, or as descriptive information attached to tables, rows, columns, cells or windows.

We needed to build into the DTIF format a set of hierarchical relationships between layers, and to provide a scheme for inheritance of attributes. The hierarchical design reinforces or models the logical structure and revisable nature of a data table and helps modularize processing. The inheritance mechanism reduces encoding space. The same attribute need not be encoded at each level in the hierarchy, and the logical table structure (analogous to variable scoping in a programming language) is reinforced. Thus, the DTIF format is a modular design, with distinct logical layers.

Related data items, such as column attributes or cell data, are grouped together. Outer layers of the document structure contain global and generic information. Inner layers refer to more specific information, such as cell values.

As noted earlier, the DTIF design accommodates applications that deal with sparsely populated and densely populated data. To do this, individual cells, groups of cells, and entire rows or columns that are not specified may be omitted from the encoding. Data representation is more compact and efficient, and data organization is not lost.

Through the use of inheritance, creators of DTIF tables may define commonly used attributes at higher levels of the hierarchy. Default definitions can exist at various levels, with overrides specified only when necessary.

In addition, generic columns and named edit strings may be used to share cell types and formatting types within a document. For example, an edit string named ZipCode might contain standard formatting attributes for the display of zip codes. Changing the attributes of all zip codes involves only changing a single edit string, which further reduces the size of data encoding.

Although some features of table processing may require multiple passes over the data, DTIF encoding does not require nonsequential access. Processing proceeds sequentially, using less memory and CPU time. Sequential processing also ensures that the DTIF format is generally usable on smaller systems.

Because extensibility was a major goal, the entire DTIF domain may be extended to support new features and capabilities. The DTIF grammar can be extended upward over time. Many elements can also be extended without grammar changes. New preference table items, collating sequence names, and edit string names may be named and used by groups of applications. To avoid conflicts with other applications, certain naming conventions should be used.

Preference item names include an application prefix, a dollar sign (for Digital applications) or an underscore (for non-Digital applications), and the item name, e.g., ECALC$ITEM or XYZCORP_ITEM.

Private extensions of applications can also be added to the basic DTIF structure. Some uses of private extensions include:

- Specialized display attributes, such as fonts and color
- Support for data types not included in the DTIF format or specialized semantics for a given data type or table
- Printing information, such as report layout or printer settings

Private extensions are defined by an individual application or by a set of cooperating applications. They are used to store information that is not defined by the DTIF format. Examples of private extensions include editing context data, report layout semantics, and format attributes for color. Private extensions may also be used to enhance DTIF semantics. For example, a format type may be set to currency, with a private extension indicating that negative values are to be displayed in red.

Private extensions are defined as a sequence of name/value pairs. The name is an ASCII character string which uniquely defines the data value. Its format includes an application prefix and a data item name, separated by an underscore (e.g., XYZCORP_XXX). The data value can take a variety of forms:

- A single, structured value (e.g., an integer)
- A sequence of structured values (e.g., a list of integers)
- An unstructured value (e.g., a byte stream)
- An externally defined value (e.g., a DDIF document)

Private extensions may be added at most levels within the DTIF domain. While private extensions may be included in a DTIF document, an application must not require that the private extension be present in order to process the document.

The DTIF format is intended to encompass a wide range of data-processing requirements, from relatively simple, to highly sophisticated and interrelated. However, the DTIF format is not designed to serve as the standard for all enumerable table features.

In general, we looked toward spreadsheets, database products and tools, and dictionaries to develop the DTIF feature list. In choosing features, we tried to avoid those that were too specialized and those that could not be generalized to a larger set of applications. This is not as restrictive as it sounds, because applications can still encode specialized features using private extensions.

Another criterion used in choosing features for the DTIF format was their relevance to most, if not all, applications. For example, window information is not considered interesting by some applications. Windows are used by spreadsheet applications to define one or more visual views into a table. Database access applications have their own concept of a view based upon a selection criterion that is very different from that of a window. Window information is, therefore, not really useful for interchange between database and spreadsheet applications since neither supports this feature in the same way. Early DTIF designs did not include windows. However, when interchange between spreadsheet applications was considered, window information became relevant and was added to the DTIF format.

The remainder of this paper discusses the syntax and semantics of a DTIF document. Some of the items discussed are required in every DTIF document, and others are optional. The CDA Toolkit provides a design center for creating and processing DTIF documents for four reasons.[3] First, the toolkit enforces and guarantees the creation of correct documents. Second, it controls versions by checking for obsolete versions and automatically upgrading documents to newer versions. Third, the toolkit provides a level of abstraction between the details of syntax/encoding and the document semantics. The application developer can concentrate on semantics rather than encoding/syntax issues. Fourth, the CDA Toolkit facilitates application development by providing standardized access and storage of DTIF document elements and constructs.

Document Structure

A DTIF document consists of three sections: the document descriptor, the document header, and the document tables. The document descriptor identifies the DTIF version used to encode the table and the application which created the encoding. The document header contains the document title, creation date, and the resources used in processing the table. Resources include a list of external references, a list of languages used in the document (defined according to ISO standards 639 and 3166), language preference tables, and generic column attributes. Document tables are discussed in detail in the Tables section.

Resources

External references define the source for any externally defined content contained in the document tables. This information is included in the document header to inform an application early in the document's processing that the document contains externally defined data. This notification permits the application to act or prepare itself accordingly (allocate memory buffers, alert user to external content, etc.).

External references are defined by the data type of the external content as an object identifier (e.g., DTIF, DDIF), a description of the data type (e.g., human-readable), a label used to locate the external content (e.g., file name), and a copy-on-reference flag.

The copy-on-reference flag indicates whether or not an external reference is copied when its root or parent document is mailed. The DOTS data object transport syntax within DECwindows Mail reads the DTIF documents and processes external reference tags. Root documents and their copy-on-reference external documents are packaged into a single message and then unpacked on the receiving end. This packaging is really a recursive procedure that extends to all child documents.

Language preference tables define table-processing information that is specific to a particular language, country, application, and/or user. A preference table includes display formats for particular data values, collating sequences, the language to use when processing the table, and other language-specific items, such as the names of months. A preference table may also contain a sequence of edit strings that are used throughout a document.

A DTIF document may contain more than one language preference table. Each language preference table is assigned an index number, which identifies its position within an array of preference tables. The first preference table has an index of 1, and subsequent preference tables are numbered sequentially. Within a DTIF document, a language preference table may be referenced by its index wherever format information is stored.

A good example of how language preference tables can be used to extend tabular data interchange capabilities is monetary values.

The currency symbol, radix point, and digit separator character(s) used in one country may not match those used in another country. In this case, if a table created by an application in one country is processed or displayed by an application in another country, the results might be very misleading. One application might encode the amount one dollar and 20 cents, as $1.20. Another application may decode and display this value as francs and centimes (e.g., 1,20Fr). Since the exchange rates are not equal, this value is incorrect. To ensure correct interpretation of currency values, a language preference table defines the currency symbol, radix point, and digit separator for use with the table.

Generic columns are another resource used by the DTIF format, because data tables often contain columns with similar or identical attributes. In database applications, a global field defines attributes for column types. The global field is referenced for attributes by one or more table columns. Global fields help reduce storage space, because common attributes are stored only once. They also provide a convenient way to indicate that these fields have related data types and attributes. A change to the global field affects all columns that reference it. Global fields are encoded in the DTIF format as generic column attributes that define attributes' column types. For example, a money column type may be defined to contain numeric values with two fractional digits and a particular display format. The attributes for a generic column are the same as those for a table column.

A table column explicitly references a generic column to become a specific instance of that generic type. It then inherits all attributes associated with that type. These inherited attributes apply to all cells in the table column. If an attribute is specified in both the referenced generic column and the table column, the specific attributes override generic attributes.

More than one table column may reference a generic column, and each inherits the generic attributes. For example, in a seven-column table, each column can contain floating point values representing currency, and each can have a specific display width. The values in the first six columns can be left-justified within the column, and the seventh column's can be right-justified. A generic column type could be defined with the attributes floating point, currency format, left-justified, and display width. All seven table columns would reference the generic column attributes, thus inheriting all the attributes. The seventh column, however, would also access a specific format attribute for right justification. In this case, the specific right-justification format attribute would override the generic left-justification attribute.

Tables

The structure of a DTIF table is shown in Figure 1 and described more fully below.

Table Metadata

Column Attributes Table column attributes define the number of and the attributes for each column within a table. A table column is identified by a

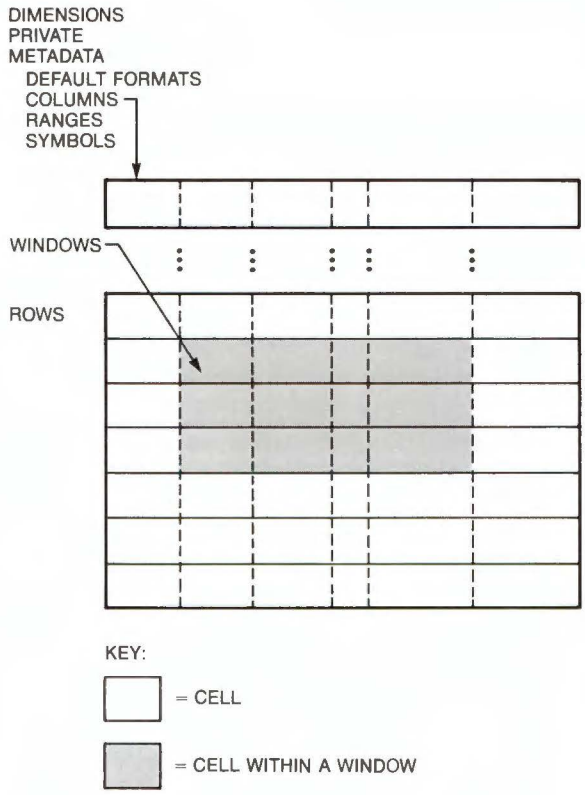


Figure 1 DTIF Table Structure

name and an integer value. Descriptive information can be used to record the column's revision history or other explanatory information. The header text and query string can be used for report headers and shorthand references to the column. Other attributes include

- The formatting information to be applied when displaying cells within the column
- The type of data values the column is expected to contain
- A scale factor applied to each value in the column
- An expression to compute values in the column
- A value denoting null or missing values in the column

A column attributes element must be defined for each column in the table containing cell data. The DTIF format does not support implied columns where the definition of a column is derived from the existence of a cell stored within the column. In other words, there must be a table column attributes definition in order for cells to occupy a given column. Every column attributes definition, however, need not have cell data associated with it, e.g., empty columns. In this case, the column attributes definition is used to hold a place for the column within the table.

Windows, Ranges, Symbols A window identifies a particular view of a table. It is used primarily by spreadsheet applications to define display-specific information pertinent to a table. Each view references a set of cells that is part of the window, and describes which cells are scrollable and which cells are fixed or locked in place. Cells may also have specific window format information, which gives a truly separate view.

A cell coordinate identifies a particular cell in the table. A cell coordinate is defined by a column identifier plus a row identifier. The coordinate is used in range definitions and formulas to indicate a position within a table or window.

A range is a logically related set of cells, rows, and columns, or any combination of these. Ranges are defined as one or more pairs (start-position, end-position) where each value is a cell coordinate, column identifier, or row identifier. The end-position may be omitted if the two positions are the same. This allows a simple and compact encoding for ranges consisting of a single cell, column, or row. A range can be named, and the name can be used when referring to the range in expressions or other range definitions. The DTIF format supports range names composed from multiple character sets. A range may also be tagged to further indicate its use as a titled or scrolling section of a window, or as a sort range. The range definition syntax is used in DTIF Parts I and II, which were defined earlier in the Background section of this paper. Expression ranges are encoded in the same manner as table ranges.

A symbol is like a named global value that can be used similarly to a cell or range reference. It may be a primitive value, a list of values, an expression, or an external variable. As an external variable, applications may use it to reference a variable name or some externally named value. Symbols are used in DTIF Part I and Part II.

Table Rows and Cells

The data portion of a table consists of a sequence of rows, each of which contains a sequence of cells.

The contents of a cell correspond to the intersection of a particular row and column within a table. Any column with cells must have a corresponding

column attributes definition defined in the table metadata. A cell is identified by an integer value that corresponds to the integer value of the column containing the cell. Cells are stored in increasing order within a row. Empty cells may be omitted from the table.

A cell may contain the following information:

- A state indicating whether the cell contains a value, is empty, or contains an error value
- The last computed value stored, for example, integer, floating point, text, date/time, scaled integer, and complex
- An expression used to compute the value. (Note: Expressions are defined using DTIF Part II, canonical form expression (CFE), described in more detail in the next section.)
- The formatting information defining presentation characteristics. (Note: Some formatting information is defined using DTIF Part III, edit string format (ESF), which is described in more detail in the Edit Strings section.)
- A description
- The application-private data

Formulas

Canonical form expression (CFE) is defined in Part II of the DTIF format to describe the syntax for encoding expressions stored within or outside a DTIF document. CFE is also based on the DDIS standard for encoding, but it is a separate domain from that of the DTIF format. This separation enables CFE to be used independently of a DTIF document and to be used by applications that need a common way to specify expressions, e.g., database query tools and dictionary tools.

Expressions are used to calculate a value for a particular cell or all values within a column. Expressions are also used in symbols. CFE is based on several expression architectures including the following:

- IREP — An internal representation format that expresses spreadsheet formulas and is used in VAX Xway. IREP uses an infix notation for describing tokenized expressions.
- VBLR — The VAX Information Architecture binary language representation that expresses relational database requests. VBLR uses a tag-length-value prefix notation scheme to describe expressions.

All CFE expressions are defined using function notation. Each function is defined as a DDIS tag-length-value item. The tag field identifies the function and the value field identifies the function arguments, if any. A function argument may itself be an expression or a recursive definition that provides support for simple or complex nested expressions. Function notation was selected over other potential representations such as prefix, infix, or postfix, for a number of reasons.

- Applications can easily identify and process the expressions. Each tag identifies the function and each value identifies the arguments. All arguments are explicitly scoped within the function. No counting or fixed argument mechanism is needed as in prefix, infix, or postfix representations.
- Applications can easily identify those functions (subexpressions) that they cannot evaluate.
- The function notation provides a straightforward mapping into a DDIS grammar, and the grammar can be used to check the expression syntax.

CFE contains a comprehensive set of features found in the above architectures and others. These features include commonly available functions found in most spreadsheets, as well as some esoteric functions. The actual features are too numerous to mention; however, their categories include literals and variables, basic arithmetic functions, Boolean and relational expressions, statistical functions, conversion functions, character string manipulation functions, choose and lookup functions, date/time functions, cell-related functions, financial functions, series functions, trigonometric functions, transcendental functions, binary functions, miscellaneous functions, and private functions.

Private functions allow an application, or group of applications, to extend the CFE encoding to support private encodings. A private function may either refer to an external function or a function internal to the application.

Applications are not expected to support all CFE functions. However, when an application finds a CFE function it does not support or cannot understand, it is expected to flag the problem to the user. Under such circumstances, certain applications also display the formula as a text string.

Format Information

Format information defines presentation characteristics for a table value.

Format attributes may be defined as specific to a particular data type (numeric, text, date/time), or as applying to a value of any data type. The DTIF format defines a number of standard format types (such as $MONEY, $PHONE, $FLOAT, or $INTEGER) and also includes a mechanism for storing application- or user-defined edit string formats (ESF). Other format attributes include display width, alignment, and rendition information (e.g., italicized, bordered, hidden). Some format attributes are relevant only in conjunction with other attributes, and some attributes have no significance if other attributes are present.

Format information can be specified at, or inherited from, several levels within a DTIF document. Although each attribute is inherited independently, format attributes are grouped together at each level of the document. If an attribute is not specified at a given level, it is inherited from the next higher level. If an attribute is not specified at all, the application must select an appropriate default. In order of decreasing precedence, the levels are as follows:

- Cell format attributes apply to the particular cell in which they are stored. Cell attributes override any attributes inherited from the column, row, window, or table.
- Column format attributes *or* row format attributes apply to all cells in the column or row of the table. Either column or row attributes take precedence as indicated by a flag in the table metadata section. The default value for the flag is format-by-column, chosen because most database applications are column-oriented.

  Table columns may reference generic column attributes and inherit all those attributes. If the table column explicitly defines an attribute already defined in the generic column, the table column attribute takes precedence.
- Window format attributes, if specified, apply to all columns, rows, and cells displayed in a particular window. Each format attribute element includes an optional window identifier that indicates the window to which the format attribute applies.
- Table format attributes define format attributes that apply to all columns, rows, windows, and cells of a table.

Figure 2 illustrates attribute inheritance.

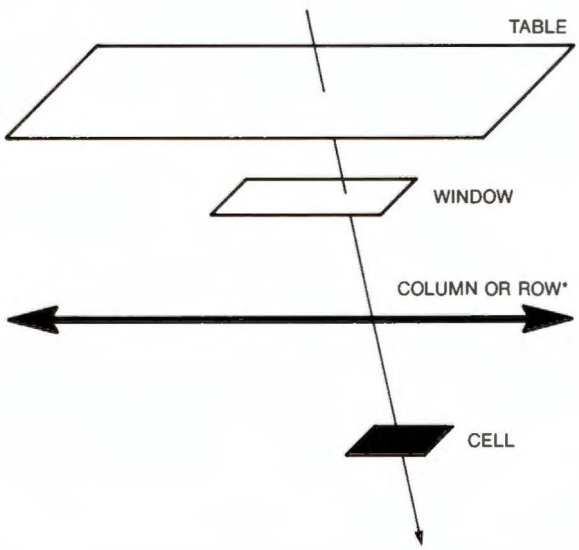
Edit Strings

Edit string format (ESF) is defined in Part III of the DTIF format and describes the syntax for encoding user-defined edit strings stored within or outside a DTIF document. ESF is also based on the DDIS standard for encoding, and is a separate domain, for the same reasons as CFE. The requirements for ESF were based on features taken from COBOL picture strings, VAX TEAMDATA, VAX DATATRIEVE, and several spreadsheet products.

While the DTIF format provides edit strings for predefined numeric, date, and text data type formats, ESF allows applications to extend this capability to define and apply their own formats. Edit strings are used to define format information that is specific to a particular language, country, application, or a particular user's preferences. Within a DTIF document, the language preference tables provide this level of specification. Edit strings may be included within a DTIF document at the table, window, row, column, or cell level.

The edit strings for standard format types or other format-related information may be defined in a language preference table. The language preference table can be referenced by its index in a format attributes element.

An ESF edit string pattern is defined by a sequence of one or more edit string tags, each of



*COLUMN MAY REFERENCE GENERIC ATTRIBUTES. COLUMN OR ROW TAKES PRECEDENCE ACCORDING TO TableMD/tmd-flags VALUE.

Figure 2 Attribute Inheritance

Edit String	Data Value	Edited Value
`{`		
`str-literal "Profit was: ",`	150	Profit was 150 dollars
`minus-literal-begin "(",`	−150	Profit was (150) dollars
`decimal-digit,`		
`decimal-digit,`		
`decimal-digit,`		
`minus-literal-end ")"`		
`str-literal " dollars"`		
`}`		

Figure 3 ESF Edit String Example

which applies to a particular data type. Each tag specifies the format of the next insertion character or sets a mode on further formatting. Some example tags include decimal/octal/binary/hex digit, digit separator, radix point, sign, exponent, currency symbol, month, day, year, weekday, and reverse. (Note: The digit separator, radix point, and currency tags each have two variants. One is sensitive to the language preference table, and one takes an explicit literal.)

ESF provides edit strings for special sequences as well. These include minus literal insertion characters for negative values, replacement characters, floating characters, and repeat sequences. The ESF can also be further extended by use of application-private data.

Figure 3 illustrates an ESF edit string.

Summary

The DTIF format defines the syntax and encoding for documents containing revisable data tables, such as database query results, spreadsheets, and scientific experimental data. The DTIF format enables application-independent data tables to be interchanged through a high degree of revisability, support for international applications, and compact representation of commonly used table attributes and densely and sparsely packed data tables. The three-part DTIF architecture defines a robust set of data table attributes, and includes support for application-defined extensions.

Acknowledgments

The authors wish to acknowledge Mark Anderson, Linda Busdiecker, Michael Glantz, Scott Kardon, Paul Reilly, and Tony Vlatas for their technical contributions to the design and testing of the architecture, and Leanne Olson and Dan Laukaitis for their contributions to the DTIF architecture specification.

References

1. *ISO DIS 8824 Information Processing — Open System Interconnection — Specification of Abstract Syntax Notation One (ASN.1),* (July 10, 1986), *ISO DIS 8825 Information Processing — Open System Interconnection — Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1),* (July 10, 1986).

2. Several CDA converters are used in conjunction with the DECdecision software: front-end converters translate WK1, CALCGRD, ASCII, and DIF to the DTIF format. Back-end converters translate the DTIF format to ASCII and WK1. WK1 is the storage format for Lotus 1-2-3 Release 2.01 spreadsheets. CALCGRD is the storage format for VAX DECalc version 3.0a and VAX DECalc-PLUS version 1.0 spreadsheets. DIF is the storage format for Software Arts Visicalc spreadsheets. DIF is also supported by other applications.

3. R. Gumbel and M. Jack, "Development of the CDA Toolkit," *Digital Technical Journal*, vol. 2, no. 1 (Winter 1990, this issue): 38–48.

Richard T. Gumbel
Martin L. Jack

Development of the CDA Toolkit

Application program access to CDA documents is complex because of the many types of data these documents contain and their complex internal structures. The CDA Toolkit addresses the problem of access by providing a portable procedure library. The toolkit's primary feature is a procedural interface that enables applications to create, modify, read, and write compound documents. Designers of the toolkit's interface focused on the definition of the mapping between the stored document content and the document content in memory. The basic unit of interaction between the toolkit and the application is an in-memory data structure, termed an aggregate. Layered above the toolkit is a converter architecture that imports and exports documents to and from non-CDA formats. The converter makes available a variety of document sources and destinations to application programs.

The CDA Toolkit is a portable, efficient, extensible, procedure library. Using a single in-memory document format, the toolkit supports a wide range of document access and conversion services.

This paper describes the development of the toolkit's primary features, including the procedural interface, converter architecture, and toolkit internals. Then, it describes additional software within the operating system that allows existing applications to access CDA documents without reprogramming. The paper begins with an overview of the CDA characteristics that shaped the development of the toolkit software.

Compound Document Requirements and Toolkit Goals

The CDA environment provides users with a common way to store and exchange data, including line art, images, and text.[1] As such, the products in this environment have many of the same properties users expect of traditional text-based storage systems.

- The encoding mechanism is efficient.
- Applications can support CDA data formats as a matter of course for data storage and data interchange.

However, access to stored compound documents is much more complex than is access to traditional text documents. Compound documents

- Contain many types of data
- Have complex internal structures
- Must be computer architecture independent

To address this complexity, CDA designers built a limited number of standard document formats. This approach allows applications and support software to be developed once and then shared by all conforming applications. Examples of support software are document editors, conversion modules, and printing system interfaces.

In addition, CDA designers selected a standard encoding to be used throughout the environment, called the DDIS syntax (Digital data interchange syntax). The DDIS syntax conforms to Abstract Syntax Notation One (ASN.1), which is defined by the international standards ISO 8824 and ISO 8825.[2,3] For implementation economy, DDIS syntax excludes a few ASN.1-defined constructs. It also defines a set of additional constructs by using the extension mechanisms available in ASN.1.

The following data formats and syntax were developed as components of the CDA environment:

- DDIF document interchange format for text, graphics, images, and page layout
- DTIF table interchange format for spreadsheets, databases, and charts
- DOTS data object transport syntax to support the packing, mailing, and unpacking of multiple, related files

With these parts of the CDA environment outlined, several goals for the toolkit development were established.

- Provide a procedural interface for document access by application programs
- Support CDA data syntaxes
- Support all features of each data syntax
- Support encoding and decoding of files using the DDIS syntax
- Operate on VMS, ULTRIX, and possibly other operating systems
- Allow existing applications to coexist within the CDA environment

An aggressive schedule called for the release of the toolkit simultaneously with DECwindows version 1.0. This schedule was met. The project started in mid-June 1987; we completed a version for internal use by other developers that supported DDIF document access and conversion in late October; and we submitted the final version for DECwindows external field testing in early December.

The CDA Toolkit is a procedure library accessed by applications in order to create, modify, read, and write compound documents. The toolkit is used by numerous DECwindows applications that use CDA formats for file storage and for the exchange of data in interprocess communication. The CDA converter architecture, discussed in a later section, is layered upon the CDA Toolkit. The converter imports and exports documents to and from non-CDA formats.

The Procedural Interface

The toolkit's fundamental function is to present an application program with a procedural interface to read or create, edit, and write compound documents.

The decisions for the procedural interface design center around the definition of the mapping between the data elements in the stored document and the document's content as it is made accessible to a processing application in main memory.

The standard method for application access to documents is insufficient for accessing compound documents. Standard text files can be transferred as a sequence of character strings to or from the calling application program. Simple buffering and processing techniques can be used, because each element of the document has a simple structure. In comparison, a compound document is a complex data structure that is encoded into a byte stream for storage. More complex abstractions are needed that draw on dynamic storage allocation and list processing techniques.

In the following sections, we describe these abstractions, including a brief summary of ASN.1, the critical role of data structures in modeling composite data elements, mapping of primitive data elements, two modes for document transfer, and the design of the data path and toolkit options.

Document Data Structures: ASN.1 Fundamentals

ASN.1 specifies both a description language for data syntaxes and rules for encoding the data syntax into a byte stream in a way that is independent of specific computer storage formats.

An encoding consists of many elements, each having a type, length, and value. Primitive elements contain a single value of a data type specified by the data syntax. Constructed elements contain a number of nested primitive or other constructed elements as their value. The type field labels each element with a specified code that identifies the element in context. A specification that defines the meaning of each instance of an element must accompany the written data syntax.

ASN.1 defines several types of structured elements used in the CDA environment. The "sequence" structure contains a list of nested required or optional elements, each with a defined data type. The "sequence-of" structure contains a list of zero or more nested elements, all of the same defined data type. Lastly, the "choice" structure specifies the occurrence of one element selected from a defined list of possibilities. In each case, the nested elements can themselves be primitive or structured.

Document Data Structures: Aggregates

We designed the toolkit to provide a thin veneer over the data syntax and to deal only limitedly with document semantics. Another approach would have been to completely hide the data syntax. For example, the toolkit could operate on abstract document objects such as footnotes, rather than exposing the detailed DDIF structures that store the text content and layout information that make up the footnote. We took the simpler approach to minimize development time.

The basic unit of interaction between the application program and the toolkit is an in-memory data structure termed an aggregate. The purpose of each aggregate is to contain document data that corresponds to a particular area of a given ASN.1 data syntax, usually a "sequence" type. Thus, the

toolkit defines many aggregate types for each supported data syntax (DDIF, DTIF, and DOTS) that break the document data into manageable units.

The aggregate type that models the outermost structure of each data syntax is called the root aggregate. It does double duty in that it represents the document as a whole and the data path associated with the document encoding. It contains hidden toolkit context that is needed to manage the data path.

Each aggregate contains a defined list of items, from one to fifty or more, that are specific to the aggregate type. The purpose of each item is to provide a storage cell for a subsection of the data syntax that the aggregate models. Each item has a predefined data type that can be a scalar type (such as integer or floating point), a single-dimensioned array (such as integer array), or variable, where the value of a second integer item is used to select from several predefined data types for the primary item.

The variety of data syntax structures has a commensurate number of aggregate representations. Table 1 details typical cases. Since aggregates can contain items that reference subordinate aggregates or aggregate sequences, tree and list structures that represent the ASN.1 structural elements "sequence" and "sequence-of" can be formed. The ASN.1 "choice" structure can be modeled by defining two or more sets of mutually exclusive items within an aggregate — where each set models one element of the choice — and then by defining an integer item in the same aggregate whose value determines which choice has been selected. Ultimately, primitive elements of the ASN.1 data syntax map to primitive items within aggregates.

Table 1 Mapping of ASN.1 Elements to Aggregate Representations

ASN.1 Element	Aggregate Representation
Primitive	Scalar
Sequence	Aggregate
Sequence of primitives	Array
Sequence of sequence	Sequence of same aggregate type
Choice of primitives	Variant scalar
Choice of sequences	Variant item subrange
Sequence of choice of primitives	Variant array
Sequence of choice of sequences	Sequence of different aggregate types

A unique 32-bit handle identifies each instance of an aggregate. This handle is returned to the application upon creation of an aggregate and is used in subsequent operations on the aggregate.

Toolkit procedures allocate and deallocate aggregates, store and fetch the values of aggregate items, sequentially read aggregates from and write aggregates to documents, and create and traverse aggregate sequences. The toolkit provides public interface files that define numeric equivalents for aggregate type codes, aggregate item codes, and aggregate value enumerations, together with documentation of the data type of each item.

During decoding of an input document, the toolkit creates aggregates, stores document data into the items, and returns the handle of each aggregate to the application. The application examines the document data and later destroys the aggregate. On output, the application creates and populates aggregates, and passes them to the toolkit for encoding.

To allow the document data to be broken into small units for transfer to and from applications, we identified a fixed subset of the aggregate types for each data syntax that models the outer elements of the data syntax. Known as top-level content, this subset passes to and from the application during encoding and decoding. As we have seen, however, a complex tree of substructures may be present in subordinate aggregates linked to top-level content.

Representation of Primitive Data Elements

As we have noted, ASN.1 primitive elements are encoded in an architecturally neutral manner. To allow applications to manipulate primitive elements directly using native programming languages, the toolkit converts ASN.1 encodings to and from storage representations supported by the target computer hardware. In many cases, the mapping varies depending upon the intended application of the value. For example, the toolkit maps ASN.1 Boolean values to single bits and transfers ASN.1 octet string elements unmodified.

Where the values are suitably constrained, ASN.1 integer values are represented as signed 32-bit integers. However, in the numeric applications within the DTIF format, they are represented as integers of arbitrary length so that no loss of significance can occur. Similarly, DDIS-defined floating point elements are converted to single-precision floating point in the DDIF format, where precision is not an issue. However, the same type is maintained

internally as a 128-bit value in the numerically critical areas of the DTIF format. As a service to the application, the toolkit can convert these general floating-point values to any hardware-defined precision or transfer the internal 128-bit value.

Incremental Mode and Document Mode

The toolkit provides the application designer with two modes of document transfer: incremental and document. The incremental mode is somewhat more difficult to program, but consumes less dynamic memory. The document mode provides an easy-to-use interface, but requires that the entire document reside in memory.

In incremental mode, one aggregate of content per toolkit call flows to or from the application. Aggregates processed in this way represent the highest level of document content. For example, a segment header or an element of text content might be transferred to or from a DDIF document. In the case of the segment header, aggregates that specify segment attributes, such as font selection and type definitions, are attached to the segment aggregate as substructure.

The document mode is internally layered on the incremental mode. In document mode, the toolkit reads or writes the entire document to or from the in-memory structure. The root aggregate forms the root node of the document tree.

In incremental output mode, the application must invoke scope control procedures at known points to encode ASN.1 constructor elements surrounding defined portions of the document content. In document mode, the toolkit produces these elements without additional programming in the application.

Input and Output Data Paths

The toolkit supports a rich variety of data sources and destinations. We wanted to enable the compound document formats to pervade data interchange within the system in the same way that ASCII historically has. We therefore took care to design input and output data paths that were complete, general, system independent, and easy to use.

We defined a data path, which we call a stream, that offers the application complete control over data buffering. On input, an application-specified callback procedure returns the address and length of each new input buffer. On output, the application specifies the address and length of the initial output buffer when the stream is created. An application-specified callback procedure disposes of each full output buffer, and specifies the address and length of the next successive output buffer. When the stream is closed, the toolkit invokes the callback procedure to dispose of the last partially filled buffer.

File support is layered on stream support. For ease of use, a single procedure provides system-specific assistance in opening or creating a file with the proper file system attributes, buffer allocation, and toolkit-supplied input-output services.

As we noted earlier, the source or destination of a document file can be an entire hierarchy of aggregates linked from the document root aggregate.

Multiple Data Syntax Support

When the DOTS data syntax was defined and implemented, the toolkit design was extended to support multiple data syntaxes and document nesting. Because the data syntax definition itself was only a few lines, the implementation effort was relatively simple. The implementation was focused on syntax and nesting, rather than on the work of mapping the ASN.1 data syntax to aggregates.

The starting point for document nesting is the ASN.1 "external" data type. It is a constructor that specifies the "object identifier" and other descriptions of a nested data syntax. The receiver is thus able to determine the data syntax in advance and prepare to process the data. The data follows in one of three representations: an ASN.1 data syntax, a string of bytes, or a string of bits. Meaningful communication, as always, depends upon agreement between the sending and receiving application on the interpretation of the data.

In the applications of this feature envisioned for the initial release, the nested documents were CDA data syntaxes that could be interpreted by the toolkit. Thus, the items in the "external" aggregate, which specify the representation choice and its value, identify the data syntax as ASN.1. They also reference the root aggregate of a nested compound document.

However, it was critical to lay adequate groundwork for the full generality of the ASN.1 "external" type. Future uses were unquantified. We could not assume that a nested ASN.1 data syntax would be implemented in the toolkit. Moreover, we could not assume that the byte and bit string data types would be completely available in memory at the time the "external" element was being encoded or decoded. At the same time, we needed to provide an easy-to-use access path for CDA formats.

The key to the design was a mechanism to read or write the ASN.1 encoding to the point where the nested encoding actually begins and then return control to the application. The application must then read or write the nested encoding, and call the toolkit to finish processing the "external" type.

On output, the application has two options: (1) to store the nested encoding in the "external" aggregate before writing it, or (2) to use incremental or document mode output on the same stream to write the nested document. On input, the toolkit provides a procedure to read and store the encoding completely.

In the future, the toolkit ASN.1 encoding and decoding services layer could be exposed and new functions added. An application would be able to incrementally read or write a string encoding, or to process nested ASN.1 data syntaxes not implemented by the toolkit.

Limited Semantic Processing

We wanted to simplify the development of applications that access compound documents, such as document viewers or page description language converters. Thus, we provided a general mechanism to specify semantic processing of input documents. We also added a few toolkit options to enable very limited processing of common document semantics. In this mode, the toolkit can resolve references made to content in external documents, to type definitions, or to generic content; and it can apply attribute inheritance rules.

Converter Architecture

As noted earlier, the CDA converter architecture comprises a layer on the toolkit that supports non-CDA formats. The goals for the converter architecture project were as follows:

- Provide a procedural interface for data format conversion
- Support many document formats
- Incorporate non-Digital conversion modules
- Select conversion modules dynamically
- Build on existing CDA Toolkit services

The converter architecture is based on a conversion hub model. A front-end module converts an input document to a hub format. A back-end module then converts the hub format to an output format. The hub format can be in either DDIF or DTIF format.

The hub model has the significant advantage that the number of conversion modules increases relative to the sum, rather than the product, of the number of input and output document formats supported. However, the model can be used only when the hub format can fully express the semantics of input documents; otherwise, loss occurs on conversion to the hub format. The CDA formats were designed to avoid this problem.

The in-memory aggregate structures defined for document transfer to and from encodings are the same structures used for document transfer through the conversion modules. Thus, the converter architecture is an extension of the toolkit procedural interface.

As in the case of the CDA encoding and decoding interface, a variety of document sources and destinations are available. Files and streams must be supported by conversion modules. In-memory aggregate trees are supported when the source or destination format is a CDA format.

The following sections present more details about the document conversion procedure within the model and the translation that occurs from one hub format to another, also called domain crossing. Front- and back-end interfaces are also discussed further.

Conversion Control Procedure

A single control procedure that is packaged with the toolkit initiates a document conversion. The application provides

- The name of the source and destination document format
- Identification of the data source and destination
- Parameters that are ultimately passed through to the specific conversion modules to control their operation

The control procedure prepares for conversion by determining the location of the requested front and back end. Three means of determining the location are possible. First, the toolkit contains a CDA front end and CDA back end for use with DDIF or DTIF source or destination formats. The toolkit's function is only to encode or decode the aggregate stream. Second, the application can specify the address of a procedure to act as the front end or back end. Third, the control procedure uses the specified format name to locate and activate an external program image that contains the front end or back end.

Thus, as illustrated in Figure 1, the converter control procedure assembles the complete conversion program at execution time based on the conversion modules installed and available at the time they are referenced. Figure 2 shows document data flow through the conversion.

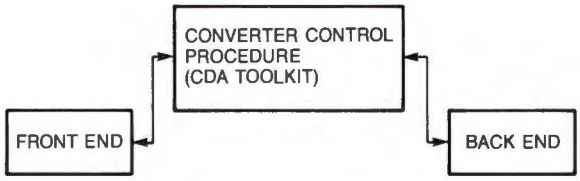


Figure 1 Complete Document Conversion Program

Each operating system includes a small number of essential conversion modules as standard equipment. Because the procedural interface is published and the binding of conversion modules is dynamic, Digital and other vendors can easily make available an unlimited range of optional conversion modules that increase the value of their products. Many application developers who use CDA conversion services have added logic that scans for all available conversion modules and displays a conversion format menu dynamically tailored to the actual environment.

Domain Crossing

As we have seen, the toolkit supports both DDIF and DTIF conversion hub formats. Stated differently, a front-end or back-end module operates in either the DDIF domain or DTIF domain, depending upon which hub format the module produces or consumes.

In many cases, the front end and back end operate in the same domain. For example, in a word processing document conversion, the front and back ends both operate in the DDIF domain. In a spreadsheet conversion, both operate in the DTIF domain. Clearly, however, a user might wish to print a spreadsheet or incorporate it into a memorandum. In this case, the front-end module produces DTIF format, and the back-end module expects DDIF format.

We added additional logic to the converter control procedure to permit such domain-crossing conversions. We also developed a DTIF-to-DDIF conversion module. A domain conversion module receives aggregates from one domain and translates them to aggregates of another domain. A domain conversion module does not process files; its inputs and outputs are in-memory aggregates.

If the converter control procedure determines that the front and back ends operate in different domains, it attempts to locate a domain conversion module that converts the input domain to the output domain.

If one is available, the control procedure alters the conversion data flow such that aggregates flow from the front end into the domain conversion module, and from the domain conversion module into the back end. The invocation of a domain conversion is transparent to the front end, the back end, and the application requesting the conversion service. Again, the control procedure makes a dynamic search with a stylized name so that the set of available domain conversion modules is extensible.

The DTIF-to-DDIF domain conversion module is thus the single point that performs report writing and formatting operations for tabular data. Thus, as DTIF-compliant applications rely on the converter architecture for the interchange of tables with other formats, so they can rely on this module for printing or document viewing requirements. Each application need not contain this logic.

Front-end Procedural Interface

The primary function of a front end is to read the input document format that it supports, translate the document semantics to the hub format, and return content aggregates one by one upon demand. Therefore, a front end must present a defined interface to the converter control procedure. Front ends must define four procedures: initialize, get-aggregate, terminate, and get position.

The initialize procedure initializes the front end for the conversion. Because the procedure has a

Figure 2 Data Flow in Conversion

known name, it can be located by the control procedure. Typically, this procedure opens an input file, allocates and initializes a context block, and processes options passed to it by the control procedure. The context block is used to maintain current state information about the conversion. The initialize procedure must also return the addresses of the get-aggregate, terminate, and position procedures to the control procedure.

The get-aggregate procedure performs most of the conversion. It reads from the input file and produces DDIF or DTIF aggregates. This procedure does not convert the entire input document in a single call; rather, it reads from the input document until it is able to produce the next sequential top-level content aggregate. This aggregate is then returned to the control procedure. Subsequent calls to this procedure continue to build top-level aggregates until the end of document is reached.

The control procedure calls the terminate procedure when an end-of-document is returned from the get-aggregate procedure or when an error has occurred. The terminate procedure typically closes the input file and deallocates its context block.

Finally, the get-position procedure is used by applications that must report the progress of a conversion. For example, a document viewer can use this information to position a scroll bar within the document window.

Back-end Procedural Interface

The function of a back end is to receive content aggregates from a front end, translate the hub domain document semantics to the output document format that it supports, and write the output document. A back end must also present a defined interface to the converter control procedure. Back ends must define a single procedure analogous to the initialize procedure of a front end. This procedure can be located by the control procedure because it has a known name. Back ends run to completion rather than operate on one aggregate at a time as front ends must.

The back end calls a toolkit procedure to return the next sequential content aggregate to the control procedure, just as if it were reading from a file. The control procedure, in turn, calls the front-end procedure to return the aggregate. The back end then translates these aggregates to its output format. This process is repeated until the end of the input document is reached. The back end closes its output file, deallocates other resources, and returns control to the control procedure.

Conversion Module Packaging for VMS and ULTRIX Systems

Conversion modules are built and packaged differently for each operating system, based on the features of that system. On the VMS system, conversion modules are shareable images that are dynamically activated. Because the application, the front-end module, the domain conversion module (if used), and the back-end module share a single address space, aggregate data passes directly through these components. On ULTRIX platforms, conversion modules execute in child processes. The control procedure creates pipes to transfer aggregate data from the front to the back end. However, the toolkit masks these differences, and the program code can be identical on the two operating systems. Only the linking procedure varies.

Conversion modules are located in a known directory. Each has a stylized file name that allows the control procedure to locate the module for a desired format. For example, the name of a front end on a VMS system is domain$READ_format and the name of a back end is domain$WRITE_format. The domain variable is the DDIF or DTIF format, and the format name is specified by the user. The file name of a domain conversion module is CDA$in-domain_TO_out-domain, for example, CDA$DTIF_TO_DDIF.EXE. In all three cases, the initialization procedure has the same name as the file.

Document Viewers in the Converter Architecture

This section illustrates how an application can exploit the power of the CDA converter architecture to provide seamless access to documents from a variety of sources.

Two compound document viewer applications are packaged with the VMS and ULTRIX operating systems. One is designed to display a faithful rendition of the document in a DECwindows display; the other does a conversion for character-cell output within the stricter limits of that display technology.

The viewers are callable and can in turn be used by a higher level application. For example, the DECwindows Mail, videotex (VTX), and electronic conferencing (NOTES) applications provide CDA document viewing capability.

To access the input document, the viewers call the converter control procedure, specifying that the viewer is the back-end conversion module. All of the usual front-end selection and domain evaluation mechanisms come into play. Thus, any document format that can be converted to CDA

aggregates can be viewed. A DDIF document produced by the DDIF document editor is a simple case, flowing through the internal CDA front end directly to the viewer. A non-Digital spreadsheet format for which the vendor supplied a conversion module flows through the non-Digital front-end module, and then through the DTIF-to-DDIF domain conversion module to the viewer. The end user may be completely unaware that different internal processing has occurred.

We had to provide just two additional mechanisms for the document viewer. The first is support throughout the architecture for document position sensing, which is needed to draw document scroll bars correctly. The second is the ability to suspend and then transparently resume a conversion operation, an ability which users need during document viewing.

Toolkit Internals

Thus far, we have described the architecture and design of the procedural interface to the toolkit, as visible from an application. We now examine aspects of the implementation of that interface that are not visible from outside the toolkit.

Several factors made it clear to us that elegance of expression was essential in implementing the basic toolkit.

- The size of the document data syntaxes (several hundreds of lines each)
- The need to have the toolkit support the entirety of each data syntax
- The changeable state of the data syntaxes while still under active development

We had to design mechanisms that were economical to implement, easy to modify, and easy to verify. Thus, the low-level design for the toolkit focused on encoding as much information as possible in tables and then writing simple interpreters for the tables, rather than writing code to deal with each construction.

Because so much of the toolkit operation depends upon access to tables, we needed to design efficient ways to check the validity of an operation and access the relevant data. The solution was to partition the integer codes that identify aggregate types and items into subfields, as shown in Figures 3 and 4. Simple extraction of bit fields then yields the data syntax, aggregate index, and item index values.

Figure 3 Bit-encoding of Aggregate Code

Figure 4 Bit-encoding of Item Code

Each of the four major subsections of the toolkit follows the table-driven interpretive model. We will examine aggregate and item access, document encoding, and document decoding in turn.

Aggregate and Item Access

The procedures that create and destroy aggregates, and those that store and fetch aggregate items, operate from two sets of tables. Aggregate description tables encode all aggregate-specific information: the length of the aggregate storage and the number of defined items. Item description tables encode item-specific information: the item data type, the item input semantic processing algorithm, and the offset to the item's storage area.

Encoding and Decoding Interpreter

The fundamental function of the encoding and decoding interpreters is to translate the standard in-memory representation to and from the ASN.1 encoding. The interpreters are each implemented in two layers. The lower layer is oriented toward ASN.1, and the upper layer is oriented toward application data syntax.

As described below, the ASN.1 encoding and decoding layers share many lower level utility services and design characteristics. The upper encoding and decoding layers are quite different in concept, as will be seen in subsequent sections.

ASN.1 Encoding and Decoding Layer

The ASN.1 layer has four main functions. First, it performs translations between the interchange-oriented ASN.1 type, length, and value encodings and the hardware-oriented representations present in memory. For example, this layer on a VAX processor must reverse the bits within each byte of a bit

string to translate between the natural VAX bit and the ASN.1-specified ordering. Second, it checks the syntax of the encoding according to parse tables for the CDA data syntaxes contained within the toolkit. These tables are produced by a parser generator that processes ASN.1 data syntax descriptions. Third, the layer checks the encoding for conformance to ASN.1 encoding rules. Fourth, it returns the parse table entry number on input, so that the input decoding interpreter can access the parallel table that controls its processing.

Document Encoding Layer

The document encoding layer is an interpreter that executes a per-aggregate program encoded in the toolkit's static tables. The interpreter has only a handful of operators, which are about evenly divided between performing data output and control operations. We reduced the number of cases the interpreter had to handle by choosing consistent representational mappings between ASN.1 elements and in-memory aggregate items. Table 2 describes the operations that the encoding interpreter supports.

The interpreter must support a nonsequential execution model because of the existence of the ASN.1 optional and choice constructions, which denote portions of the encoding that may or may not be present in a particular document. To support this model, we included operators to test aggregate item presence, and an operator to test the contents of an aggregate item for a given value.

CDA data syntaxes often employ a given ASN.1-constructed type in a variety of contexts that generate different encodings. Consequently, we had to model this usage in the encoding layer by passing context information to a recursive call of the encoding interpreter. The layer includes conditional flow operators to test this context information.

Table 2 Output Machine Operations

Operation	Description
HALT	Terminate processing of the current aggregate and advance to the next aggregate in sequence; if none, return.
TAG(t)	Write the ASN.1 type *t* with a zero or indefinite length; the element has no value.
EOC	Write an ASN.1 end of constructor.
VAL(t,i,f,c)	If the item *i* is present, write its value with the ASN.1 type *t*. The value *f* selects from one or more available standard representations. The value *c* specifies a data type specific constant. For example, in an array-of-string type, it specifies the ASN.1 type of the primitive octet-string elements; in an aggregate-valued type, it specifies the context value for the recursive output machine call.
CON(t,c)	Write the ASN.1 type *t* and value c; the element is of type "integer."
EXT(i)	Write the introducer for the "external" encoding corresponding to item *i*.
BR(b)	Skip *b* instructions.
IFNONE(b,ia,iz)	If every item in the range *ia* through *iz* is absent, skip *b* instructions.
CTXL_BNE(b,c)	If the aggregate context is not equal to *c*, skip *b* instructions.
CTXL_BEQ(b,c)	If the aggregate context is equal to *c*, skip *b* instructions.
CMPL_BNE(b,i,c)	If item *i* is absent or its value is not equal to *c*, skip *b* instructions.

Document Decoding Layer

The document decoding layer also consists of an interpreter that executes a static table within the toolkit. First, the ASN.1 level returns an element and its parse table entry number. The CDA level then indexes the parallel "get-descriptor" entry to obtain processing instructions for the element. Because the single entry must embody the complete element semantics, the table entry is quite large; it contains 112 bits divided into six fields, as shown in Figure 5. The table entry resembles a typical control word for a hardware microengine.

Table 3 describes the operations that the decoding interpreter supports, with reference to the field names shown in Figure 5. They are executed in the order shown, so that an operation that requires several steps can be predictably formulated. Table 3 references three registers that can be set and used by microoperations: VIC, the value item code; CIC, the constant item code; and AGG, the current aggregate pointer.

Because ASN.1 structures can be nested, the interpreter maintains an aggregate stack to create the corresponding tree and list structures.

Testing and Performance Methodology

A thorough verification of the toolkit required the development of specialized testing machinery. The amount of program code in the toolkit is not large, and thus program flows were relatively easy to test. However, the definition tables that make up the item access, input, and output machines are large; and it was critical for the product's success that every detail be correct.

We first developed a back-end conversion module that could produce a low-level textual dump of any CDA document. This module, the analysis back end, is a part of the released product. We also developed a front-end converter that could accept directly the output of the back-end converter. We could also create input files with a text editor.

Thus, we were able to write input scripts, pass them through the converter to produce an encoded file, and pass the resulting file through the converter to produce a file dump. We could then repeat the process with the file dump as input to complete a "round trip." Comparisons of the encoded files and the file dumps with the original input scripts then revealed errors.

We instrumented the input machine to detect elements of the parse table that were not reached in a group of test runs. Using this information, we added elements to the input scripts to achieve exhaustive coverage of each data syntax.

Finally, we used the Performance and Coverage Analyzer product to ensure that all code paths were exercised, and to make local coding changes to improve performance.

Toolkit Portability

Today, the toolkit operates on three platforms: VAX with VMS, VAX with ULTRIX, and RISC with ULTRIX systems. This wide range of support across multiple hardware architectures and multiple operating systems was a significant goal for the initial software design project.

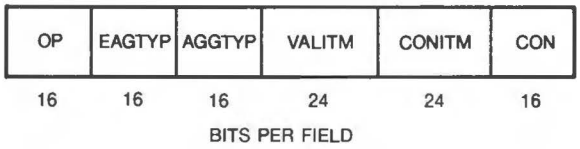


Figure 5 Bit-encoding of Get-description Entry

Table 3 Input Machine Operations

Operation	Description
SPC(conitm)	Special case: dispatch to code identified by *conitm.*
EAG(eagtyp)	Create early aggregate of type *eagtyp,* set AGG.
SCI(conitm)	Set constant item: CIC receives *conitm.*
SVI(valitm)	Set value item: VIC receives *valitm.*
AVI	Add value item: VIC receives VIC plus the saved VIC of the aggregate stack entry. This operation is used when parse tables merge to get relative item addressing.
STC(con)	Store constant: Item CIC in AGG receives *con.*
STD(con)	Store discriminant: Item VIC-1 in AGG receives *con.*
PSH	Push aggregate stack, save VIC and AGG, enable POP.
AGG(aggtyp)	Create aggregate of type *aggtyp,* set AGG.
STV	Store value: Item VIC in AGG receives data from ASN.1 encoding.
HLD	Hold: Disable POP operation.
REL	Release: Enable POP operation.
POP	If stack is empty, return AGG as top-level content. If enabled, restore VIC and AGG, pop aggregate stack.

A large part of the toolkit is system independent. With careful type casting, the code compiles and runs on all platforms. There is, of course, some operating-system-specific code. We have isolated most of this code to a few modules; of the 70 modules that make up the toolkit, only three are system-specific. In a few cases, we have used conditional code to deal with architectural and system dependencies.

The two biggest problems we encountered in porting to new platforms were word length dependency and bit-within-byte ordering. We have carefully used variable declarations and casting to circumvent the word size problems. The bit ordering problem affects a small amount of code that has been assigned conditions to support either ordering. A compile-time switch selects the appropriate ordering.

Related Operating System Support

The success of the CDA program has been due in part to its seamless integration with the VMS and ULTRIX operating systems. Components of both systems have been modified to effect integration with CDA files and utilities.

Data Syntax Recognition

On the VMS system, a new file attribute, termed stored semantics, was added to the RMS software (a record management system). The attribute identifies the data syntax of a file and is the same as the ASN.1 Object Identifier assigned to the data syntax. The directory utility was enhanced to display the stored semantics value as a string. Further, the SET FILE command now sets or clears the stored semantics value.

On an ULTRIX system, the FILE command was enhanced to recognize the initial ASN.1 type field assigned to CDA files.

To copy CDA files across systems in a network, a new VMS utility, exchange/network, has been added. With this utility, semantically tagged files can be copied to non-VMS operating systems, and non-VMS system files can be copied into VMS-tagged files.

Traditional Application Access to CDA Files

The customer's investment in existing software must be protected, even as new compound document applications are being developed. We therefore sought to allow compilers to process source programs written with the compound document editor without forcing the modification of the many available compilers. Many of these compilers are supplied by other companies.

The answer was to use the stored-semantics tag of the VMS system to transparently invoke a conversion. During the file-open sequence, the system makes the file's semantic tag available to the application. The application can then declare a desired semantic tag. If these are different, the system attempts to locate and connect a translator, referred to as an RMS extension. A program without the logic necessary to employ the new mechanism invokes a translator that makes the file appear to be an ordinary text file.

As part of the toolkit project, we developed an RMS extension for the DDIF data format. The extension performs syntax analysis of the DDIF format that requires only a small finite-state machine, rather than a full parse. The analysis returns only the text content of the DDIF file.

The output of the RMS extension is designed to be identical to that of the DDIF text back-end converter. On an ULTRIX system, a user achieves the same effect by constructing a pipeline of the text converter and the desired utility.

Conclusion

The CDA Toolkit provides a set of data access and conversion services used by numerous applications. A consistent, general, layered approach to document format conversion has been effective in achieving a high level of application interoperation.

Acknowledgments

The authors wish to acknowledge the contributions of the CDA Toolkit development team: Linda Busdiecker, Regina Collins, Pat Justus, John Middleton, Leanne Olson, Lauren Sacco, Roy Stone, Jeff Tancill, and Tony Vlatas. Bill Laurune developed the ASN.1 parser generator. Gary Allison and Stu Davidson provided VMS Engineering support.

We would also like to thank Mark Bramhall and Bob Travis for their invaluable technical guidance.

References

1. R. Travis, "CDA Overview," *Digital Technical Journal*, vol. 2, no. 1 (Winter 1990, this issue): 8–15.
2. *Information Processing Systems, Open Systems Interconnection, Specification of Abstract Syntax Notation One (ASN.1)* (International Standards Organization, reference no. ISO 8824:1987(E), May 1987).
3. *Information Processing Systems, Open Systems Interconnection, Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)* (International Standards Organization, reference no. ISO 8825:1987(E), May 1987).

Baldwin K. Cheung
Neal F. Jacobson

Interapplication Access and Integration

Applications within the CDA architecture can share and interchange data through the DECwrite and DECdecision LiveLink connection. Applications developers can build more tightly integrated levels of applications with the AIL library, while DECdecision's Builder allows application integration at the user-interaction level. AIL is a platform-independent subroutine library that provides application invocation, data exchange and flow control services for interacting applications. Builder can be used as either a conforming LiveLink application or stand on its own. Together, these tools form an interapplication architecture that permits easy application access and integration.

A primary goal for the DECwrite and DECdecision software developers was to design a family of products that would work closely together. Specifically, we wanted to be able to call an application, use its application features, and incorporate the result in an automated fashion. To achieve these goals, we needed to be able to do application invocation and interapplication communication.

This paper describes two services that address interapplication integration. The first is the AIL application interface library that allows application developers to build sets of applications with tighter levels of integration. The second is the DECdecision Builder tool (herein referred to as Builder).[1] Builder allows applications to be integrated at the user-interaction level.

We begin with an overview of the LiveLink function that AIL supports.

Data Access through LiveLink Connections

The DECwrite and DECdecision LiveLink functions automate the processing of external information and free the user from these manual application transfer chores.

Simply put, LiveLink software is a connection to external data that is stored independently from a document. This external data can be stored locally or distributed over a network. LiveLink software provides the mechanism to streamline accessing and managing this information. The LiveLink connection can also be used to share information in a work group in an orderly and managed way.

There are two types of connections: data link and application link. A data link is a direct link to some external data. The data, in this case, is in final form format (e.g., an illustration) that is usable directly by a document processing application. An application link is a link that involves an external application, referred to as a LiveLink application. The application link is responsible for performing appropriate manipulations on the external data, which is normally revisable form data. It then converts this data to final form result for inclusion into a document.

Conceptually, the data used in a LiveLink connection is point to information. This approach is unlike the DECwrite menu feature include, where data (text and/or graphics) is included and becomes a permanent part of the document.

One advantage of the LiveLink connection is that it uses the latest information without requiring a manual user update. By definition, data links are always up-to-date because they are direct links to the current final form data.

There are some complications with application links because the referenced data is revisable data for another application. To solve this problem, we implemented an automatic update feature in the DECwrite editor, to maintain the latest information characteristic offered by LiveLink connection. This feature tracks the revision time of the external data (the revisable data, in this case), and, upon user consent, updates the data when the source data has been revised. The update process is automatic and requires no user intervention. The LiveLink application to which the data belongs regenerates the final form result for use in the document.

The LiveLink connection in the DECwrite editor uses a special object called a data block as a container for LiveLink information.[2] To the user, a data block is a rectangular region on a page which is used to display information (such as a picture, table, or flow diagram) from an external source. It can be manipulated just like other document objects.

AIL Library

As noted earlier, the purpose of AIL is to support the implementation of the LiveLink function. AIL is a subroutine library that contains a standard set of functions. It defines a set of rules for application interactions and provides a common interface for integrating applications.

Following are some basic design considerations:

- Subroutine interface. The AIL library provides a simple-to-use interface. Although a message-based service might be more flexible and extensible, it requires more programming effort. A subroutine interface tends to be more convenient to the programmer and easier to use.
- Operating system independence. The AIL interface is designed to be operating system-independent in order to be more portable. It hides all the operating system-specific features, such as low-level communication services, within its implementation.
- Reusable application. The startup time for some applications can be quite time-consuming. The AIL library has an operation model which allows an application to be reused to perform multiple application sessions. In between sessions, an application remains in the system in a clean state and waits for reuse. A reuse session can typically come up in one-third the time of image startup.
- Single library for parent and child. Both the AIL library parent and child functions are combined into one library, which makes it more convenient for those applications that want to support both parent and child functions.
- Single application image. For maintenance and packaging issues, it is desirable to minimize the number of image versions released for a particular application. An application built with the AIL library can be run as a stand-alone image and as a child application.
- Asynchronous environment. Target applications are supposed to operate under an asynchronous or event-driven environment, such as the DECwindows environment. In an event-driven environment, an application is required to service multiple activities at the same time. In particular, it is not acceptable for an application or a service to engage in a time-consuming task. To suit this environment, the AIL library is designed with nonblocking calls.

Application Interaction Model

We adopted a parent and child model for application interaction under the AIL library. Because of this model's simplicity and conceptual cleanliness, we could accomplish what we wanted to do for the current functionalities in the DECwrite and DECdecision products.

As shown in Figure 1, a parent application can have one or more child applications. A child application can itself be a parent application to other child applications. Generally, a user starts with one application. From this application, the user can invoke another application to perform functions that are not available in the first application. Depending on the nature of the first application, the invocation of the second application can be initiated either explicitly by the user or transparently by the first application. Expanding the hierarchy further, the second application can, in turn, invoke some other applications.

In theory, the invocations follow a simple tree structure that can be extended to many layers. However, in practice, the number of application layers will generally not get higher than two.

Data Exchange between Applications

The ability to exchange data is an important part of application integration. Depending on the nature and capability of participating applications, the

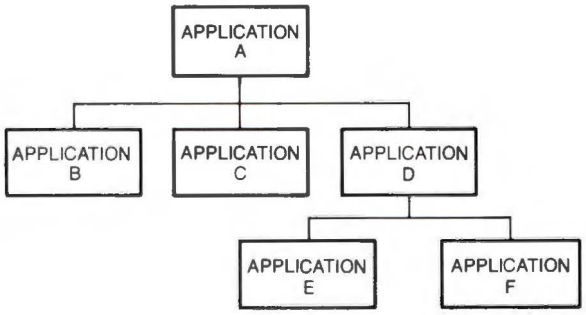


Figure 1 Parent and Child Application Interaction Model

requirements for data exchange can range from a simple input/output black box level to an ongoing dynamic data dialog level.

Input/output Black Box Model We believe that the classic input/output black box model can satisfy a large class of data exchange situations between applications. In this model, a child application is treated as a black box in which input data is supplied at the beginning of a session, and output data is produced at the end of the session. The input data to the child application is the child application's own input data, normally in revisable form or in a command script.

The output data consists of final form data and possibly updated revisable data. The parent application would be interested in the end result only (i.e., the final form data). An advantage of this model is that all child applications interact the same way with a parent application, and the parent application does not need special programming to deal differently with each child application. It also enables conforming applications, present and future, to be easily adapted to work with one another.

Dynamic Data Dialog Model Although the input/output black box model is simple and universal, it has limitations. Applications that wish to work closely with each other need more flexible and more efficient ways to exchange data. Just as the more two people know about each other, the more topics of interest they find to discuss, the more details applications know about each other, the more specific they can request and provide needs to each other. The dynamic dialog model supports applications that communicate with each other with frequent dialog and numerous topics identified as items.

AIL Routines and Operations

The AIL library is designed as a procedural interface. Conceptually, applications are connected and interfaced to each other only through the AIL layer.

Figure 2 shows the logical layers within a typical application that incorporates AIL. AIL is either an image that can be shared or an object library that is linked into application programs. To work with AIL, an application must supply a set of action and callback routines.

AIL Routine Types

There are three types of AIL library routines: function, action, and callback.

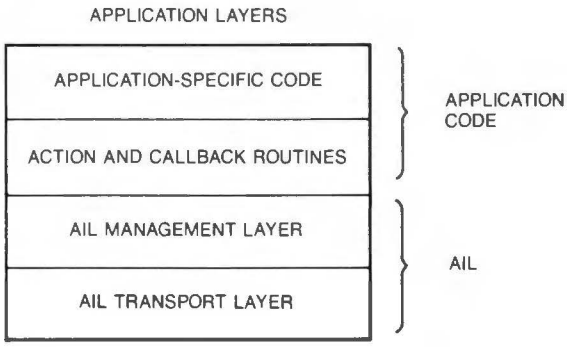


Figure 2 Logical Application Program Layers with AIL

Function routines are the AIL functions supplied in the AIL shareable image or object library. An application calls a function routine to perform or initiate a particular function. Some of these routines only interact with AIL to set up internal states or to ask for information. Others cause action routines in other applications to be invoked.

Action routines are supplied by an application. They are used to trap functional requests that come from other applications. An action routine is called as a result of an AIL function being called in another application. An application is responsible to provide and declare appropriate action routines to work with the AIL function.

Callback routines are also supplied by an application. They enable an application to operate smoothly in a single-threaded, event-driven environment. All AIL functions that are potentially time-consuming accept callback routines as part of their subroutine calling parameters. For these AIL functions, AIL returns control to the application without waiting for the request to complete. AIL notifies the application through the appropriate callback routine when a request is completed.

Figure 3 illustrates a typical calling cycle involving a function routine, an action routine, and a callback routine. Applications A and B are separate applications interacting with each other through the AIL library. The following steps occur in the calling cycle.

1. Application A issues a function routine call to the AIL at the Application A side to request Application B to perform a task.

2. If there is no obvious calling error, Application A's AIL transfers the request to Application B's AIL and gives control back to Application A.

3. The Application B's AIL calls the appropriate action routine in Application B, which services the request and acknowledges its completion by returning control to its AIL.

4. Application B's AIL then hands over the acknowledgment to Application A's AIL, which calls the callback routine in Application A to report the completion status.

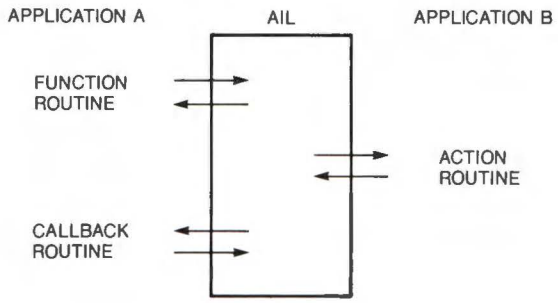


Figure 3 Interaction Cycle Involving Various Types of AIL Routines

Basic AIL Usage Sequence

Figure 4 depicts the basic calling sequence between a parent application and a child application in the DECwindows environment. Note that this figure includes only the routine calls related to the basic control flow. Other calls, such as those for data exchange, are excluded for the sake of simplicity.

Also, only the arrows showing the directions of the function calls are shown. The arrows for routine returns and callback routine calls are omitted.

The following steps outline the basic sequence in which a parent application executes a child application.

1. The parent application is started. It first calls the AiStartup routine to initialize the AIL environment and then the AiSetActions routine. The latter sets up a set of action routines to handle interactions from child applications.

2. When a parent application is ready to access a child application, it calls the AiInvokeAppl routine to load and invoke the child application. The AiInvokeAppl routine uses an appropriate system mechanism and brings up the specified child application.

3. If the child application is started successfully, the parent application can begin an application session. It calls the AiInitializeSession routine to tell the child application to prepare for a new application session. At that point, the parent application also passes input data to the child application and sets up execution attributes.

4. To start the child application session, the parent application calls the AiExecuteSession routine. The child application then brings up its user interface and starts processing input from the user.

5. An application session can be terminated by either the child or the parent application. If a child application initiates the termination, the parent application receives a call to its AppSessionExit action routine. If a parent application wants to terminate a child application, it can call the AiTerminateSession routine. As part of a normal session termination, the parent application expects the child application to call the appropriate parent action routines to save all its existing work and pass the presentation result back. At this point, a conforming child application should hide its user interface, free all unnecessary resources, and wait for the parent to start another application session. This enables a faster restart when the child application is reused.

6. When the parent application no longer needs the child application, the parent application calls the AiTerminateAppl routine to disconnect the child application.

The following steps outline the basic sequence in which a child application is executed when it is called by a parent application.

1. The child application is started and calls the AiStartup routine to initialize the AIL environment. As a return status, AIL informs the processing application whether it is invoked as a child application.

2. If an application is invoked as a child application, it calls the AiSetActions routine to set up action routines to handle interactions from the parent application.

3. The child application enters an event processing loop to wait for instructions from the parent application. In the event processing loop, the instructions or interactions from the parent application come through as calls to the action routines that the child application has established.

The parent instructions include starting and stopping application sessions, setting attributes, exchanging data and shutting down the application.

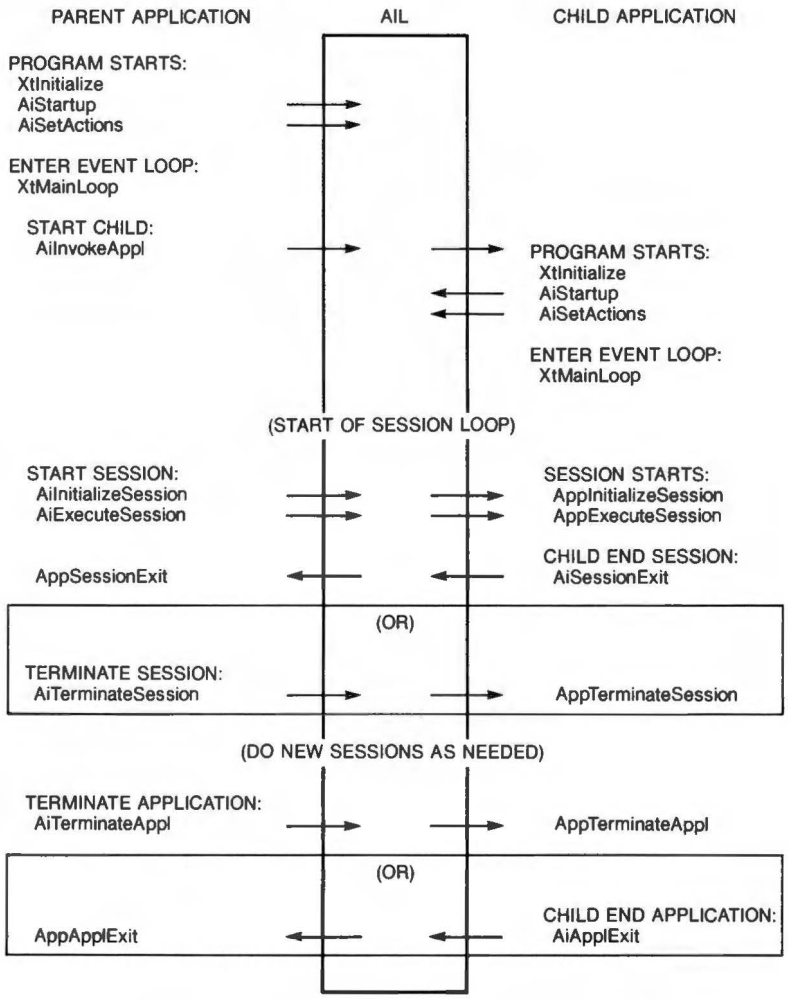


Figure 4 Basic AIL Routine Usage Sequence

As responses to user actions or as consequences to other events (including parent instructions), the child application can also call AIL functions to initiate various functions such as exiting the current application session or delivering data to the parent application.

Builder

Builder is a variation on the application link theme. Its revisable data is a script, or blueprint, that allows applications to be integrated at the user-interaction level. In addition to being a conforming LiveLink application, it can also stand by itself. In fact, Builder is also a component of the DECdecision family of applications.

The remainder of this paper presents an overview of Builder and its design and infrastructure.

Builder Overview

A user typically performs a series of steps to accomplish a logical task or, in Builder's terminology, a blueprint. Each blueprint may entail invoking many different applications and dealing with an equal or greater number of files or data objects. Builder uses a tape recorder paradigm as a model for describing application integration. The user turns on Builder to start a task, and applications are transparently recorded as they are used. This blueprint recording may be played back to reproduce the task.

Builder is one of five DECdecision components that run on the DECwindows platform. The others are as follows:

- Access — a table-oriented, database access tool

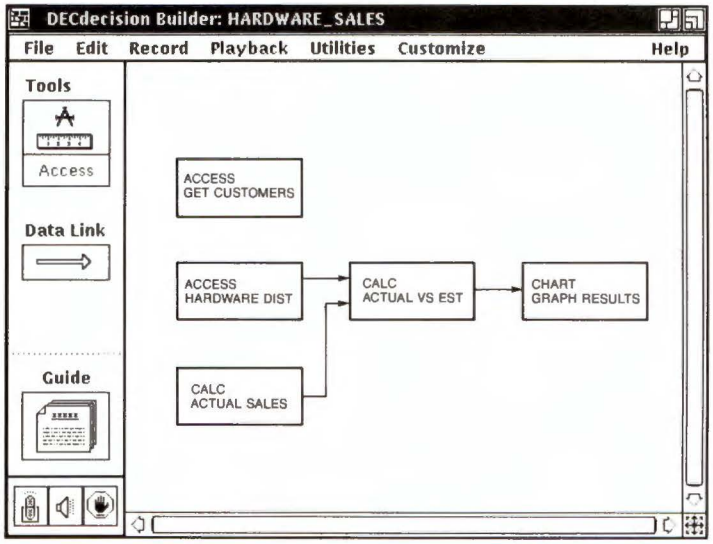


Figure 5 Sample Blueprint Diagram

- Calc — a spreadsheet package
- DECchart — a charting package
- Control — the manager for the other components

Builder ties together one or more of these components into a single task.

As applications are recorded, Builder graphically represents their activity with a series of boxes and arrows as shown in Figure 5. Each box represents an application instance; and each arrow, or data link, represents the flow of data (e.g., cut and paste) between a pair of applications. The user may select and display the contents of an object to see the operations that correspond to it, as shown in Figure 6.

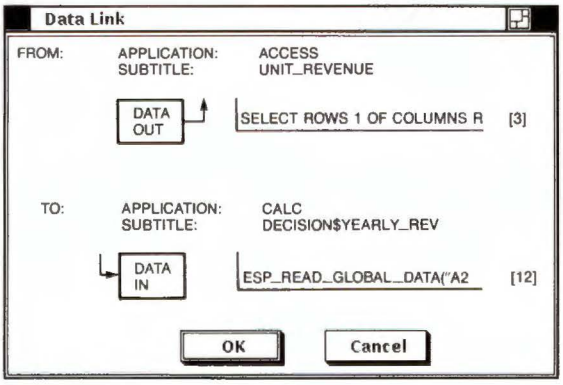


Figure 6 Sample Data Link

Early in the project, we decided that a record of keystrokes and mouse tracks was of limited use and could not always be counted on to reproduce the intended task. For example, if the application was run on a workstation with a different display size, or the menu choices were rearranged, many such recordings would be invalidated. We found that a functional record of operations, that is, a description of the intended operations, was much more useful and could be relied upon to represent user tasks.

The record and playback mechanism is therefore a cooperative effort between Builder and integrated applications. Applications that understand Builder are referred to as client applications. In record mode, a client application provides Builder with a functional representation of each user-level operation, which is recorded in the blueprint. Since only the application knows what a series of keystrokes and mouse tracks means, this cooperation is essential.

In addition to the basic record and playback capabilities, Builder also allows users to debug a blueprint during playback (e.g., single stepping), and to splice new operations into existing blueprints (a combined record and playback mode for modifying a blueprint).

Goals and Considerations

Besides the end-user goals, we designed Builder to meet key integration goals.

The main goal of Builder was to provide a facility for the recording and playback of operations that span more than one application. This included maintaining a record of the ordering of operations to account for concurrent applications.

Another goal was to support both procedural and nonprocedural applications. Many applications are process-oriented and are only controllable in a procedural or command-oriented way. Others are nonprocedural in nature or deal with objects, not commands. Both types of applications are supported. (Note: This paper refers to any operation that is recorded as a command, even if that operation was produced by a nonprocedural application.)

Further, we felt that introducing yet another command language would be detrimental to application integration and would discourage support for existing and future applications. Instead, Builder adopted an ecumenical doctrine with respect to command languages — any one is just as good as any other. With this method, application developers do not need to recode applications and cultural bias in command languages is avoided.

A prime goal for Builder was to support international applications. We had to recognize that applications can be translated to various degrees. Some areas that we addressed were as follows:

- Character set support: 8-bit strings, 16-bit strings, ISO Latin 1, etc. Blueprints are files encoded in the DDIF document interchange format. Commands can be recorded as compound strings to support a wide variety of character sets.
- Translatable and nontranslatable command languages. Applications may record a translatable and/or a nontranslatable command string for every operation, corresponding to the type(s) of language interfaces they support. If both string types are specified, they are treated as a logical entity. A scheme is provided for language-independent command execution that does not sacrifice the native language command interface.
- Language environment verification. Builder and client applications can verify that the record mode language is compatible with that of playback. For example, a blueprint recorded in German is meant to be played back using the German language version of an application.

Another goal was to ensure reusability. To minimize the impact of process startup and application initialization, Builder supports a mode of reusability whereby application processes are reused after task completion.

Builder also allows blueprints to be shared. Blueprints must be able to be mailed and used by other users. Since blueprints may contain references to other, embedded, blueprints, it was highly desirable for them to be shared as a whole. The DDIF data format encoding made this goal achievable. Embedded blueprints are named as external references in the DDIF data format file. The DOTS data object transport syntax packages and unpackages the resulting encapsulation.

Two final considerations shaped the design. One is support for nonintegrated applications. An application that does not support Builder may still be incorporated into a blueprint in a limited way by using it as a black box. The application is invoked and run to completion without the benefit of a continuous dialog with Builder.

The second consideration is coexistence with event-driven environments. Increasingly, application environments are centered around event-driven mechanisms that handle asynchronous events in an operating system-independent manner. Callbacks seem to be the most common and good solution to support this environment. Builder provides a callback registry for dispatching of operations. This mechanism also works well in a non-event-driven environment.

Design and Implementation of Builder

Sessions

When a blueprint is recorded or played back, Builder creates a new session. A session logically contains all information pertaining to the task at hand.

- Application ordering
- Blueprint commands
- Command sequencing
- State (record, playback, paused, etc.)
- Options (slow-motion speed, debug breakpoints, etc.)

Sessions are unique and are guaranteed not to conflict with other Builder users. Multiple sessions may be active simultaneously. Associated with every command is a unique sequence identifier (similar to a time stamp) that determines command ordering during playback.

When an application starts executing, it queries the Builder runtime services to determine whether Builder is active. If it is, the application joins the session and follows further instructions.

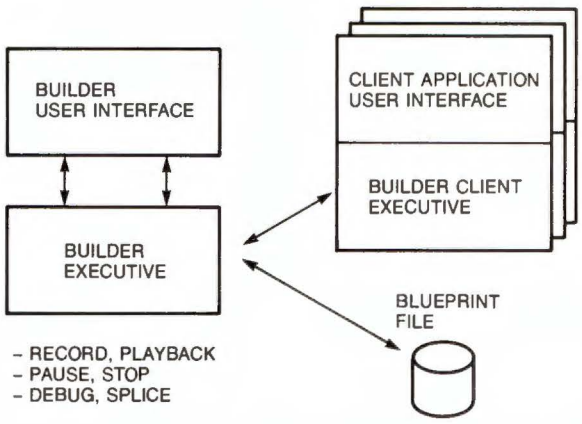


Figure 7 Builder Communication Model

An application may be instructed to process commands directly from the user, from Builder, or both. The Builder executive, in conjunction with the client executive, coordinates the switching of command focus. During record mode, applications process commands directly from the user, and during playback directly from Builder. Splice mode entails a bit of both.

The Builder Communication Model

The Builder communication model is basically a classic client-server model. The server is the Builder process, and the client is the client application. In Figure 7, the Server is labeled as Builder executive and the client as Builder client executive.

Builder Executive The Builder executive acts as a central dispatcher for sessions, handles all session-level operations, and communicates session information (updates) to client applications. Some examples of this include:

- Identifying which application starts next
- Invoking applications (and processing startup and termination information)
- Synchronizing cross-application events (e.g., which command executes next?)
- Handling errors during playback

The Builder executive can also start, stop, and pause a session, and reuse applications. However, it has absolutely *no* user interface. Instead, Builder uses the Builder executive as a session manager to derive the information needed to maintain the blueprint diagram. This is done by means of a call interface and callbacks, which are triggered when the Builder executive processes certain session events. These are separate and distinct layers, as depicted in Figure 7.

The Builder executive communicates with all client applications through a communication layer in the Builder client executive. (See Figure 8.)

Builder Client Executive The Builder client executive maintains the Builder context for applications, and does the actual communication between the client application and Builder. Client applications use Builder's runtime services as the interface to join a session, record commands, and indirectly communicate with Builder.

During record mode, the client application translates user operations into a command which it then passes to the client executive through the runtime services. The client executive assigns a unique, ordered sequence identifier, and records the result. These sequence identifiers represent the order in which the blueprint was recorded.

During playback, the client executive either determines or is notified when the next command in its command cache may be executed. When this occurs, the command callback registered by the client code is invoked and given the cached command. The application then parses and executes the command.

Blueprint File Blueprints are recorded on disk as a DDIF-encoded file. The blueprint file has a table of contents section and an application command table section. Figure 9 shows the file's logical layout.

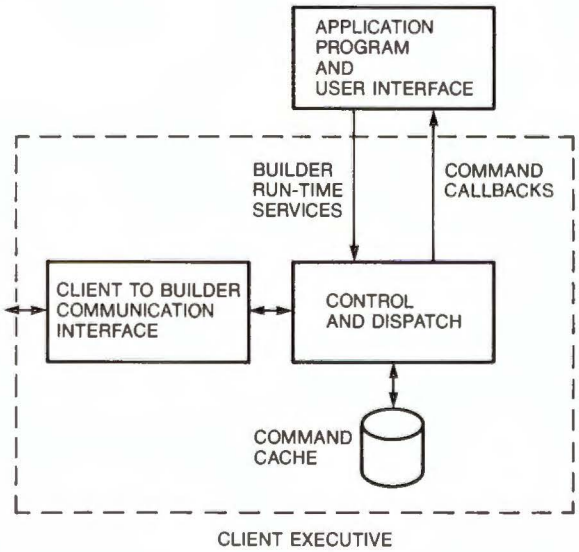


Figure 8 Builder Client Executive

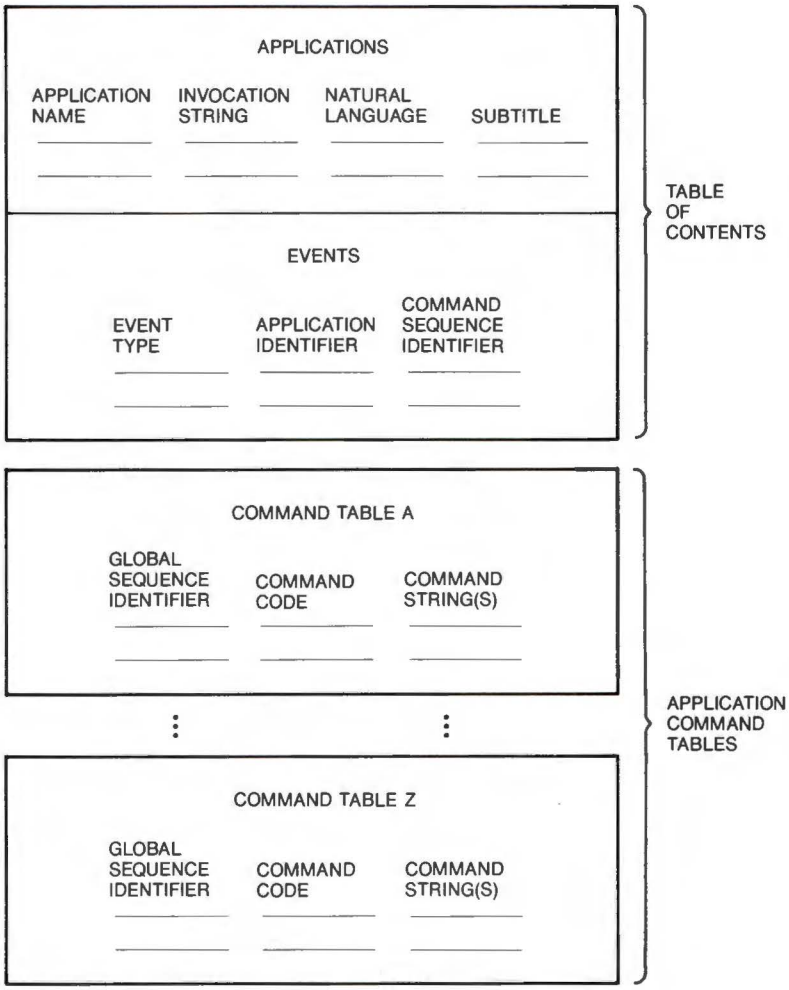


Figure 9 Blueprint File Layout

The table of contents section describes the applications that comprise the blueprint and their interrelationship. Included are application name invocation strings, and native language identifiers, among other application instance attributes. This information provides a framework description from which the user interface constructs a diagram. The table of contents also describes the cross-application events within the blueprint. The Builder executive uses these events when calculating application ordering.

One command table for each application is listed in the table of contents. All commands associated with a given application instance are grouped together. This grouping makes it easy to select and process all commands for a given application instance.

Translatability Applications may record one or two variants of command syntaxes:

- A native language-oriented and translatable syntax
- A common and nontranslatable syntax (which may or may not be culturally biased)

The advantage of supporting a native language interface is that the command language always addresses the end user in the local language. The disadvantage is that a series of commands recorded by an application in one country may not be understood by the same application in another country (assuming the product and command language have been translated).

The advantage of supporting a common command language syntax is that the same syntax is

understood across all application variants. The disadvantage is that common command languages tend to be heavily, culturally biased and, as a result, are only understood by a subset of users.

If an application supports a common and a native language interface, then the disadvantages of either in isolation are eliminated. This support also avoids the necessity of understanding all language variants of a command syntax. In essence, commands will execute regardless of application variant and without sacrificing the native language interface.

When an application supports both language syntaxes, Builder displays the native language commands but gives client applications the common language to execute. In addition, a degree of automatic translation may be achieved.

Trade-offs and Optimizations

In this section we address some of the design tradeoffs and optimizations made during the development of Builder.

Communication Protocol Builder's communication and notification protocol is designed to be tolerant of slow-responding applications. For instance, a client executive always has full access to the session state information, in case an application is in the middle of a long operation when a session update notification arrives. The client executive can reflect the current session state immediately instead of trying to process obsolete information.

Update Notification All communication can be handled in a straightforward manner between the Builder executive and client code through a bidirectional communication mechanism. The Builder design also allows certain communication operations to be implemented in a distributed manner, including

- Updating session information
- Updating sequencing/synchronization

Session and sequencing information needs to be updated and distributed quickly to all client applications. If distribution is not quick enough, playback especially will not appear to be responsive to the Builder user interface.

On the VMS system, session and sequencing information is maintained in locks, using the VMS $ENQ and $DEQ services. Notification of asynchronous behavior, such as changes to the playback speed, is handled using a technique called blocking Asynchronous System Traps.

The current sequence identifier and general session information are maintained in separate locks. They can be modified independently, and applications are notified of the updates. The lock protocol is designed so that the Builder executive can trigger callbacks in *all* client executives that are part of the session by simply seizing and updating a lock. This optimization obviously can only be used when Builder and the client are on the same VAX system cluster.

Local Command Caching The client executive can send commands directly to the Builder executive during recording, or it can record commands locally and have them merged into a master blueprint later. These two methods are functionally equivalent.

However, the latter is more efficient and is used during normal record mode. The former is used only when the Builder user interface wants to maintain an up-to-date display of commands, such as during splice mode.

During playback, rather than send application commands to the client executive one at a time, the Builder executive either ships them all at once when the application starts up or points the client executive to a local file. This is more efficient and does not introduce any playback latency due to temporarily slow communication conditions.

The goals in these optimizations were to keep communication between Builder and client at a minimum and to distribute as much work as possible. Any potential bottleneck in the processing of communication messages that would have made the Builder user interface seem unresponsive was avoided.

The client executive only notifies the Builder Executive when the following events occur:

- An application joins the session
- A cut or paste operation is encountered
- The application is synchronizing on another (application's) command
- An error is detected
- The application exits the session

These events might require the user interface to be notified or some interapplication synchronization.

Command Synchronization In keeping with our desire to minimize communication with Builder, the client executive only requests command synchronization if it cannot proceed on its own. Because

all commands are sequentially numbered, the client executive can examine the next command in its cache to determine whether it will execute next.

Parallel Playback The current implementation supports only single-threaded execution. However, the design does allow Builder to relax this constraint and execute playback in parallel. The design identifies certain operations that would require synchronization even while in parallel playback, including application startup, application termination, cut, paste, etc. These operations constitute hard synchronization points that must be obeyed for playback to perform properly.

This feature was not made available through the user interface because of possible situations unique to the user's environment.

For instance, a user may have a blueprint in which one application updates a personal customization file (operation A) and then relies on that within another application (operation B) in the same blueprint. Since the relationship between the two operations lies in the user's mind, Builder cannot synchronize them. In parallel playback, sometimes operation A would execute first and other times operation B.

Summary

The AIL library and DECdecision Builder meet the goals we initially established. Primarily, the software was to provide the capability to perform application invocation and interapplication communication, thus enabling the DECwrite and DECdecision products to work closely together.

We used the same project approach for each. We began with an initial basic design on which we layered components for more sophisticated processing operations. As we progressed, we made design trade-offs and optimizations that resulted in an effective and efficient implementation. In the end, we delivered interapplication access and integration support that enables an application to be called, its features used, and the result incorporated automatically.

Acknowledgments

The authors wish to thank the many individuals who have participated in the design and review efforts. Special acknowledgments are given to Seth Cohen, Carol Young, Peter Savage, Peter Bower, Ann Wong, and Brian Simons who contributed to the initial design and led the DECwrite and DECdecision products in incorporating AIL and Builder supports. We would also like to thank our managers Shapoor Shayan, Dennis Saloky, and Steve Baron who encouraged and drove the application integration effort.

References

1. A. Sung, N. Jacobson, and C. Young, "The Design and Development of the DECdecision Product," *Digital Technical Journal,* vol. 2, no. 1 (Winter 1990, this issue): 60–72.
2. S. Cohen and W. E. Morgan, "The Relationship between the DECwrite Editor and the Digital Document Interchange Format," *Digital Technical Journal,* vol. 2, no. 1 (Winter 1990, this issue): 73–82.

Alan Sung
Neal F. Jacobson
Carol A. Young

The Design and Development of the DECdecision Product

The DECdecision product is an end-user decision support application composed of five components that perform database access, spreadsheet, charting, flow control, and management functions. Each component presents a consistent, rich, graphical DECwindows user interface. Users can easily share data between the components, or with other applications, using the DECwindows QuickCopy and clipboard facilities. The CDA architecture supplies the foundation for this data interchange, as well as support for reading or writing DECdecision data in a variety of formats. The DECdecision product provides a level of sophistication and seamless data integration not found in many products. The DECdecision product is one of the first, large-scale applications to showcase the capabilities of DECwindows and the CDA architecture.

Introduction

The DECdecision application provides an end-user decision support environment for data-dependent professionals. Its target audience includes both novice and expert users in any profession that relies on data (e.g., finance, marketing, sales, engineering). The DECdecision product is composed of five integrated components:

- Access, a table-oriented database access and query package
- Calc, a programmable spreadsheet package
- DECchart, a business charting package
- Builder, a flow control facility
- Control, the manager for the other four components

Access provides an end-user view into a relational database through sophisticated ad hoc query and reporting features. Calc provides analytical and "what-if" spreadsheet capabilities, including remote grid consolidation and linking, charting, and a procedural macro language. DECchart supports business charting and annotation, including standard and user-defined charting styles. Builder brings the actions in these three DECdecision components together into a blueprint that can be replayed at a later time.[1] Control manages the components. It serves as a single point from which each component can be invoked and a central point from which customization features can be controlled.

The DECdecision product showcases the capabilities and features of two major Digital architectures: DECwindows and the CDA architecture.[2,3] The DECdecision product is one of Digital's first, large-scale applications to demonstrate in a single product the power of the graphical, mouse-based, DECwindows user interface, and the data interchange and conversion capabilities of CDA.

DECdecision's user interface highlights the wide range of graphical building blocks (i.e., widgets) available to a DECwindows application developer. These widgets include dialog box, list box, scroll bar, push button, text entry, and file selection widgets. Each of these widgets is supplied with the XUI toolkit. When used in conjunction with the recommendations defined in the *XUI Style Guide* and the User Interface Definitions Language (a utility to define and build user interfaces), the DECwindows widgets make it easy to create applications that have a consistent look and feel.[4]

The DECdecision product also demonstrates the interactive data interchange capabilities provided by the DECwindows clipboard and QuickCopy mechanisms. The clipboard provides a system-wide repository for application data. It stores or retrieves data passed by or requested by applications. As opposed to the clipboard, QuickCopy uses a direct connection for data transfer between applications.

The CDA architecture provides the underlying data interchange format that the DECdecision product uses in both the clipboard and QuickCopy

operations. The CDA architecture supplies a set of services that facilitate data interchange between applications, including services to convert data to and from the CDA formats. With the CDA services, the DECdecision components can easily share data among themselves or with other CDA-compliant applications. The CDA converter architecture allows DECdecision components to read and write data in a number of data formats, such as WK1, DECalc, ASCII or PostScript.

Through the DECdecision product, we hope to encourage additional application development based on DECwindows and the CDA architecture, both within Digital and by third parties. The DECdecision product demonstrates that a consistent and flexible DECwindows user interface combined with the CDA data interchange and conversion architectures can provide a powerful base for application developers to build sophisticated, state-of-the-art applications.

DECdecision Product Background

A major schedule goal for the DECdecision product was to release it very soon after the release of the DECwindows system. To meet this goal, we re-used code from existing Digital products where possible to reduce development time.

The DECdecision components Access and Calc were developed from the existing Digital products VAX TEAMDATA and VAX DECalc, respectively. These existing products have a character-cell terminal user interface. This user interface was replaced with a DECwindows graphical, mouse-oriented user interface, which was layered on to the processing engines of each component. New functionality was also added to each component that was not present in the existing products.

The DECdecision DECchart, Builder, and Control components were developed from scratch because there were no existing Digital products that provided the functionality these components needed.

The DECdecision product development project presented some interesting engineering challenges. The two existing products had been developed as standalone applications. Each had its own particular look and feel as well as its own particular user community accustomed to operating in a particular way. In merging these two products and the new components into the single DECdecision product, we placed particular emphasis on the integration points between the components: user interface, data interchange, and Builder interaction.

This paper focuses on some of the more important DECdecision design and development goals and decisions. This discussion highlights how DECwindows and the CDA architecture were used to develop the DECdecision components. In some cases, the DECdecision development actually enhanced the DECwindows product. This paper also discusses the above-mentioned integration points, some of the more interesting component designs, and some of the optimizations built into the DECdecision product.

Integration

User Interface

User interface consistency is very important in the DECdecision product because the product includes three data access and analysis components, each of which has specialized capabilities. While some capabilities and features of each component are quite different, the components also share many similarities. We felt it was important that similar or identical commands and operations be consistent across each component. Consistency reduces the time it takes a new user to learn the system because it is easier to transfer knowledge learned in one component to another.

We concentrated on consistency of menus, dialog boxes, selection operations, and keyboard accelerators. Similar or identical menu commands in each component have the same names; where appropriate, dialog boxes have similar or identical options and layout. Selection within the table work areas of each component is identical, and the same keyboard accelerators are used across components.

The DECdecision product conforms to the *XUI Style Guide* and uses the XUI toolkit widgets and User Interface Definition Language (UIL/UID). Through internal reviews, we were able to achieve a high degree of user interface consistency among the DECdecision components and between the DECdecision product and other DECwindows applications.

Data Integration

The second DECdecision integration area is data integration. While each DECdecision component alone can be used for many useful functions, the real power of the DECdecision product lies in its ability to integrate data from a variety of sources and to interchange data between the components.

Data interchange allows the DECdecision components to communicate with each other as well as with other applications.

Results from an Access query can be moved to a Calc spreadsheet for analysis. Charts can be created from database or spreadsheet data, and the chart can be updated as the data changes. Data tables and spreadsheets created by external applications can be imported into the DECdecision product. These imported data tables retain most, if not all, of their revisable data (e.g., formulas, data typing, formats). Conversely, DECdecision data tables and spreadsheets can be exported (written) to externally defined formats. Even text which *looks* like a data table, such as a mail message, can be easily read into the DECdecision product for analysis or computation. Finally, data tables or charts can be linked to compound documents using the DECwrite editor's LiveLink mechanism.[5]

This flexible data-sharing is accomplished using the CDA architecture and the DECwindows clipboard and QuickCopy mechanisms.

CDA Architecture Support

The CDA architecture is a set of architectures and services which facilitate the interchange of compound documents. The CDA architecture has three main component architectures that are relevant to the DECdecision product — the DTIF table interchange format, the DDIF document interchange format, and the CDA converter architecture. The DTIF table interchange format defines a format for the interchange of revisable data tables.[6] The DDIF document interchange format defines a format for the interchange of compound documents containing text, graphics, and images.[7] The converter architecture defines a mechanism to convert document data to and from the CDA formats. The CDA toolkit provides services to create, process, or view CDA documents and to invoke document converters.[8]

Each DECdecision component supports the CDA architecture. Data interchange between DECdecision components uses the DTIF format, since the DECdecision components are primarily data table-oriented. Data interchange between DECdecision and other applications, such as the DECwrite editor or the Mail utility, uses the DDIF format, since communication and data exchange outside the DECdecision components are often in the form of a report or chart.

The DTIF format is the native storage format for Calc spreadsheet and DECchart worksheet files. DTIF format data tables may be read or written by Access, which uses Rdb/VMS as its native storage format.

Data transfer between components using DECwindows QuickCopy or the clipboard also uses the DTIF format. (This is discussed in more detail in the QuickCopy section below.) Access and Calc send DTIF-formatted data to DECchart when creating or updating charts.

The DDIF format is the native storage format for DECchart chart, style, and overlay files. Printed output from Access, Calc, and DECchart, such as reports and charts, is written in the DDIF format. LiveLink data, sent when a DECdecision component is linked to a DECwrite data block, is encoded as a DDIF document.

Access also allows those compound document references that are stored within a table cell to be displayed using the CDA viewer.

For example, a real estate database might store references to house pictures or a parts database may store references to schematics.

CDA Converter Support

The CDA converter architecture provides a mechanism to convert document data to and from CDA formats, using programs called converter modules. Modules that convert data from a non-CDA format to a CDA format (DTIF or DDIF) are called front-end converters. Modules that convert data from a CDA format to some other data format are called back-end converters. In this model, the DTIF and DDIF data formats are conversion hubs. Each front-end or back-end converter module connects to the hub. The architecture also allows new converter modules to be easily added to the system. The CDA toolkit includes the routines to initiate a front-end or back-end conversion. (See Figure 1.)

The DECdecision Access, Calc, and DECchart components all use the CDA toolkit to read and write DTIF or DDIF data. In addition, we wanted each component to be able to read or write non-CDA data formats, such as WK1, DECalc, DIF, or ASCII. Using the CDA converter architecture helped us solve this problem in two ways:

- The conversion hub model enabled each component to support a single format (DTIF). The converter architecture manages the translations to and from the hub format at the component's request.

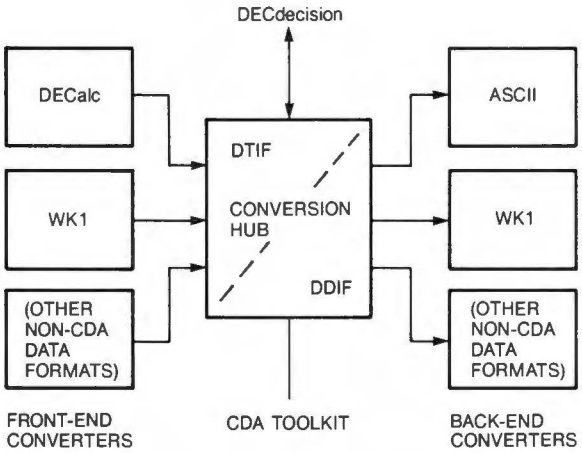


Figure 1 Conversion Hub Model

A considerable amount of coding and testing work was eliminated because each component already included support for the hub format.

- The actual conversion process is de-coupled from the application requesting the conversion. This de-coupling means the application needs no direct knowledge of the available converters, and converters can be added to or subtracted from a system without requiring any modification to the DECdecision application.

We designed a dynamic scheme to display the list of converters currently installed on a system each time the DECdecision product displays an import or export dialog box. DECdecision contains no hard-coded information about the converters available on any given system. The converter architecture simplified this task for the DECdecision product to the point where import and export operations became primarily a user interface issue.

Each time the user selects an import or export operation, the DECdecision product displays a list box that contains the names of all converters available on the system at the time. This list is created by searching the CDA converter library directory for files that match the CDA converter naming rules.

For import operations, the DECdecision product searches for DTIF front-end converters using a wildcard file specification. All files matching this specification are assumed to be valid CDA converters. The format name is extracted from the expanded file specification and added to the format choice list box. When a format choice is selected, its format name string is passed to the CDA converter kernel to indicate which converter is requested. The converter kernel invokes the appropriate converter, and the requested format data is converted to a DTIF in-memory structure. The DECdecision product then processes the DTIF in-memory structure as if the data had been originally entered in the DTIF format.

Export operations are similar to import operations. However, the DECdecision product searches for back-end converters whose hub is either DTIF format or DDIF format. The DDIF hub format is available to the DECdecision product because the DTIF-to-DDIF domain converter creates a bridge between the two hub formats. The domain converter transforms DTIF tabular data into a DDIF document, applying information included in the table such as column widths, value formatting, line and page breaks. Once the table has been converted to a DDIF document, it can then be converted into any format for which there is a DDIF back-end converter, such as PostScript or ASCII. The DECdecision components Access and Calc use the domain converter for print and report operations as well.

DECwindows Data Interchange

The DECdecision product utilizes two techniques for interactive data interchange between its components and other DECwindows applications. These techniques are QuickCopy and the clipboard. Using these techniques, DTIF data is interchanged between the DECdecision components to preserve as much tabular information as possible, and ASCII data is interchanged with other applications that do not support the DTIF format.

QuickCopy

The DECwindows QuickCopy feature is a highly efficient method to transfer data between applications. The DECdecision components use QuickCopy to exchange tabular data among themselves. The user initiates QuickCopy by selecting one or more sections of data to be transferred. The user then moves the mouse to the data's target location and transfers the data by pressing the third mouse button.

The Access, Calc, and DECchart components support two data formats, STRING and DTIF. STRING identifies ASCII-encoded data, and DTIF refers to the DTIF table data format.

DTIF-encoded data may be quite extensive. It can range from a single cell value to an entire spreadsheet or data table. Because of memory and

other application resource restrictions, it would be unwise to transfer this data all at one time. Instead, the DECdecision components use incremental QuickCopy to transfer DTIF data in manageable units. The size of STRING data, on the other hand, is usually quite small (e.g., data obtained from a screen window). Therefore, the nonincremental QuickCopy mechanism is used.

The incremental QuickCopy protocol was not initially part of the XUI toolkit. The DECdecision product defined the protocol and call interface and was the primary tester and consumer of this functionality. These routines are now part of the XUI toolkit and can be used by any two cooperating applications. The following is an example of how two applications exchange data through the incremental QuickCopy mechanism.

The sending application, in response to the user selecting a data range, exerts ownership of the primary selection by making a call to the routine XtOwnSelectionIncremental. It also registers a callback routine name, which will be called when data is requested by the receiving application.

When the receiving application wants the DTIF-encoded data, it calls the routine XtGetSelectionValueIncremental. It also registers a callback routine, which is called when the data actually arrives at the receiver's end. This starts the send and receive communication between the two applications.

The first time the sender's XtConvertSelectionIncrProc callback routine is called, it encodes the selected region of the table into DTIF data. The data is actually written to a temporary file to make the data processing more manageable.

The first data increment is read from the temporary file and returned, along with its length, to the XUI toolkit. The size of the data buffer is based on the optimal network packet size, which the XUI toolkit passes to the sender's callback routine. The XUI toolkit subsequently sends the data to the receiving application.

On the receiver's end, its XtSelectionCallbackProc routine is invoked when a data increment is received. The receiver writes this data to its own temporary file, which it maintains as long as the data transfer is active.

These sending and receiving operations continue until all the data has been transferred. The sender indicates this by returning a "null" data increment, that is, one whose length is zero. At this time, the sender may choose to delete its temporary DTIF data file.

When the receiver sees a null data increment, it realizes the entire data transfer has completed and closes its temporary DTIF data file. Then, it reads the DTIF-encoded data and decodes the data into its own internal form. After the DTIF data has been processed, the receiver may choose to delete the DTIF file.

Temporary files are not strictly necessary in this operation, but we chose this approach for several reasons. First, the total size of the transferred data cannot be known in advance. In fact, it is not always possible to allocate sufficient memory to hold all the data. Second, storing the entire data segment in memory reduces the amount of memory available for other uses, such as cell data storage. Figure 2 provides a detailed example of the incremental QuickCopy procedure in the DECdecision product.

In some cases, the sending application's XtConvertSelectionIncrProc cannot convert the data in its internal form to DTIF format. When this happens, the sender indicates to the DECwindows toolkit that the sender is not able to provide the data in the requested format. The receiving application has two options when the sender has not provided the data. Either it can abort the data transfer and display the appropriate message, or it can retry the data transfer in another format. The Access, Calc, and DECchart components all retry the data transfer operation utilizing nonincremental QuickCopy with the STRING data type. This operation is simpler since the receiving application will receive only one buffer of data and no handshaking is necessary. The Calc and DECchart components process this buffer of STRING data directly into their internal form. The Access component writes the buffer to a temporary file and imports this file as an ASCII tabular file into its internal form.

Clipboard

An alternate method of transferring DTIF data to and from the DECdecision product is the DECwindows clipboard. The main reason to use the clipboard instead of QuickCopy is that the clipboard has some semblance of permanence. In a QuickCopy operation, both applications must be active to interchange data. With the clipboard, the sending application can place data on the clipboard and then terminate. At a future point, the receiving application can retrieve the data from the clipboard without the sending application being active.

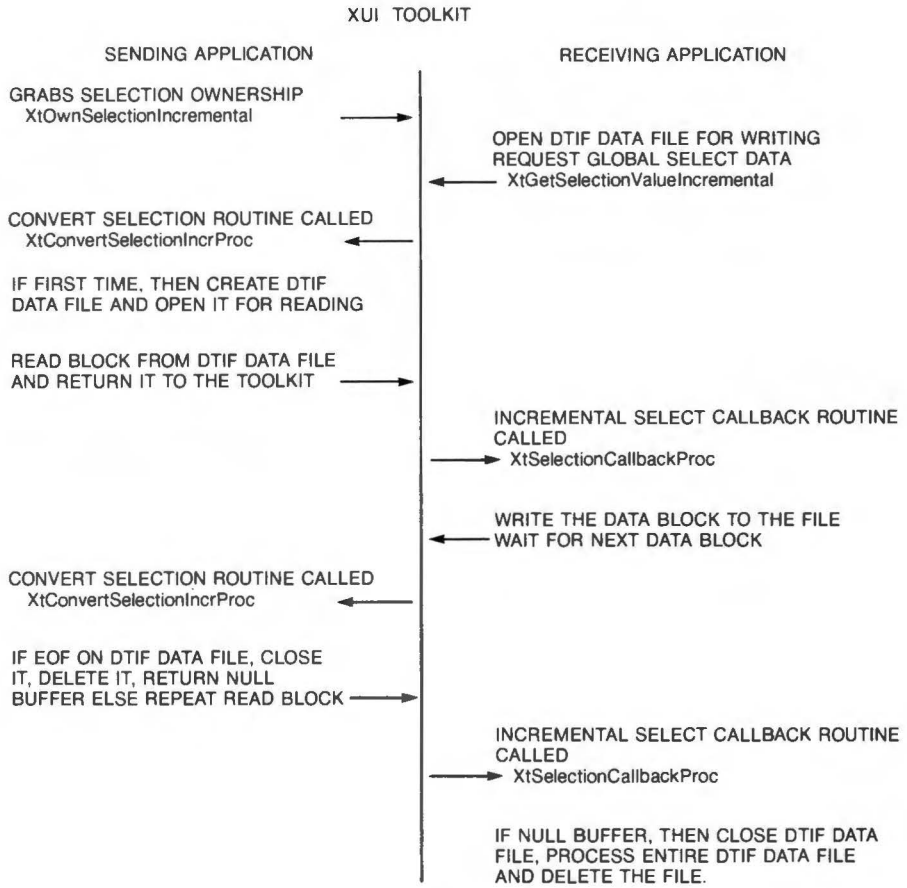


Figure 2 Incremental QuickCopy

The DECdecision components place data on the clipboard using the pass by name method, whereas other applications might use pass by value. In pass by name, when a cut or copy operation takes place, a snapshot of the data is placed in a holding spot within the component. A named reference is then placed on the clipboard rather than the actual DTIF data. The rationale for this is that a cut or copy operation does not necessarily imply data interchange between applications. Therefore, with the clipboard we could defer the transfer of the actual data until the data is really needed. For example, a cut operation on a table might be simply to clear the contents of some cells. The user does not intend to pass data to another application. By using the pass by name reference, we avoid encoding the data into DTIF format, transferring the data to the clipboard, and consuming excess server memory in this type of situation.

Because the clipboard can only transfer a limited amount of data and does not provide any incremental data transfer method as does the QuickCopy operation, QuickCopy is the preferred method on the DECdecision product for data interchange between active applications.

Charting Using the DECchart Component

The Access and Calc components utilize DECchart for charting functionality. Access charts statically, meaning that the user explicitly initiates the draw chart operation. Calc can also chart statically, but has the unique ability to chart dynamically. Dynamic charting means that charts are redrawn automatically as the values within the worksheet that affect the chart change.

A component is responsible for maintaining the information necessary to control and communicate

with the DECchart process. This includes collecting the data to be graphed from Access or Calc, along with any annotation, such as titles, subtitles, and axis labels. This information is then sent to DECchart using the dynamic data exchange feature of AIL.

Dynamic charting from Calc is handled as functions within cells. These functions include some predefined chart style functions PIE( ), LINE( ), BAR( ), etc., including a general purpose CHART( ) function.

This process allows Calc's internal calculator to track the dependent values in the worksheet that comprise the chart. When the internal calculator recalculates a spreadsheet and detects that the dependent values have changed, it automatically sends the necessary data to DECchart through the same basic mechanism as in static charting.

For charting functionality, both Access and Calc control and communicate with DECchart using the Application Interface Library (AIL) rather than a standard call interface. AIL provides an easy and extensible method for passing data and commands to DECchart. Since DECchart occupies its own process space, AIL allows chart drawing operations to be done in parallel with Access and Calc operations. The three main requirements for the interface were:

- Use the existing DTIF architecture to pass data
- Provide a means to pass pieces of data as logical entities
- Provide a means to pass action command

Drawing a chart consists mainly of building an item list of logical entities (each logical entity in the list contains a value, a command, and a format code), and calling the AiSendData routine. When the sending application has sent all relevant information, it instructs DECchart to draw the chart. DECchart then synthesizes all the information stored internally, including any information passed to it, to create the chart to be displayed.

Logical entities are divided into two categories. The first category is comprised of descriptive entities and the second is comprised of data entities. All entities can be sent in any order and can be received in any order.

In addition to logical entities, commands to control charting operations can be sent through the interface. The most commonly used command is the draw chart command. When DECchart receives this command, it opens and reads all received files. All descriptive entities received, such as title and subtitle, are merged internally. The DTIF data is processed, and the final chart is displayed to the user.

The following steps outline the basic sequence of displaying a static chart:

1. The information about the chart to be drawn must first be collected. Typically, a dialog box is used to gather the data, although the information can be specified on a command line or from within a macro.
2. The information from the dialog box or command line representing descriptive entities is then processed into an AIL send data item list.
3. The data range or series specified must then be encoded as a DTIF document. Both Calc and Access encode the DTIF document directly into dynamic memory to avoid any file input/output overhead. A pointer to this memory is appended as a data entity to the send data item list.
4. The entire send data item list is communicated to DECchart.
5. After DECchart receives the descriptive and data entities, a control command is sent to generate and display the final chart.

Access and CALC Components

This section briefly describes the internal design of the DECdecision Calc and Access components and provides some insight into the engines beneath the graphical user interface.

DECdecision Calc Design

The Calc component was designed in conjunction with the DECwindows VAXTPU text processing utility. With a spreadsheet capability layered over a text processing utility, Calc has more power and functionality than the majority of other commercial spreadsheet packages.

From the outset of the DECdecision Calc design effort, it was a requirement that to be competitive the component must have some macro language capability. DECdecision Calc's spreadsheet engine, the subsystem within a spreadsheet responsible for internal calculations and matrix management, was derived from that of the DECalc product. However, at the time, DECalc did not have any macro language. To retrofit the DECalc product's spreadsheet engine with a powerful and flexible language would have required many man-years of effort. While our development team had considerable expertise in spreadsheet operations, we had little expertise in

the development of third-generation languages, such as compilers and interpreters.

The search for a high-level, procedural language subsystem led to the VAXTPU utility. VAXTPU is a high-performance text processing utility that had a complete, extensible, well-tested and mature language. We decided to use the VAXTPU language as the basis for the DECdecision Calc language. This decision solved the problem of learning another programming language and environment. If a user understood the fundamentals of the VAXTPU language for text processing, then the learning curve to use the language in a spreadsheet processing environment would be minimized.

Another benefit of using the VAXTPU utility as the basis for our new design was that the VAXTPU utility adhered strictly to the separation of form and function. VAXTPU provides only the function, or the raw text processing primitives. It contains no user interface. The user interface, known as the form, is provided separately. We wanted to have a similar model. That is, we wanted a spreadsheet processing utility that provided only the function, with a separate user interface package layered on this newly created spreadsheet processing utility. The major development tasks to develop this for DECdecision Calc were to:

- Build a layer around the existing DECalc product's spreadsheet engine. This layer, when combined with the VAXTPU utility, enabled the VAXTPU utility to understand and manipulate spreadsheet functionality.
- Create a new spreadsheet user interface using the design of EVE as the basis for the design of the new user interface. The EVE extensible video editor was the textual user interface layered upon the VAXTPU utility.

Access to the functionality and display of spreadsheets was accomplished through the VAXTPU extension mechanism. This extension consisted of creating:

- A set of four new VAXTPU data types to enable VAXTPU's interpreter to understand and manipulate spreadsheet entities
- A new, additional screen update mechanism to handle the display of tabular data
- An environment in which the VAXTPU base system could operate
- A set of built-in procedures and keywords which would allow the VAXTPU programming language access to the low-level callable interface of the spreadsheet engine

A single VAXTPU data structure, known as the extension table descriptor, pulls together the various modules described above. This extension table descriptor is registered with the VAXTPU base system during the VAXTPU initialization phase.

There are many similarities between a text editor and a spreadsheet since a spreadsheet can be construed as a tabular data editor. Operations such as entering, inserting, moving, deleting data, and file operations are analogous. The basic difference is the primitive data types that are manipulated. Text editors process on a character-by-character basis and have an inherent understanding of words, sentences, and paragraphs. The spreadsheet operates on a cell level and processes data in rows, columns, and rectangular regions. In order for VAXTPU to understand these new primitives, four new data types were introduced. Each new data type was modeled after an analogous entity in the text paradigm. These entities included:

- A grid which is a collection of cells in a tabular form. The grid is analogous to the text buffer, which is a collection of a stream of characters.
- A view, which is a tabular viewing area for the display of a grid. A viewer is analogous to a text window, which is used to display a buffer.
- A grid marker, which marks a point on a grid. A grid marker is analogous to a text marker, which marks a point on a buffer.
- A grid range, which marks a rectangular area of a grid. A grid range is analogous to a text range, which marks the beginning and end of a sequence of characters in a text buffer.

The creation of the view data type required an additional mechanism to display tabular data. The existing VAXTPU display mechanism could display individual characters on a terminal output device. To display tabular data, the table widget was developed. The table widget is basically a display for tabular data that handles cell storage, alignment, borders, rendition, etc. It can only store those cells that are currently visible. Therefore, cell data must be provided for the widget when the table is scrolled or increased in size. The screen updater controls

the table widget. A minimal update algorithm maintains a list of cells that change values as the internal spreadsheet engine computes. When no further calculations are necessary and there are no user events or requests to process, the screen updater is run. VAXTPU can invoke screen updaters that have been specified in any of the registered extension table descriptors. When Calc's screen updater is called, any cell that is on the modified list and visible is retrieved from the grid, formatted, and sent to the table widget for display.

A set of built-in procedures that permitted the language system to access the actual spreadsheet functionality were then created. The base VAXTPU system provides built-in procedures for text manipulation, such as moving text, copying text, text insertion and deletion, and window control. The Calc built-in procedures operate on the newly created data types and provide analogous functionality to the VAXTPU built-in procedures for cell level manipulation, such as, moving cells, copying cells, entering and clearing data from cells, and view control.

The Calc built-in procedures are grouped into several different categories. These categories include screen layout, cursor movement, editing position movement, text and data manipulation, file processing, spreadsheet calculation, pattern matching, editing context status, defining keys, multiple processing, program execution, DECwindows processing, DECdecision Builder recording and playback, and AIL processing.

Each built-in procedure and variable is described in the *DECdecision Calc Macro Guide* and the *VAX Text Processing Utility Reference Manual.*[9,10]

With all these components in place, one can think of Calc as an erector set of spreadsheet parts. The Calc language uses the spreadsheet parts as the building blocks to create a worksheet user interface. The flexible combination and use of the built-in procedures, even at run-time, make the Calc component a unique spreadsheet application.

The Calc user interface was developed by both layering upon EVE and by utilizing the newly created spreadsheet-specific data types, keywords, and built-in procedures. In parallel to our design project, the VAXTPU group was working to add DECwindows support to the base VAXTPU system. The DECwindows VAXTPU project gave the newly created Calc user interface access to a subset of the DECwindows functionality. This access included creating and managing widgets and widget callbacks, and access to the DRM (Digital Resource Manager) database. This support in the base VAXTPU system significantly reduced the time and effort necessary on our part to create a robust Calc user interface. The new user interface, called ESP or extensible spreadsheet package, is completely customizable and extensible. Few DECwindows applications possess the customization features available in Calc.

The majority of ESP is stored in the DECdecision product on a permanent basis in a VAXTPU section file. Widget definitions and internationalizable text strings are stored in a UIL/UID file. ESP can be changed through the textual processing portion of the user interface or EVE. Users can either create new or modify existing macros and procedures, key and mouse button definitions, and widget definitions. These changes are made known to the system by compiling the language statements, a process known as extending. Once in a compiled form, the VAXTPU interpreter can easily process the new or altered functionality. If any run-time alteration is made to ESP, the altered user interface can be saved to a new section file. If a different section file is specified when the Calc component is invoked, a different user interface appears. By using section files, users can fine-tune their own spreadsheet operating environment, in the same way text editors can be fine-tuned. This flexibility has proven highly successful and desirable in both EVE and EDT.[11]

Figure 3 illustrates the resulting design of the combination of VAXTPU, EVE, Calc, and ESP.

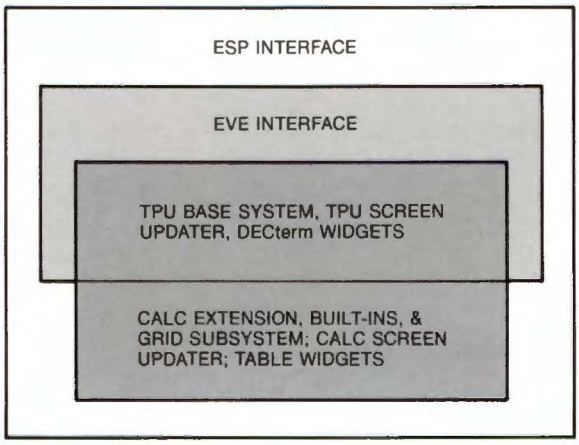


Figure 3 Picture of Resulting VAXTPU/EVE/Calc/ESP Design

DECdecision Access Design

Access provides DECdecision's database management capabilities. Data tables may be either private to a particular user or shared throughout a department or a corporation. Access presents all data in a tabular format. A row in the table corresponds to a record in the database, a column corresponds to a field. Access supports several different database formats: Rdb/VMS relational database, IDMS/R and DB2 databases, RMS files, and VAX DBMS CODASYL-compliant databases. Access uses the VAX to IBM data access connection and VAX DATATRIEVE domains to support some of these databases. Regardless of the database format, Access presents data in the same tabular form and users interact with the database through a single user interface. DECdecision Access manages the details of mapping user requests or queries into each individual database's proper language.

Access uses a collection approach to handling data. All queries and modifications operate on the current collection. Users can express complicated queries in steps, which eliminates the need to learn a cumbersome data manipulation language (DML).

Internally, Access is divided into three subsystems: user interface, collection manager, and query emulator. Figure 4 shows their interrelationship.

The Access user interface subsystem handles all screen and user interaction. This interaction includes gathering query or other command information from the user and displaying a formatted representation of the current collection. The user interface also processes DECwindows events such as exposure, selection, or QuickCopy data requests. The user interface component has no direct knowledge of the database underlying the Access system. Database manipulation operations, such as queries, record modification, or column creation operations are handled by the user interface generically. Generic requests are sent to the collection manager, which either performs the operation directly or translates the request into operations that the underlying database understands.

The collection manager subsystem maintains knowledge about the currently active database connections, as well as the functions supported by each database system. Generic requests received from the user interface are translated into the specific syntax used by a particular database. The collection manager can receive a request for an operation that is not supported by the database. In this case, the collection manager will attempt to emulate the operation to provide the requested

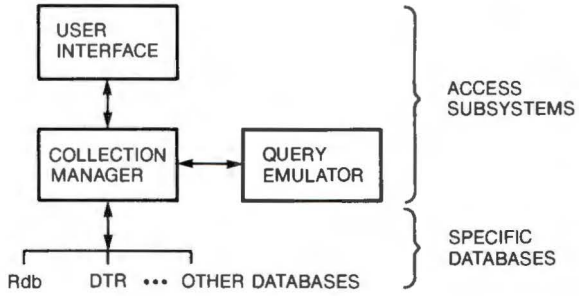


Figure 4 Access Subsystems

information to the end user. Database operations, such as table joins, are emulated by the collection manager itself. Query evaluation and aggregation, such as computed columns, are emulated by the query emulator.

For example, for a database system that does not support queries based on partial string matches (e.g., find states which start with *N*), the query emulator will emulate the operation itself. The query emulator also performs operations on temporary columns, which are not actually stored in the database. Temporary columns may be created as the result of a table join or an aggregation command (such as TOTAL). Table join operations are discussed in more detail below.

The collection manager maintains a list of unique keys, one for each record in the current collection. A key is returned by the database after a query, and may be subsequently used to directly access the record. The collection manager maintains a cache of records from the current collection in memory to minimize database access requests. It does, however, limit the amount of cached data stored to avoid consuming large amounts of memory.

The collection manager also maintains a list of the queries used to form the current collection. As subsequent queries are requested, the collection manager attempts to combine the queries to form a single new query. The single query is then sent to the database. For example, the initial query is "Find employees where salary >$40,000." The database returns a sequence of records matching the query. These records become the current collection. The next query is "Find employees where age <35." The collection manager combines this query with the previous query to form the new query, "Find employees where salary >$40,000 AND age <35."

As noted above, the collection manager emulates table joins (e.g., across different database systems).

The current collection can be joined over common columns with another database relation, which results in a new collection that contains columns from each relation. For example, the current collection could contain employee salary data indexed by employee identification. A second table could contain employee benefit information, also indexed by employee identification. The collection manager joins the benefits table with the salary table by using the common field, employee identification. The result is a single table that displays both salary and benefit information. While the user interface believes it is displaying a single data table, the join is, in fact, being emulated by the collection manager.

Builder Integration

In order to accomplish a task, a user typically invokes the DECdecision components, incorporates data from one component to another, and possibly processes it further in another.

Builder is a flow control facility that enables users to record operations from the DECdecision components, including their interactions. These recordings are called "blueprints" and can be played back to automate a desired task.

Builder uses a tape recorder paradigm as a model for describing component interaction. The user turns on Builder to start a task, and components are transparently "recorded" as they are used. As components are recorded, the Builder diagram is updated to reflect the task.

Boxes within the diagram represent component instances, and arrows, or data links, represent the interaction between applications.

Other Builder features include a debug mode (e.g., single stepping through playback), and a splice mode, which combines record and playback capabilities for modifying a blueprint.

How Builder Works

The Builder component is divided into several pieces, including the user interface, a session manager, and a run-time library component interface.

The session manager is a separate code layer that allows the user interface to control a session and the components participating within that session. This control includes the ability to start, stop, and pause a session.

When a component starts, it registers itself with Builder. This registration enables Builder to determine, at any point, which components understand Builder. If a record or playback session is active, the component may be instructed to join the session. If a session is not active, this registry leaves enough information for Builder to contact the component if a session is started.

While a session is active, Builder communicates with components and instructs and coordinates their activities. A component may be instructed to process commands directly from the user, from Builder, or both. A component calls Builder when there is an operation to be recorded. Builder contacts the component to play back an operation.

Since Builder and the other DECdecision components are in separate processes, Builder does not directly call components. Actually, most of the record and playback logic is distributed between the Builder process and the Builder run-time library. Components link with the Builder run-time library, which does the actual callbacks to the component. The two Builder interfaces conduct a communication dialog, when necessary.

For additional information, the Interapplication Access and Integration paper in this issue explores the design, architecture, and communication of Builder in more detail.

Record Mode

A DECdecision component joins the session when it is notified that a record mode session is active. As part of the join session process, the component tells Builder its name, invocation string (i.e., how to run the component), and the native language it understands (e.g., French). These component attributes are saved with the blueprint, and are used to identify the component and activate it during playback.

When a component joins a record session, Builder draws a box within the its diagram region to represent the component. This region is highlighted to show that it is active. Builder also allows the user to incorporate a component into a record session, even if the session was started *after* the component started. This late session entry is called post start-up record mode.

As the component is used, it translates the operations into command strings which are passed to Builder for recording. Builder attaches sequence numbers to the command strings to order these operations not just within the given component but across components.

Builder never interprets a command string because the syntax of commands is private to a

component. However, Builder needs the information that certain commands have occurred. The component flags these commands to Builder. Examples of these command types are:

- Open and close (e.g., file open/close)
- Cut-and-paste (i.e., clipboard and QuickCopy)
- Exit (e.g., file quit/exit)

Cut-and-paste commands are flagged to enable Builder to know which components are exchanging information. Builder needs this information to draw data link arrows within its diagram region. Information about the other commands is used primarily for user feedback, when Builder displays commands through its command browser and editor.

When a component is finished, it notifies Builder and exits the session. After doing some additional bookkeeping, Builder removes the highlight for the component from the appropriate box within Builder's diagram region. The recording process continues until the user stops the record session. At this point, Builder merges all operations into a single master blueprint file.

Playback Mode

During playback, Builder invokes applications with instructions to execute operations in the same order in which they were recorded.

After an application is started and registered, the Builder registry code determines what session to join and what set of commands Builder expects the component to execute. The component joins the playback session and waits for further instructions. At this point, Builder highlights the component box associated with the component to indicate that it is active.

Part of the join session procedure includes an execution registry callback. Builder calls this routine when there is a command for the component to execute. The Builder run-time library, which is linked with each component, contains playback logic, which determines when a command is executable. Since a blueprint may include operations recorded across several simultaneously active components, the playback logic synchronizes command execution across the components.

When a component's command can be executed, the playback logic calls back to the component to execute the command. The component parses and processes the command using the syntax that was recorded.

If an error is detected during playback, the component notifies Builder and the user. Builder immediately pauses the session, highlights the component box of concern, and notifies the user. The user is instructed to resolve the problem. When the problem is resolved, the session's playback resumes.

The playback process continues until a quit operation is processed by the component, in which case, it exits the session. Builder removes the highlight from the component box. If this is the final component, the playback session is stopped.

During playback, Builder supplies components with a private global select type for use in QuickCopy operations. This guarantees that these QuickCopy operations will not conflict with either normal QuickCopy operations or those QuickCopy operations in process in other playback sessions. Builder also reuses these atoms during subsequent playback sessions, to reduce the resource demand on the DECwindows server.

DECdecision Control Component

The DECdecision Control component provides a single, central point from which all the DECdecision components can be invoked, and a central point from which common customization features can be controlled. The command line syntax used to invoke the Control component itself is inherited by any of the components invoked by this component. Using the same line syntax maintains a level of consistency at the command line interpreter level across all components.

In the DECdecision product each component invoked by the Control component can place itself into a re-usable state after the component session terminates. Once a component is in this re-usable state, it can be restarted much faster and with less use of system resources than the first time the component was invoked. The Control component manages the components in their active and re-usable states. This management is achieved through a combination of AIL for component invocation and control, and an internal list of components and their current state. The Control component does not place itself into a re-usable state since it always takes on the role of the invoker (i.e., parent component) instead of the invokee (i.e., child component).

For a component to enter this re-usable state, its normal image exit sequence has to be modified. First, the component releases whatever resources that the current session has acquired (e.g., widgets, windows, memory blocks, dynamic strings) and

hides its user interface to give the appearance of a normal exit. Second, the component purges its working set back to the working set quota. Purging releases stale data and code pages from the working set. (Note: Neither of these steps is required for component re-usability. The steps minimize system resources demand, which allows these resources to be available to other processes on the system.)

The component next communicates with the invoker (parent image or Control component) that the child session is complete. Accompanying the message is an implied question of whether the child should terminate or enter the re-usable state. The component then establishes a time-out period and waits for the termination or restart message from the parent. If no message is received, the component defaults to a re-usable state. The time-out period ensures that the component does not stay in the reusable state for an indeterminate period of time. When either the termination message is received or the time-out is triggered, the component performs the remainder of its image exit sequence. If a restart message is received, the component reinitializes and becomes active.

The benefit of re-usability is quicker start-up time on subsequent uses of the component. The second initialization of a component does not require the entire initialization sequence to be processed. Many operations performed during start-up need only be done once. For example, the overhead of image activation is eliminated. Similarly, the connection from the client to the server and the associated overhead of DECwindows initialization, such as the loading of the resource database, is only performed once. Each component also saves component-specific initialization data. In Calc, loading the section file is a once-only operation. In Access, attaching to the database and opening of the folder of tables is performed only once.

Summary

The DECdecision team met its design goal to deliver an end-user decision support application based upon DECwindows and the CDA architecture. It met its schedule goal of release near the time of the DECwindows release. It met both these goals by extending existing Digital products where possible and by developing new applications and utilities where necessary. As such, the DECdecision product provides a level of sophistication and seamless data integration not found in many products. The design and development of the DECdecision product serve as models for future applications development using the features of DECwindows and the Compound Document Architecture.

Acknowledgments

The authors wish to acknowledge Peter Savage and Peter Bower for their technical contributions. They would also like to acknowledge and thank each DECdecision developer for the hard work and long hours spent on this project.

References

1. B. Cheung and N. Jacobson "Interapplication Access and Integration," *Digital Technical Journal,* vol. 2, no. 1 (Winter 1990, this issue): 49–59.
2. R. Travis, "CDA Overview," *Digital Technical Journal,* vol. 2, no. 1 (Winter 1990, this issue): 8–15.
3. *CDA Reference Manual,* vols. 1 and 2 (Maynard: Digital Equipment Corporation, Order Nos. AA-PABUA-TE and AA-PABVA-TE, 1989).
4. *XUI Style Guide* (Maynard: Digital Equipment Corporation, Order No. AA-MG20A-TE, December, 1988).
5. S. Cohen and W. E. Morgan, "The Relationship between the DECwrite Editor and the Digital Document Interchange Format," *Digital Technical Journal,* vol. 2, no. 1 (Winter 1990, this issue): 73–82.
6. C. Young and N. Jacobson, "The Digital Table Interchange Format," *Digital Technical Journal,* vol. 2, no. 1 (Winter 1990, this issue): 28–37.
7. W. Laurune and R. Travis, "The Digital Document Interchange Format," *Digital Technical Journal,* vol. 2, no. 1 (Winter 1990, this issue): 16–27.
8. M. Jack and R. Gumbel, "The CDA Toolkit," *Digital Technical Journal,* vol. 2, no. 1 (Winter 1990, this issue): 38–48.
9. *DECdecision Calc Macro Guide* (Maynard: Digital Equipment Corporation, Order No.: AA-ML44A-TE, July, 1989).
10. *VAX Text Processing Utility Manual* (Maynard: Digital Equipment Corporation, Order No.: AA-LA14B-TE, June, 1989).
11. *VAX EDT Reference Manual* (Maynard: Digital Equipment Corporation, Order No.: AA-LA16A-TE, April, 1988).

Seth S. Cohen
Wm. Eugene Morgan

The Relationship between the DECwrite Editor and the Digital Document Interchange Format

The DECwrite editor is Digital's new DECwindows-based compound document editor. It is also the first compound document editor to implement the CDA architecture. The DECwrite editor supports the creation, editing, formatting, and printing of compound documents across multiple computing environments. DECwrite uses the DDIF document interchange format to support the editing of both CDA documents and those based on other formats, including SGML and GKS. One of the design issues faced by the DECwrite editor was how to fully conform to the DDIF format's interchange goals without compromising formatting speed and ease of editing. The DECwrite editor overcomes these conflicting needs by isolating their side effects to the DECwrite editor's read and write code.

Businesses are learning that documents are rarely created and controlled by one author or even one group. Information is contributed, shared, accessed, and revised by multiple users. Organizations must be able to create documents that combine information, in forms such as text, graphics, images, and spreadsheets. These documents must be kept up-to-date with the most current information until the moment of distribution.

Together, the CDA architecture and DECwrite editor meet these needs. The CDA architecture is an open, integrated architecture that facilitates creating, publishing, storing, and retrieving compound documents throughout a networked, heterogeneous computing environment. The DECwrite editor is a compound document editor that allows businesses to create and edit documents in this environment.

In designing a compound document editor to support the CDA architecture, a choice had to be made. We could work with an independent software vendor and have it adapt an existing product to support CDA documents, or we could develop a new product for this purpose.

The decision to develop a new compound document editor was based on several factors. The CDA architecture and the DECwindows architecture are Digital-developed architectures. In many ways, it was easier to develop a new product in conjunction with developing these architectures than it would have been to coordinate development with an independent software vendor. Second, such an editing system could include more features that directly supported the CDA architecture. Third, since the CDA architecture is an open architecture, an editing system developed for that architecture can be more easily extended as the architecture is extended.

The DECwrite editor combines "what you see is what you get," or WYSIWYG, text processing with graphics creation. The DECwrite editor runs under the DECwindows graphic user interface, on workstations running either the VMS or ULTRIX operating system. The DECwrite editor supports keyboard interfaces for the WPS-PLUS, EDT, EVE, and EMACS editors, as well as its own keyboard interface.

In addition, through the LiveLink feature, the DECwrite editor permits users to incorporate business graphics and data from other supported applications directly into documents. The LiveLink connection ensures that documents contain the most timely and accurate information. It also allows the user to update the data while they are editing the document.

This paper focuses on the relationship between the DECwrite editor and the DDIF document interchange format.[1] DECwrite uses the DDIF format as its file format. The DDIF format is suitable for several reasons. First, it is interchangeable. For example, the same DDIF file can be read on a VMS system, VAX ULTRIX system, or RISC ULTRIX system. Second, the DDIF format provides a way to represent all the

features that the DECwrite editor offers. The DDIF format can represent all of the document's raw content, as well as all the descriptive information needed to tell a formatter how to visually present the document. For example, the DDIF format stores with each paragraph whether it should be presented centered, left-aligned, right-aligned, or justified. Because the DDIF format is extensible, we can ensure that it supports new DECwrite features in the future. Finally, the CDA converter architecture supports the translation of documents to and from the DDIF format. Thus, the DECwrite editor can work with any format that can be converted to and from the DDIF format.

When a user running the DECwrite editor wants to edit a document, the open function is used. This function causes DECwrite's read code to read the DDIF file containing the document into DECwrite's address space. We call the DDIF file our on-disk document format. We call the result after reading it our in-memory document format. The job of the read code is to convert the interchangeable on-disk representation of a document into the editable in-memory representation of the document.

Conversely, the save function is used when a user is finished editing a document. This function causes DECwrite's write code to convert the editable in-memory representation back to the DDIF representation and write the result to disk. Therefore, using the read and write codes means the bulk of the DECwrite editor does not need any knowledge of the DDIF format. (See Figure 1.)

Because of their difference in goals, interchangeability versus editability, the DDIF format and the in-memory format differ in some ways. Data on disk only needs to describe *what* is in the document, whereas data in memory also needs to tell an application *how* to process what is in the document. An on disk format is optimized for interchange. It represents data in a system-independent fashion and has an explicit self-describing structure. An in-memory format is optimized for editability to provide good performance and easy data manipulation. For example, system-dependent data is preferred in memory because less system time is required to access system-dependent data. These differences account for the general activities required to convert the DDIF format into the in-memory format, and vice versa.

The next section of the paper summarizes the general activities required during conversion. The remainder of the paper describes the details of converting the major entities in a document: text, elements, objects, layout, LiveLink connections, and styles.

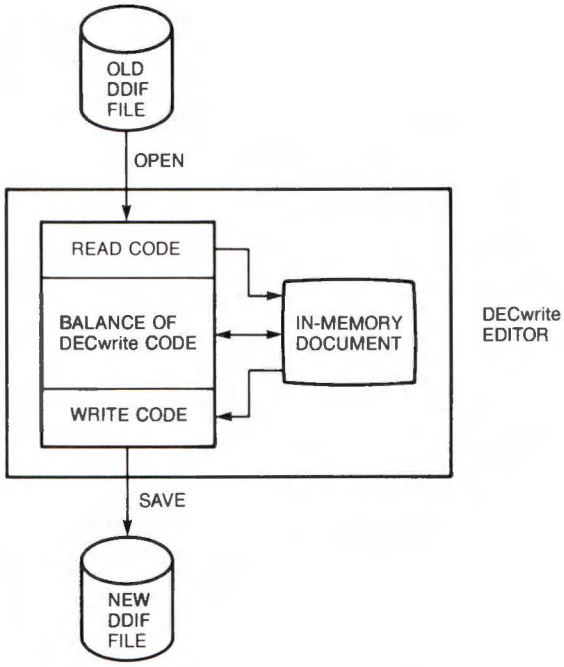


Figure 1 DECwrite Document Processing

General Conversion Activities

After a document is read into memory, it contains most of the same data items as it did on disk. For example, a graphic object has the same attributes on disk and in memory, such as fill pattern, outline width, and color. However, a simple mechanical conversion still has to occur as all data items are read or written, because the data on disk is in tag-length-value (TLV) octets, whereas the data in memory is a collection of binary structures that are interconnected by address pointers. The DDIF format uses the TLV coding format to permit binary data to be represented in a system-independent form. The TLV format also allows upwardly compatible extensions to be made to the DDIF format. The TLV format is not suitable in-memory because it is not compact and it is slower to access.

Some of the other general conversion activities also relate to performance and easy manipulation of the data.

- Changing how structure is represented. In the DDIF format, structure is explicit. For example,

a subsection segment is physically embedded with the segment of its section. In memory, the DECwrite editor often represents structure as an attribute value. This representation simplifies the code that allows users to edit structure. For example, if a user wants to change part of a list into a sublist, the use of an attribute value makes this easier to manage.

- Changing how attributes are organized. In the DDIF format, an embedded segment may inherit some of its attributes from the outer segment when an attribute of the outer segment is not present on the inner segment. For example, if a bold piece of text contains some bold italic text, the DECwrite editor represents this in the DDIF format as bold on the outer segment and italic on the inner segment. However, if there were plain italic text embedded in the bold text, the inner segment would have an explicit no-bold attribute to override the setting on the outer segment.

  The DECwrite write code uses inheritance as much as possible to reduce the size of the DDIF files it creates. For in-memory data, all attributes are always explicitly present with each piece of content. This simplifies the data structures needed to represent the attributes and data structures needed to represent the attributes and reduces the system time required to access them.

- Changing how logical connections are represented. In memory, the relationships among objects are represented almost exclusively by address pointers. This representation is the most efficient mechanism for navigating document data structures. The alternatives to using address pointers to represent logical structures are physical adjacency of objects, which requires much shifting of objects during editing; or tagging each object with a unique label, which requires significant overhead to maintain the currency of labels and references and perform lookups. On the other hand, these addresses cannot be stored on disk, because address pointers are system-dependent. Therefore, in the DDIF format, relationships between objects are maintained in one of two ways. Relationships can be either structural (i.e., nesting of elements or physical adjacency) or symbolic (i.e., referencing an object by name).

- Creating symbol tables for user-specified names. The user can create various named entities in the DECwrite editor, such as the document's styles and reference identifications. In memory, the DECwrite editor maintains symbol tables, represented as binary trees, in order to quickly find entities from their names. Because the on-disk document is used for storage only, the need for fast access to named entities does not exist. Thus, these symbol tables are not written from the document when is returned to the on-disk state.

The other general conversion activities required were:

- Binding to external resources
- Processing non-DECwrite constructs
- Reading old documents
- Preserving the current formatting of the document

Binding is the process of accessing an actual entity. For example, to bind to a style in a style file, the DECwrite editor must read the styles from the document's style file and then access the desired style from the style symbol table so created. When a document is read into memory, it is necessary to bind to all referenced entities so that the document can be displayed and revised. For example, on disk, it is of no immediate consequent if a paragraph refers to a style that does not exist. In memory, however, the absence of the style becomes critical. It is the style's attributes that allow the DECwrite editor to display the document with the proper formatting. When the DECwrite editor cannot bind to the style, it must use default formatting for the paragraph in question.

Processing the DDIF constructs is generally not a problem for the DECwrite editor. However, there are three cases in which the editor does not fully understand a construct in a DDIF file.

The first is unsupported constructs, which are DDIF constructs that have no analog in the DECwrite editor, e.g., bezier curves or markers as line attributes. If it can, the DECwrite editor uses a fallback representation for the construct. For example, DECwrite version 1 does not support color processing. When the DECwrite editor reads a color, it converts the color to the gray-scale fill pattern that the color would have on a black and white TV screen. If it cannot generate a fallback for a construct, the data is lost.

The second is private data, which is content that is meaningful to a single application (or a number of cooperating applications). For example, on disk, an application might store a where-it-left-off value. The next time the application is started, it can re-establish the left-off state for the user. In DECwrite version 1, private data is lost. The third case is a

supported construct in an unexpected context, e.g., text attributes on an object. If the construct makes no sense at all, as with text attributes on an object, the editor discards the data. Otherwise, the DECwrite editor creates a suitable context. For example, if a polyline object is encountered in the middle of a text segment, the DECwrite editor creates and inserts a default floating frame to hold the object.

Although the DDIF format and DECwrite editor are enhanced from time-to-time, the DECwrite editor must always be able to read old documents. To do this, the system writes out a version number in the DDIF file. The read code checks the version number of the file and, when necessary, does special processing to correctly read obsolete constructs.

The DECwrite editor saves the current formatting of the document for two reasons:

- Performance. When a document is opened, the DECwrite editor checks if it created the document. If the editor did, the incoming data is already properly formatted. Since formatting an entire document takes much system time, this check speeds the process of opening a document.
- Maximizing interchangeability. One of the CDA architecture's goals is to be able to print and display revisable documents from different formatting products. Because it is not technically feasible for a viewer to replicate the formatting decisions of an arbitrary formatter, those decisions must be made available to the viewer. Thus, as a CDA architecture-compliant formatter, the DECwrite editor must write out its formatting decisions when it creates a DDIF file. These formatting decisions are captured in the DDIF file as soft line breaks, galley breaks, and word hyphenations.

Processing Text

The DECwrite editor reads and writes text a line at a time. If the line just contains text, there are only two points to note. First, the read code allocates extra memory per line to efficiently process document changes. Second, the read code computes the vertical position of a line of text because this information is not stored in the DDIF file.

When a line of text contains control information, additional conversion activities must be performed. A line of text may contain the following types of control information:

- Text attributes
- Font bindings
- Language bindings
- Character sets
- Embedded markers

Text Attributes

Text attributes specify the appearance of text, such as whether it is bold, italic, or underlined.

When the DECwrite editor writes a document, lines of text are separated into pieces that share common text attributes. One DDIF segment is generated for each such piece. Also the DECwrite write code uses inheritance to minimize the number of attribute changes and the number of segments that need to be written.

As a document is read, all inherited attributes are made explicit. The end result is that each piece of text on a line is stored with all its attributes. Forcing fully explicit attributes serves two purposes. It facilitates the display of the text pieces, because their font and rendering attributes are self-contained. Also, the attributes of a text piece can be maintained during a cut-and-paste operation.

Font Bindings

The first and simpler aspect of font binding is the correspondence table that the editor must place at the beginning of a DDIF file. This table allows individual font references to be compactly represented as indices into the table. It also declares what fonts are used in a document.

When writing a document, the DECwrite editor builds the table and generates reference indices from the font names in that table. When reading a document, the process is reversed. The DECwrite editor derives a reference's font family by accessing a font name from the table index. The read code normally requires only the font family, because the write code saves text attributes explicitly rather than as part of the font name. This approach is required to keep attributes independent of one another, and is discussed further in the Processing Styles section.

The second and more complex aspect of font binding relates to reading non-DECwrite documents. The complexity arises as a result of the use of wildcard characters in a font name. A font name consists of several fields, and a field may be a wildcard character. Wildcards exist because some fields are device-dependent. Since it would be inappropriate to store device-dependent data in a DDIF file, applications that create DDIF files must place the wildcard character in each device-dependent field. Unfortunately, different products do not use wildcards in the same fields. Thus, the DECwrite editor must match a font name against its list of known fonts by doing

field-by-field wildcard matching. If the specified font cannot be found, the editor uses its default font.

A font is an external resource. To reduce startup time, the DECwrite editor does not try to locate the actual font until the page is displayed. For example, if a document references 30 fonts, but its first page only references 7 of them, this approach reduces start-up time by 23 font bindings.

Language Bindings

As with font bindings, a correspondence table is used to refer to languages. Since the DDIF format and the DECwrite editor allow language to be changed at the word level, the language table also aids in achieving compact DDIF files. Matching language names in the table is simple because the names that may appear in the table are governed by ISO standards 639 and 3116.

Character Sets

DECwrite version 1 does not support arbitrary character sets. The primary character set it supports is ISO Latin 1. Users may also enter certain special characters that are non-ISO Latin 1 characters. When the user enters such characters into a document, the DECwrite write code enters the proper character set change into the text attributes for the segment.

When non-ISO Latin 1 text is read into the DECwrite editor, the editor checks the character set and character codes to determine if any of them are special characters known to the editor. If the characters are known, special memory codes exist for them in the system, and the character codes are mapped to those codes. If the character set and codes are not known, the DECwrite editor stores the codes as if they were from ISO Latin 1 and preserves their character set identification. Even though the text will be imaged incorrectly, its character set is preserved when the text is written back to the DDIF format.

Embedded Markers

In-memory text can contain markers for floating frames, cross-references, index items, and running values. A marker is treated in a similar manner to text. The marker is stored in a line of text along with normal text. Markers are used in memory because it is unwieldly to manipulate in-line representations of entire entities.

The types of markers and their special processing needs are described below.

- Floating frame. This marker points to a frame object and serves as the anchor point for the frame within the flow. The frame object is kept in a chain of floating frames, and this chain is associated with the text block in which the frame resides. When the DECwrite editor writes the document, the frame object is written as a DDIF segment in the text stream where its in-memory marker is located. When the DECwrite editor reads the document, the frame is created and chained to the current text block. The marker to maintain the frame's position is created and inserted in the text stream.

- Running value. On disk, a running value is represented as a segment that contains a reference to a variable or counter. The definition of the variable or counter resides with the referenced entity. For example, each segment that represents a section will have variables declared for the title's name, number, and page number. Also the current value of the entity is stored as the content of the reference's segment. A running value contains a variable reference as well as a current value to facilitate interchange. The variable tells other applications what the current value represents, and therefore how to recompute the running value when needed.

  The in-memory marker contains a field that indicates the running value type, e.g., current page number or current title name. Because the value of the entity depends on context, the value is always dynamically computed rather than pointed to by the marker.

- Index item. The marker for an index item points to a specific entry in the symbol table for the entire index hierarchy. On disk, an index item is represented as a DDIF function link. The read code is responsible for incrementally rebuilding the symbol table (as each index function link is encountered) each time the document is read.

- Cross-reference. The marker for a cross-reference contains a reference type (e.g., a title or footnote) and a pointer into a symbol table of reference identifications. On writing the DDIF document, the DECwrite editor stores the reference as a DDIF variable reference to a named segment. The cross-reference's reference identification is used as the segment identification of this named segment. On read, the DECwrite editor makes the appropriate symbol table entry when it encounters either the referenced object, which has a segment identification, or the reference. When the other part of the reference is

encountered, the symbol table connections are completed. More information about the handling of cross-references is provided in the Processing Elements section.

- Footnote reference. The marker for a footnote reference is identical to a cross-reference marker. The DDIF representation for the reference is also the same, but the footnote itself immediately follows the first reference to it on a given page. The handling of footnotes is described in the Processing Elements section.

Processing Elements

The DECwrite editor supports a number of structural document elements. In memory, these elements are represented as a sequence of data structures, and pointers are used to maintain the relationships among the data structures. For example, a list is composed of a sequence of paragraphs, each of which, in turn, is composed of a sequence of lines of text. To capture this hierarchy, the list points to its first and last paragraphs, and each paragraph points to its first and last lines. Each data structure also has a back-pointer to its encompassing data structure. In this example, the paragraphs would have a back-pointer to the list. In the DDIF format, these elements are represented as segments. Each segment has a segment tag that indicates the type of element it represents. The hierarchy of elements is captured by the nesting of the segments.

When writing a document, the DECwrite editor checks each line as it writes the line to determine if the line is in a different DDIF element than the previous line. If it is, the segment for the previous element is ended and a new segment is started. Starting and ending of nested elements is handled by the DECwrite editor in an analogous fashion. For example, if a paragraph is in a list, and the previous paragraph is either not in a list or in a different list (or other type of element), then a new segment must be started for that list. The previous segment is ended, if necessary.

The element types supported in this fashion are paragraphs, section heads, lists, footnotes, tables of contents, and indices. The DECwrite editor also has a cross-reference element that supports cross-references to list items, section heads, and footnotes. These element types, and how the DECwrite editor represents them on disk and in memory are discussed below.

Lists and Section Heads

To the DDIF format, lists and sections are explicit hierarchies. A section contains subsections. A list contains list items which in turn contain paragraphs. Paragraphs are represented by nested segments. For example, a section containing a subsection would look like the following:

<Start Section>

Content if any

<Start Section>

Content if any

<End Section>

<End Section>

For in-memory data, the hierarchy level for a list and section is encoded as an attribute on the paragraphs within the list and on the title within a section. Because of this difference between in-memory and on-disk data, the DECwrite editor must analyze the paragraph attributes it is writing and build the appropriate hierarchy. When a document is read, the reverse process occurs. The explicit hierarchy is converted back to attributes.

Cross-references

There are two sides to a cross-reference: the site of the reference and the construct being referenced. These two sides are connected by the reference identification, which is the symbol created by the user to uniquely identify this cross-reference. To process reference identifications efficiently, the DECwrite read code inserts each reference into a symbol table.

The references and the construct can appear in any order in a document. It is also possible to have a construct that has no references, and vice versa.

When the first reference identification appears, the editor makes the proper symbol table entry. The symbol table connections are completed when the other half of the cross-reference is read.

The current value of the construct is stored in the symbol table entry. If the referenced construct appears first, the DECwrite read code sets the current value of the cross-reference from the current value of the construct. If a reference appears first, the same rule can be applied because the DECwrite write code always stores the current value of the reference at each reference site. However, there are some interesting subtleties related to this situation when the referenced construct is in another

document. This is possible if both documents are part of a larger document.

- If the larger document is not opened at the same time as the smaller document, the value stored at the reference site is the DECwrite editor's only clue to the reference's current value. (Note: This point also applies to numbered section heads that are at the start of a linked-to document.)
- If the larger document is opened at the same time as the smaller document, then the DECwrite editor sets the current value of the reference again when the construct is seen. The DECwrite editor does this because the reference site can be out-of-date if the construct's document was saved since the last time the larger document was saved.

Footnotes

Footnotes in the DECwrite editor always appear at the bottom of the page containing the footnote reference. A footnote text block is placed on each such page to hold the footnotes. Footnotes and their contents are linked by the editor to this text block. The relationship between a footnote and its reference is treated the same as other cross-references. A footnote has a symbolic identification that is used as the key in a symbol table. This connection ensures that the footnote moves with the reference in cut-and-paste operations, as well as from page-to-page during text editing operations.

In the DDIF format, the footnote text blocks are written as part of the document layout. The footnote is written in-line, immediately following the first reference to it. The connection between the footnote and the in-memory text block is maintained by tagging the footnote segment and the text block with a stream tag that indicates that each of these holds footnote content. The stream tag also separates the footnote from the surrounding document content. This separation enables an application that cannot handle layout to still recognize and handle footnotes when it reads the document.

On read, the DECwrite editor collects and inserts the footnote content into the current page's footnote text block. It then links the footnote to the reference that immediately preceded it. The segment identification provides the linkage. If a footnote does not have a segment identification, or if there is no reference to the segment, the DECwrite editor automatically creates a footnote reference. As long as a footnote in a DDIF file contains the proper stream tag, the DECwrite editor recognizes the footnote and creates all the requisite supporting structures, including the reference.

Tables of Contents and Indices

Indices and table of contents are generated elements. The user tells the DECwrite editor what data belongs in them and how they should look. The DECwrite editor then builds them by scanning the document. For a table of contents, it scans title elements. For an index, it scans index items. For example, for each title in a table of contents, the user can tell the DECwrite editor to generate one or more of the title's name, label, page number, or any combination of these three.

The DECwrite editor builds a table of contents or an index out of paragraphs. Thus, reading and writing a table of contents or index is similar to reading and writing regular paragraphs in a document with two exceptions:

- On disk, the paragraphs are nested within a segment that identifies the whole collection of paragraphs as a table of contents or index. In memory, the first and last paragraphs of an index are pointed to by an index or table of contents element.
- The paragraph styles used by the DECwrite editor in a table of contents or index are not the standard user-created paragraph styles. Therefore, they cannot be stored with the standard styles. On disk, the nonstandard paragraph styles are nested within the segment of the table of contents or index. In memory, they are represented as local attributes on each paragraph.

The paragraphs the DECwrite editor places in a table of contents do not contain text. Rather than simply copying the data for building a table of contents entry, the DECwrite editor inserts cross-references to the title's name, the title's label, and the title's page number. These cross-references keep the table of contents entries up-to-date while the user is editing the document.

Processing Objects

The DECwrite editor supports various kinds of objects: geometric shapes (e.g., circles and lines), graphic text, layout elements (e.g., text block and frame), and LiveLink data blocks.

Objects participate in two kinds of connections, groups and frames. A user can create compound objects, which are called groups. A group is itself an object and may consist of any number of objects and nested groups. A user can create a coordinate space and clipping boundary, which is called a frame. A user can then put any number of objects within the frame. If the user moves the frame, all

the objects move as well. The order in which objects in a frame are drawn is determined by their creation order within the frame.

Because an object can belong to both a frame and a group, nesting alone does not suffice for representing the object on disk. Objects in a given frame are contained within the frame segment. The draw order is indicated by the order of the objects within the frame. All objects within a group have a special segment tag. This segment tag indicates that the object is in a group and in which group the object resides. The use of a segment tag eliminates the need to write any additional data for the group object, which acts as the parent for all the objects in a given group. However, if an object group becomes part of another object group, a new segment tag must be written. This segment tag contains a group tag that indicates both the parent group object and the group to which the parent group object belongs.

In memory, membership in both a frame and a group is easy to represent. First, the object is on the list owned by its frame. If the object is a member of a group, it is part of a similar list owned by the group object. Because a group is itself an object, the group will itself also be on the list owned by the frame.

The DECwrite editor processes graphics attributes in much the same way as it processes text attributes, which are discussed in the Processing Text section. In memory, a complete description of the attributes for each object is saved. The DECwrite write process uses inheritance to minimize the size of segments containing graphics.

Processing Layout

Layout and content are stored separately in the DDIF format. The DDIF layout design consists of a series of pages that contains the graphic and textual elements which describe the appearance of each page. Content can later be placed into that layout, in much the some way concrete is poured into a form. Thus a document's content and layout can be changed independently of each other.

Content is organized into flows. A flow is a logical stream of text and floating frames. Therefore, pouring content actually consists of one pouring action per flow.

In memory, the content is merged into the layout. This is what WYSIWYG means, enabling a user to see the content on the page where the user wants it to appear when the document is printed.

Page layout on disk directly corresponds to page layout in memory. However in memory, the layout of a page in a two-sided document can change if an earlier page is inserted or deleted. Therefore, in-memory representation of two-sided page styles must be such that it is easy to dynamically associate the proper side of the page style with each individual page.

In the DECwrite editor, the layout elements into which flows are poured are called text blocks. Text can flow across pages because the user can connect text blocks together. Text blocks directly correspond to galleys in the DDIF format.

When the DECwrite editor writes a document, it gives each text block a symbolic name. To connect one text block to another, the system symbolically declares the next text block as the successor for the previous text block. No successor is declared for a chain's final text block. To link a flow to a chain of text blocks, the name of the first text block is declared in the most outer segment of the flow.

When the DECwrite editor reads a document, it checks to see if each text block processed is the target of any successor links. If the text block has a successor link, the system checks that a successor exists. If this check finds a successor match, the system immediately connects the two text blocks. If a text block and a successor cannot be matched, the system enters the text block's name or the name of its successor, and the text block's address in a table for future checks. As the DECwrite editor encounters each flow, it checks the table to locate the first text block of the flow. If the system cannot find the flow in the table, or if the flow does not reference a text block, the DECwrite read code creates a default text block and a default page. In either case, the read code places all the content of the flow in the chain of text blocks. (Note: There is one limitation of this algorithm. On disk, the galleys of a flow must precede the flow. If not, the document will not be read correctly.)

Because connections are made as soon as both objects are in memory, the editor never needs to re-scan the table. The table deletes information as connections are made because text blocks do not have multiple predecessors or successors. Therefore, the table remains small and efficient.

As lines of text are placed into a text block, the DECwrite read code calculates the vertical position of each line. The horizontal and vertical positions of the frame are calculated when a floating frame is anchored to a line.

When writing a document, the DECwrite editor inserts a new galley directive after the last line in each text block. When the document is read, these directives ensure that lines of text return to the proper text blocks.

Processing LiveLink Connections

The DECwrite editor uses three kinds of LiveLink connections:

- Link to picture. The LiveLink function references a data file that is imaged on the page, either a DDIF file or an encapsulated PostScript file.
- Link to application. The LiveLink function references a data file and names an application. The application can be invoked to process the data and deliver DDIF content to the DECwrite editor for presentation on the page.
- Link to document. The LiveLink function references a document, whose pages are merged with those of the referencing document. This linkage permits a document to be compounded from a number of smaller documents.

Except for links to documents, in-memory LiveLink connections are represented as data block objects. The data block identifies the data file, the application if any, and the position and size of the final result. On disk, the data block is represented as a DDIF segment that contains frame parameters (for size and position information) and a function link to hold the file and application names.

In memory, a link to document is represented in a subdocument structure that holds all the pages of the subdocument. All subdocument pages and the pages of the root document are connected into a single chain. Their primary back-pointer is to the root document. This arrangement ensures that the documents appear as a single document to the user.

As a document is written, the DECwrite write code checks for any transitions from a root document to a subdocument. At each transition, the editor writes an external reference to name the subdocument. The external reference location indicates the subdocument's proper location in the root document. After writing the external reference, the write code moves to the end of the subdocument and resumes writing the root document. When an external reference is encountered in the read code, the referenced file is read into a subdocument. A pointer to this subdocument is kept in a temporary structure within the read code, as well as the pages that mark where to insert the subdocument in the root document. When the new document is completed, the pages of all subdocuments are merged, and the flows between the documents are joined.

Binding to LiveLink Connections

LiveLink data is stored outside the document. The location of the data is specified by a file specification. On disk, this specification is simply a string. In memory, the DECwrite editor must identify the file behind the string. Finding this file usually involves transforming the file specification string because the DECwrite editor supports four file location types:

- With document. The file specification string is combined with the location of the document.
- Private library. The file specification string is combined with the current node.
- System library. The file specification string is combined with the current node's CDA system area.
- Network library. The file specification string is used as is.

As with fonts, the DECwrite editor does not process an external reference until necessary. This is done for performance reasons. In particular, a large amount of system time can be required to read an image. For picture links and image links, the link data is the data displayed on the page for the link. Such links require binding when the page is displayed or printed. For application links, the external reference is to the application's input data, not its display data. Application links display data is stored inside the document itself. Thus, binding only needs to occur if the user invokes the application and the DECwrite editor must pass the file to the application.

Links to Images

Although an image is stored and edited at its full resolution, it must be transformed to screen resolution to be displayed. Therefore, the DECwrite editor maintains two copies of each bound image: a revisable full-resolution copy and a screen-resolution display copy.

Processing Styles

Every element in a DECwrite document references a style. A style is a collection of attributes that govern the overall appearance of an element. In memory, the styles are collected into a table called a style catalog. On disk, each style is represented as a type definition within the root segment of the DDIF document.

The DECwrite editor supports nine element/style classes: paragraph, title, list, footnote, page, text block, frame, table of contents, and index.

Binding to Styles

Every style has a user-created name, but in memory the DECwrite editor uses a pointer rather than a name to connect an element to its style. In addition to elements pointing to styles, there are also cases where styles point to each other. For example, a title style can point to the page style that a title element must appear on.

When writing a style reference, the write code uses the pointer to locate the style name and that name is written to the DDIF file. Conversely the DECwrite read code uses the style name to find the style in the style symbol table and re-establish the pointer. If a style name is not found, the read code creates a style that uses the default attributes of the element's class.

Style Files

A style file is a document whose styles are referenced by a second document. As with LiveLink connections, the reference is a file specification. The specification must be transformed according to the file location type specified in the referencing document.

The DECwrite editor must read a document's style file before it can read the document's layout and content. If the specified file does not exist, the DECwrite editor displays a file selection box for the user to specify a new style file. This same process occurs if the system encounters a circular style file reference. A circular style file reference occurs when a style file refers to one of the files that was processed earlier in the reference chain.

Styles from a style file should not be saved with a document. Therefore, each style in memory identifies whether it is local to the document or was acquired from a style file.

Style Precedence

A document may contain both local styles and a reference to a style file. A local style takes precedence over a style of the same name from a style file. This style precedence allows users to tailor a generic document style for the current document.

This feature also creates the reverse possibility. A user must be able to change a style back to the generic document style. Therefore, the DECwrite read code stores both the local style and the style from the style file when both styles exist. In effect, the DECwrite editor stores the style file's style underneath the local style. If the user deletes the local style, the style file's style becomes visible again.

Local Style Attributes

A full copy of an element's style attributes is stored with the element in memory for the same reasons text attributes are stored. (See Processing Text section.) The mechanism by which this is done is also analogous. The attributes inherited from the element's style are combined with the local attributes of the element. An element's local attributes are those style attributes that the user has changed for that element alone.

When a document is read in, the DECwrite editor checks whether the document's style file has been modified since the last time the document was modified. If so, the editor reformats the document.

The DECwrite editor writes out attributes locally only when they differ from the attributes in the element's style. This process guarantees that all new attributes in the style file are not obscured by local attributes. Thus, when the user changes a style file, the desired result occurs for each element. No local attributes are lost, and all the element's other attributes are updated.

Summary

The goals of the DDIF format and a running application like the DECwrite editor are different. The DECwrite editor handles this difference by isolating the side effects to DECwrite's read and write codes. The preceding sections have illustrated how the DECwrite editor does this. We believe this approach has enabled the DECwrite editor to fully conform to the DDIF format's interchange goals without compromising formatting speed and ease of editing.

Reference

1. W. Laurune and R. Travis, "The Digital Document Interchange Format," *Digital Technical Journal,* vol. 2, no. 1 (Winter 1990, this issue): 16–27.

Neal B. Appel
Ronald M. Olson

CDA in Science and Engineering

The CDA architecture is being extended to support the specific requirements of the scientific and engineering communities. The DECview3D application is part of the CDA tools suite that enables science and engineering users to integrate two-dimensional and three-dimensional graphics into compound documents. Graphics can be translated into various formats, including the DDIF interchange format, and engineering and scientific data can be viewed and annotated.

The CDA architecture supports a revisable-form data interchange that is both open and extensible. A major benefit of this architectural design is that it can be extended to support specific industry and application needs, such as those within the science and engineering communities. The need for more sophisticated scientific and engineering electronic document processing and data interchange has grown increasingly in these communities in the last few years.

In response to this need, Digital's Engineering Systems Group (ESG) and CAD/CAM Technology Center (CTC) have jointly developed the DECview3D application.

The DECview3D application provides graphics translations, two-dimensional and three-dimensional graphics manipulation, and annotation of engineering and scientific data. The user can integrate this data into the CDA architecture. The application reads and translates engineering and scientific data files from many formats, including the Initial Graphics Exchange Specification (IGES), the Hewlett-Packard Graphics Language (HPGL), and other formats. Its IGES translator conforms to the MIL-D-28000 specification. MIL-D-28000 describes the IGES application subset of the Federal government's Computer-aided Acquisition and Logistics System (CALS).[1]

The DECview3D application includes a set of three-dimensional display and manipulation operations to mark up and prepare data for integration with other applications. These applications can then be used to create manufacturing process plans, engineering and scientific reports, and other technical compound documents. The DECview3D software operates as a standalone system or as a LiveLink application within the CDA architecture. With the DECview3D application, users can easily share and integrate engineering and scientific data files.

The DECview3D Application

To support engineering and scientific graphics, the CDA tools suite must provide for the inclusion of two-dimensional and three-dimensional graphics in compound documents. This support must include the translation of engineering and scientific graphics into various formats, including the DDIF document interchange format. The DECview3D application is the CDA tool that meets these needs. Further, the DECview3D application allows engineering and scientific data to be viewed and annotated, either by itself or in conjunction with a CDA compound document. Expensive computer-aided design (CAD) tools are not required.

The DECview3D application combines the functionality of VAXcadoc, VAXcadview, and BASEVIEW software with the benefits of the CDA architecture. VAXcadoc and VAXcadview are used within Digital to view and annotate engineering data; BASEVIEW software is Digital's VMS workstation software (VWS) product for viewing and annotating engineering data. The DECview3D application draws upon these for several useful functions:

- Translation of different engineering and scientific data formats, including IGES, into a single data structure
- The ability to view, zoom, pan, rotate, annotate, query, and print engineering and scientific data from different sources in a single, easy-to-use system that runs on both workstations and graphics terminals

- Inclusion of engineering and scientific graphics in technical compound documents through a LiveLink connection (See Figure 1.)

DECview3D Design Decisions

Our initial design decision was to build the DECview3D code on the existing VAXcadoc code. Not only did this give us an established body of code to work with, but it also provided components needed in the DECview3D code, including the following:

- Two-dimensional and three-dimensional graphics display, with zoom, pan, and rotation capabilities
- A run-time data structure for graphics (RDSG) that is extensible and has an archival format
- A device-independent, three-dimensional graphics subsystem composed of the Common Graphics Interface (CGI), which performs three-dimensional transformations and entity selection; and Digital's implementation of the Graphics Kernel System, the DEC GKS product, for two-dimensional, device-independent output[2]
- A common user interface (CUI) which allows the same user interface on a workstation windowing system and on graphics terminals that can run under the VMS screen management facility (SMG)[3]

Figure 2 illustrates the VAXcadoc architecture.

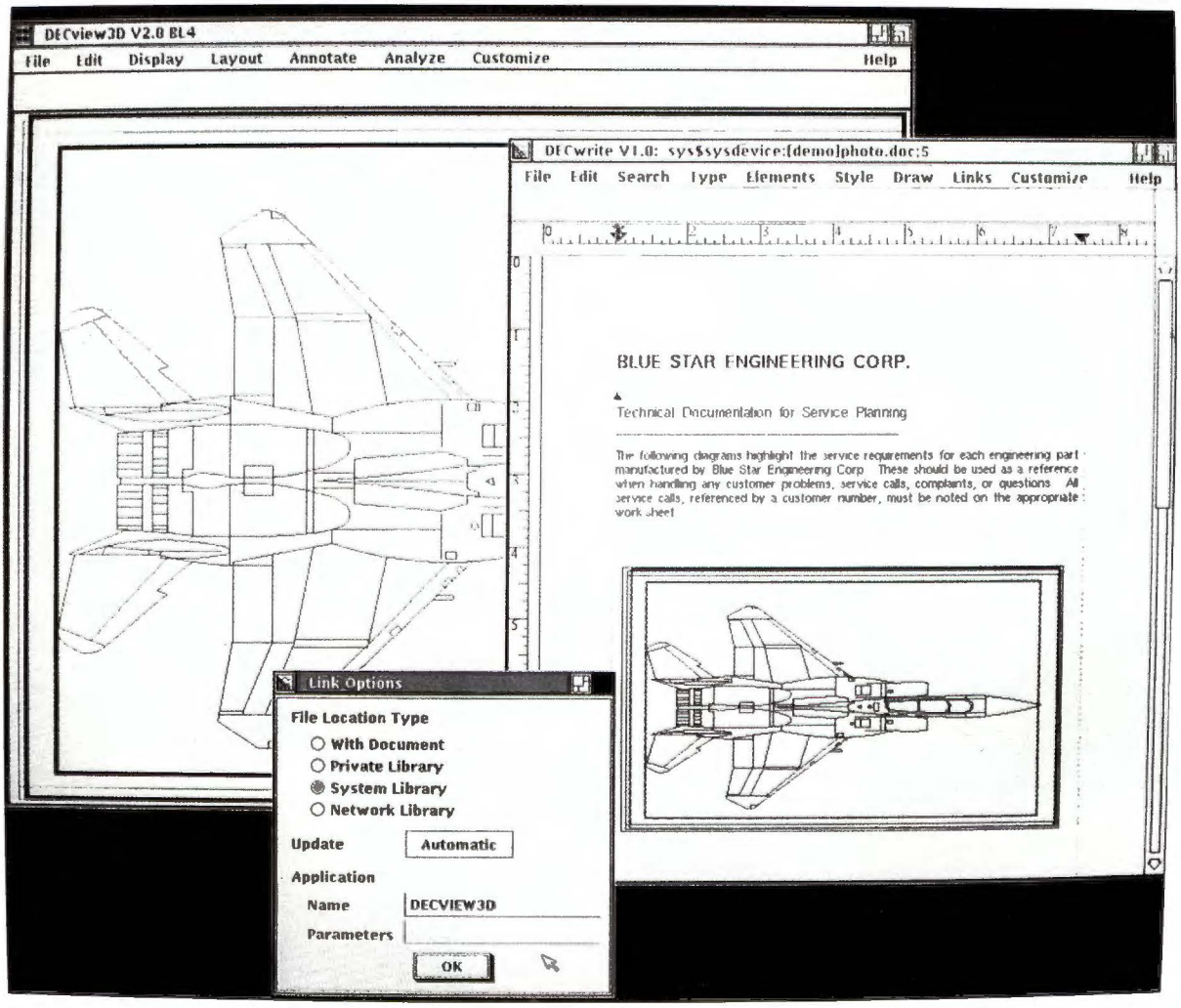


Figure 1 LiveLink Connection between DECview3D and DECwrite Software Products

The decision to use the VAXcadoc code set the stage for our subsequent design decisions. These decisions were based on major product goals:

- Inclusion of engineering and scientific graphics into compound documents created by the DECwrite editor using the LiveLink mechanism
- Translation of engineering and scientific graphics into numerous formats, including the DDIF document interchange format
- Support of non-DECwindows terminals for viewing and translating engineering and scientific graphics

The DECview3D application became a LiveLink child application to the DECwrite editor to support the inclusion of engineering and scientific graphics directly in CDA compound documents. Being a LiveLink child application means that, through LiveLink facilities, the DECwrite editor can directly invoke the DECview3D application to translate, view, and annotate engineering and scientific graphics. The user of the DECwrite editor indicates that a LiveLink connection with the DECview3D application is desired, and specifies an area within the DECwrite compound document in which to place the DECview3D graphics. The DECwrite editor then invokes the DECview3D application through the CDA application control services (ACS), thus bringing up the DECview3D user interface. When the user exits the DECview3D application, a DDIF data stream is returned for inclusion in the DECwrite compound document. The DECwrite editor automatically formats this data into the area the user has specified.

Translating engineering and scientific data into the DDIF format gives DECview3D the means to integrate three-dimensional graphics with the CDA architecture. The DECview3D application creates DDIF data through the DEC GKS product. The RDSG data is passed to CGI, which transforms the data into GKS format and then makes DEC GKS calls to create a DDIF format file. The first accomplishment after the DECview3D software was linked with DEC GKS version 4.0 was to create a DDIF format file of engineering data that could be read and displayed by the DECwrite editor.

Because of varying needs, users of the DECview3D application require a very flexible translator architecture. For example, some users have engineering graphics in proprietary data formats that they would like to bring into compound documents, whereas others have special-purpose output formats that they

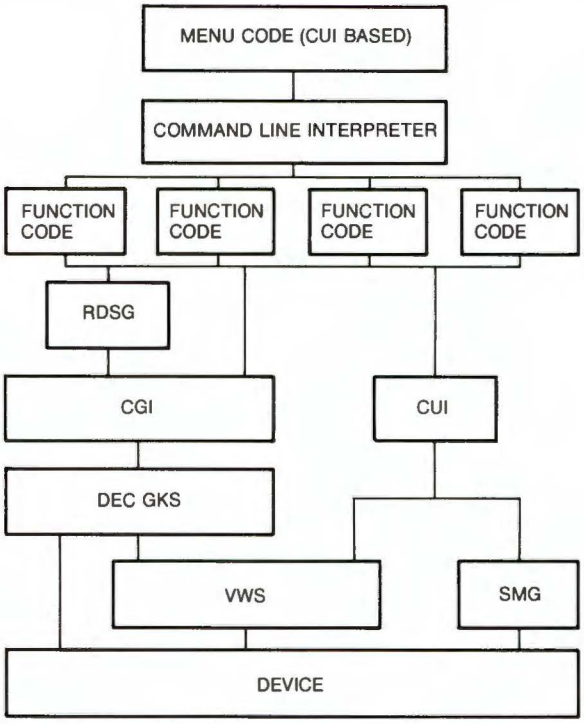


Figure 2 VAXcadoc Architecture

would like to create. To address these individual needs, the translator design allows input and output translators to be added to the product without relinking the DECview3D software.

To provide this flexibility, the DECview3D software uses VMS shareable images and a single-interface binding for both input and output translators. Any translator that is built as a shareable image and that conforms to the interface binding can run with the DECview3D application. Each input translator creates RDSG run-time data, and each output translator starts with the RDSG run-time data. Thus, mixing and matching of translators is possible. The DECview3D software allows the name of the input or output translator shareable image to be specified in the DECwindows user-interface definition (UID) file as one of the parameters of the callback routine for the menu event. As a result, any translator can be integrated into the product. Figure 3 shows the data flow and control flow for the translation process.

For the DECview3D application to work smoothly as the integrator of three-dimensional data from within the DECwrite editing environment, user interface compatibility had to be achieved. The

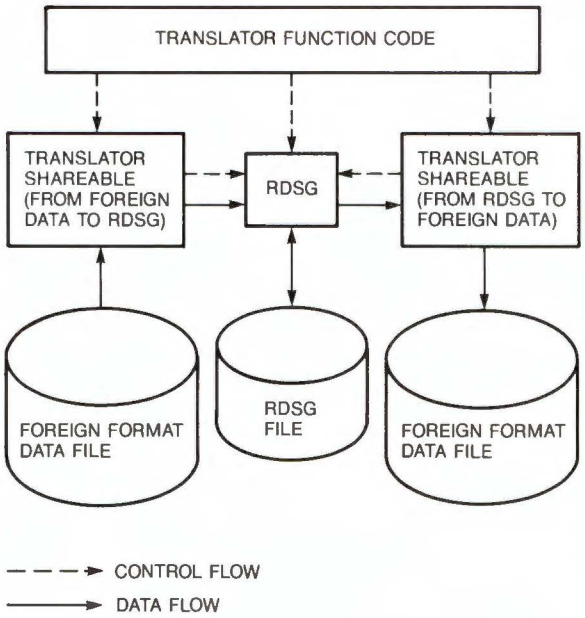


Figure 3 Data and Control Flow for Translation

DECwrite editor is a DECwindows application; the DECview3D application would run in conjunction with the DECwrite editor. Therefore, the DECview3D user interface had to conform to the *XUI Style Guide* and conform as close as possible to the DECwrite editor user interface as well.[4,5]

To conform with the *XUI Style Guide* and be compatible with the DECwrite editor, the DECview3D code had to support an object/action user-interaction style in which the user indicates the objects on which to act before indicating the action. The code also had to support an event-driven style in which each single user action is an event, and the application reacts appropriately to all events at any given time. In addition to these CDA requirements, the DECview3D application had to run not only on DECwindows devices but also on ReGIS-based graphics terminals such as the VT240 terminal.

Although the use of VAXcadoc code would expedite the DECview3D design process, it presented us with a problem in terms of the DECview3D interactive style as outlined above. VAXcadoc code implements an action/object user-interaction style in which the user indicates the desired action before selecting the objects on which to act. In addition, the code uses a request-driven style in which the application controls which single user-interaction may be processed at any give time. Also, VAXcadoc runs only on the VWS workstation windowing system, not on the DECwindows system.

We solved the conflict in styles by making an adaptation to the VAXcadoc code. In VAXcadoc, each function code makes its own entity selection in a request-driven manner. For DECview3D code, we separated the entity selection code from the function code and placed it in one location. The user-interaction style then became event driven, and the object/action style was possible. The function code in DECview3D software now receives identifiers of objects — mainly geometric entities — that have already been selected.

To make the change to an event-driven graphics interaction, the DECview3D software uses DEC GKS software in event mode on graphics terminals and the GKS widget on terminals using the DECwindows system. (Widget is a term used in relation to the DECwindows system to refer to user-interface items, such as pull-down menus, icons, and dialog boxes.) Thus the graphics user-interaction style is event-driven, and the amount of device-specific code in the graphics area is reduced to a minimum.

To complete the change to an event-driven and object/action user-interaction style, the DECview3D software uses a Digital internal user-interface utility (CIMI) for the user interface on graphics terminals. CIMI gives the DECview3D application the same window-oriented user interface and user-interaction style on both DECwindows devices and graphics terminals. CIMI supports a subset of DECwindows menu and dialog boxes on graphics terminals. It also emulates the DECwindows event-driven and object/action user-interaction style. How CIMI operates within the DECview3D application is discussed in more detail in the section CIMI User Interface.

DECview3D Architecture

The DECview3D architecture represents a combination of VAXcadoc components and new components. Each component discussed below refers to the block diagram in Figure 4.

DECview3D Initialization/Termination Code When the user invokes the DECview3D application either as a standalone application or through a LiveLink connection, control is first given to the DECview3D initialization/termination code. This code

- Initializes the application and session

- Shuts down the application and session
- Brings up and shuts down the user interface

When the DECview3D application is invoked by means of a LiveLink connection, the initialization/termination code also

- Processes all LiveLink connections and communication through application control services (ACS) routines
- Creates snapshots by producing two-dimensional projections of three-dimensional data in the DDIF data format for inclusion in a parent application, such as the DECwrite editor

CIMI User Interface CIMI allows applications to use the same user interface to run on both DECwindows devices and graphics terminals. When an application is running on a DECwindows device, CIMI passes all calls directly to the DECwindows interface. On a graphics terminal, CIMI uses SMG to emulate the DECwindows widgets. CIMI can also use the same UID file that the DECwindows system uses.

For graphics terminals support within the DECview3D application, modifications were made to both CIMI and the DEC GKS ReGIS device handler. These modifications provided a consistent user interface and smooth transition of cursor control from the CIMI widgets to and from the GKS graphics area.

Normally, both CIMI and DEC GKS control their own keyboard-based cursors, and each assigns its own meanings to keyboard input. In addition, GKS interacts directly with a graphics terminal for both input and output. With the DECview3D modifications, CIMI can now receive all keyboard input, even if GKS is running on the graphics terminal.

When the user moves the cursor to the GKS screen area, CIMI sends the keyboard input to the DEC GKS ReGIS device handler. This action allows CIMI to exercise complete control over the user input. In turn, CIMI's complete control makes the user interface consistent and smoothes the transition of cursor control from CIMI to GKS and back. While in the GKS screen area, the modified DEC GKS ReGIS device handler receives the keyboard input from CIMI. However, the device handler treats the data in the same fashion as it did before the device handler was modified.

General Widget Callbacks and Graphical Input Management Whenever the user initiates an event, CIMI or the DECwindows system calls the general widget callbacks or the graphical input management code. There is one logical interface for each event, although the same actual callback routine may handle a number of different events.

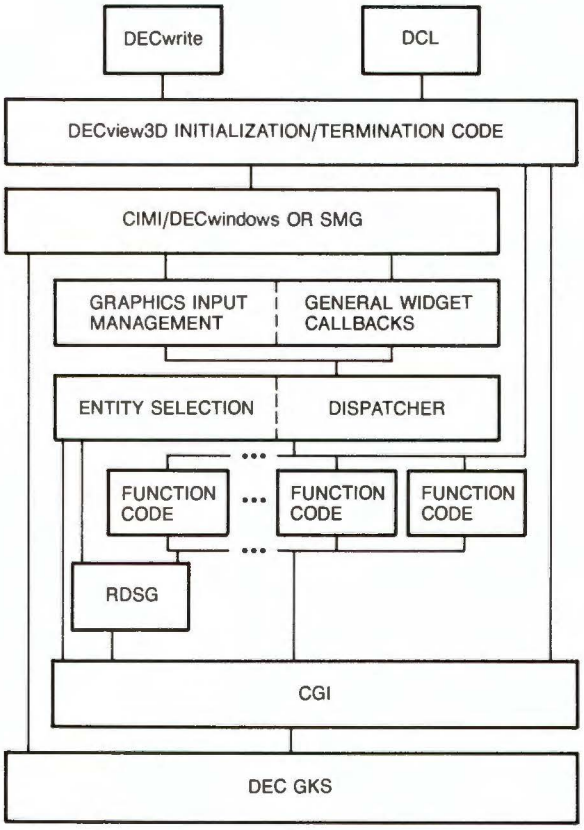


Figure 4 The DECview3D Architecture

The callback routine puts the information about the event into a data structure called an input packet. This packet is then passed to the dispatcher for further processing.

Dispatcher The dispatcher receives all the input packets from the graphics input management and the general widget callback routines. Its place in the architecture makes the dispatcher a focal point for all user input and positions it as a location for any future journaling capabilities.

When the dispatcher receives an input packet, it calls either the specific function code or the entity selection code. The code selected is based both on the content of the input packet and on the current application state. The default state for graphics input is select entity, and the default state for function code is initialize function. Many functions are done in

separate pieces and need a different interpretation of the graphics input. For this reason, the application state is set and changed by the function code.

Entity Selection The entity selection code controls the list of currently selected entities. This list is used by subsequently selected functions. When a select entity event occurs, the entity selection code calls CGI with the cursor location of the event. CGI returns an entity identifier to the entity selection code based on the proximity of the entity to the cursor location. The entity selection code then either adds it to or removes it from the list of currently selected entities. The entity selection code then calls RDSG to change the display of the entity.

RDSG The RDSG run-time structure contains all the internal entity data for the DECview3D application, including lines, splines, arcs, text, surfaces, views, and transformations. The DECview3D application uses the RDSG archival format for its native file format. All DECview3D translators use RDSG as their intermediate structure. During entity selection, the entity selection code calls RDSG to display the entity as highlighted and marks the entity in the RDSG data structure as selected. If the entity is already selected, RDSG is called to display the entity, and the entity is marked as not selected. In general, RDSG calls CGI for all entity display formats, including repainting, zooming, scrolling, and rotation. Thus, the geometric information may be transformed into two-dimensional graphic information.

CGI CGI transforms all three-dimensional RDSG data into two-dimensional graphic information for output through DEC GKS. This transformation allows graphics to be displayed on any device that DEC GKS supports. CGI supports multiple simultaneous displays of the same engineering and scientific data, each with a different three-dimensional transformation. CGI also permits engineering and scientific data to be simultaneously displayed from more than one source.

In addition to the display of entities, CGI contains a list of the geometry of selectable entities based on what has been displayed. When called by the entity selection code, CGI calls GKS to obtain an entity's input location by sampling the current cursor location. CGI then uses the list of selectable entities to determine which entity is closest.

The DEC GKS Product The DECview3D application can potentially display graphics on any display device with a DEC GKS device handler. GKS event mode allows the same user-interaction methodology on both DECwindows devices and graphics terminals. The DECview3D application supports output on multiple devices through the CDA architecture by creating DDIF format snapshots. In addition, the DECview3D application can also create snapshots in many formats, including the following:

- ReGIS
- Interleaf
- PostScript
- Sixel
- Hewlett-Packard Graphics Language (HPGL)
- Tektronix 4014

Function Code The function code controls the processing of all user requests, including view manipulation, annotation using text or simple graphics, translation, and creation of snapshots. Through the function code the DECview3D application brings engineering and scientific graphics into compound documents by translating data files into RDSG and creating DDIF snapshots from the RDSG data.[6]

DECview3D Summary

By making good use of existing software, the DECview3D design requirements were met and the product was delivered quickly. The DECview3D application provides a dynamic link between engineering and scientific graphics and compound documents. The application supports interactive two-dimensional and three-dimensional viewing of engineering and scientific data, data file translation, and graphics display on DECwindows devices and graphics terminals. The DECview3D product is a good example of Digital's use of Digital products.

Acknowledgments

The authors would like to acknowledge the contributions to the DECview3D architecture by team members Bill Carr (DECview3D project leader), William Hsu, and Jim Roth; and the help of Fiona Sanderson, DECview3D product manager, on the product description. We especially acknowledge the contributions of Barbara Higgins, the DECview3D technical writer, for improving the readability and organization of this and other DECview3D papers.

References

1. MIL-D-28000 Military Specification, "Digital Representation for Communication of Product Data: IGES Application Subsets" (Philadelphia: Navy Publications and Forms Center).
2. *VAX GKS User Manual* (Maynard: Digital Equipment Corporation, Order No. AI-HW45B-TE, February 1988).
3. *VMS RTL Screen Management (SMG$) Manual* (Maynard: Digital Equipment Corporation, Order No. AA-LA77A-TE, April 1988).
4. *XUI Style Guide* (Maynard: Digital Equipment Corporation, Order No. AA-MG20A-TE, December 1988).
5. S. Cohen and E. Morgan, "The Relationship between the DECwrite Editor and the Digital Document Interchange Format," *Digital Technical Journal*, vol. 2, no. 1 (Winter 1990, this issue): 73–82.
6. W. Laurune and R. Travis, "The Digital Document Interchange Format," *Digital Technical Journal*, vol. 2, no. 1 (Winter 1990, this issue): 16–27.

Further Readings

The Digital Technical Journal *publishes papers that explore the technological foundations of Digital's major products. Each Journal focuses on at least one product area and presents a compilation of papers written by the engineers who developed the product. The content for the Journal is selected by the Journal Advisory Board, which includes four Digital vice presidents and three senior engineering managers.*

Topics covered in previous issues of the *Digital Technical Journal* are as follows:

VAX 8600 Processor
Vol. 1, No. 1, August 1985
The design of a pipelined architecture and emitter-coupled logic

MicroVAX II System
Vol. 1, No. 2, March 1986
The implementation of the VAX architecture on a single CPU chip

Networking Products
Vol. 1, No. 3, September 1986
The Digital Network Architecture (DNA), and network management

VAX 8800 Family
Vol. 1, No. 4, February 1987
Products that support the VAX 8800 high-end multiprocessor and its family members

VAXcluster Systems
Vol. 1, No. 5, September 1987
Key hardware and software features of VAXcluster systems, and performance measurements

Software Productivity Tools
Vol. 1, No. 6, February 1988
Tools that assist programmers in the development of high-quality, reliable software

CVAX-based Systems
Vol. 1, No. 7, August 1988
CVAX chip set design and multiprocessing architecture of the mid-range VAX 6200 family of systems

Storage Technology
Vol. 1, No. 8, February 1989
Engineering technologies used in the design, manufacture, and maintenance of Digital's storage and information management products

Distributed Systems
Vol. 1, No. 9, June 1989
Products that allow system resource sharing throughout a network, the methods and tools to evaluate product and system performance

Subscriptions to the *Digital Technical Journal* are available on a yearly, prepaid basis. The subscription rate is $40.00 per year (four issues). Requests should be sent to Cathy Phillips, Digital Equipment Corporation, MLO1-3/B68, 146 Main Street, Maynard, MA 01754, U.S.A. Subscriptions must be paid in U.S. dollars, and checks should be made payable to Digital Equipment Corporation.

Single copies and past issues of the *Digital Technical Journal* can be ordered from Digital Press at a cost of $16.00 per copy.

Digital Press is Digital Equipment Corporation's international publisher of books for computer professionals who specialize in the areas of networking and data communication, artificial intelligence, computer-integrated manufacturing, windowing systems, and the VMS operating systems. Copies of the new titles now available from Digital Press that are listed below can be ordered by writing to Digital Press, Department DTJ, 12 Crosby Drive, Bedford, MA 01730, U.S.A.

COMMON LISP: The Language
Guy Steele Jr., Second Edition, 1990
($38.95 in softcover, $44.95 in clothcover)

The Matrix: Computer Networks and Conferencing Systems Worldwide
John Quarterman, 1990 ($49.95)

UNIX for VMS Users
Philip Bourne, 1990 ($28.95)

The VMS User's Guide
James Peters and Patrick Holmay, 1990 ($23.00)

A Beginner's Guide to VAX VMS Utilities and Applications
Ronald Sawey and Troy Stokes, 1989 ($23.00)

VMS Internals and Data Structures: Version 5 Update Xpress
Ruth Goldenberg and Lawrence Kenah, Volumes 1, 2, and 3, 1989 ($35.00)

VAX/VMS Internals and Data Structures: Version 4.4
Lawrence Kenah, Ruth Goldenberg, and Simon Bate, 1988 ($75.00)

Digital Guide to Software Development
Corporate User Publications Group of Digital Equipment Corporation, 1990 ($27.95)

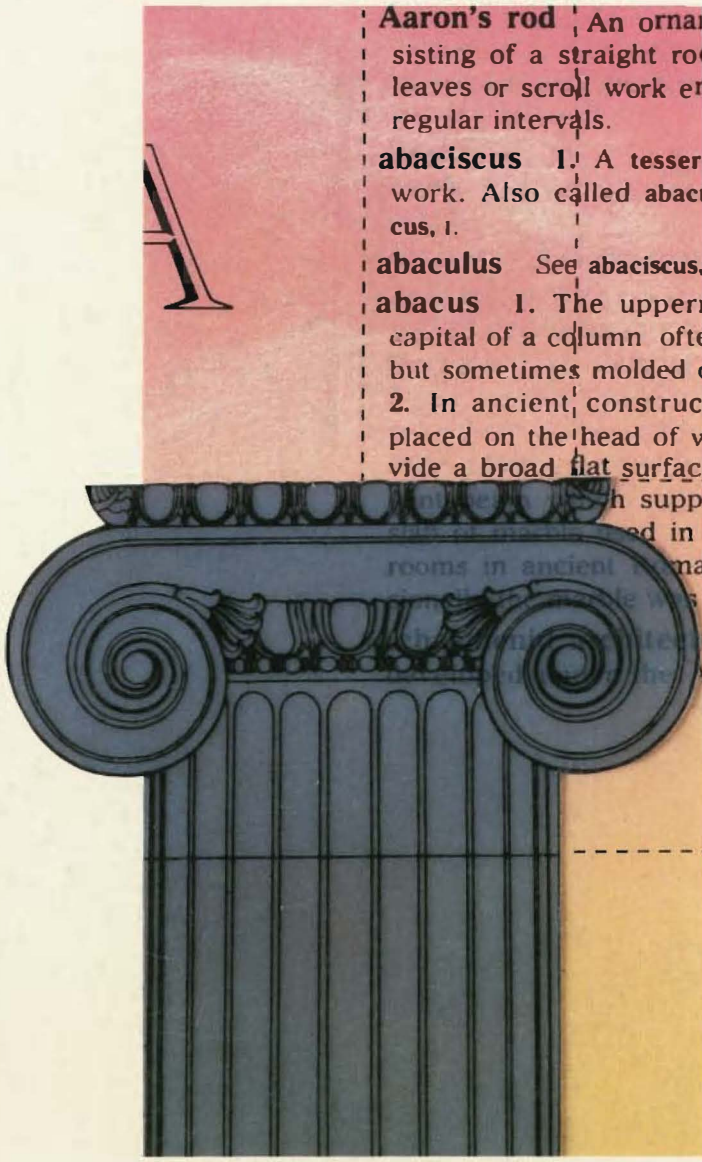
Technical Aspects of Data Communication
John McNamara, Third Edition, 1988 ($42.00)

Information Technology Standardization: Theory, Practice, and Organizations
Carl Cargill, 1989 ($24.95)

Computer Programming and Architecture: The VAX
Henry Levy and Richard Eckhouse, Second Edition, 1989 ($24.95)

ABCs of MUMPS: An Introduction for Novice and Intermediate Programmers
Richard Walters, 1989 ($24.95)

digital™



ISSN 0898-901X