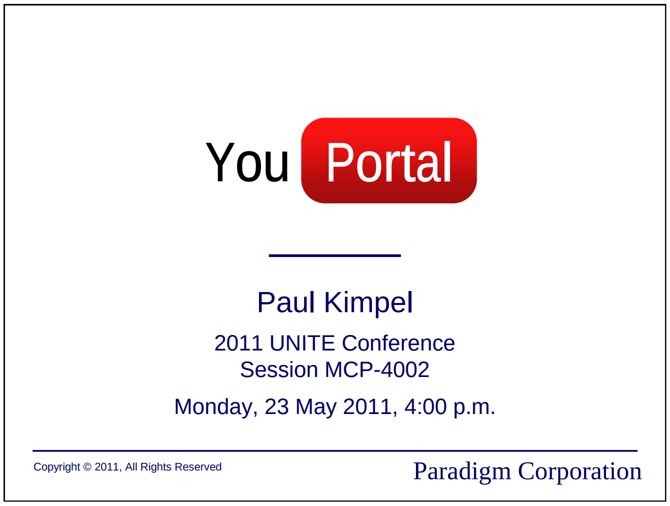


YouPortal

2011 UNITE Conference
Garden Grove, California

Session MCP-4002

Monday, 23 May 2011, 4:00 p.m.

Paul Kimpel

Paradigm Corporation
San Diego, California

http://www.digm.com

e-mail: paul.kimpel@digm.com

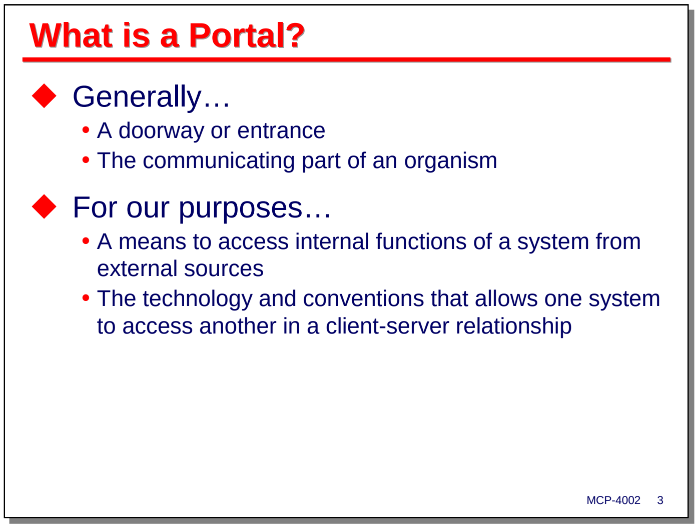
Presentation Topics

- What is a Portal?
- A Simple Portal Implementation
- Portal Components
  - Protocol and message format
  - DotNet Interface
  - MCP Interface
- Portal Application Design Issues
- Lessons Learned

MCP-4002 2

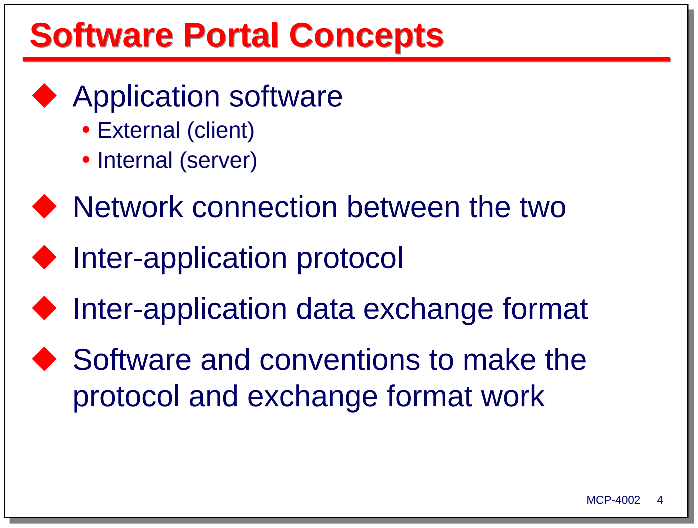
I have been working with a customer to modernize some of their MCP applications and user interfaces over the past couple of years. We have built a fairly simple mechanism for external applications and user interfaces (all Microsoft DotNet-based in this case) to connect to MCP applications and exchange data in an interactive fashion. This was easy enough to set up, and works well enough, that I thought it would make a good user-experience presentation for UNITE. Thus, this is the story of a real implementation, warts and all, solving a real problem for a real customer.

I'll begin with a brief discussion on what a portal is and what comprises one. I'll then present our simple portal implementation. The remainder of the presentation will be taken up with the components of that implementation, some portal application design issues, and a discussion of lessons we've learned along the way.



So, just what is a portal? In general, it's a doorway or entrance to something. The dictionary also defines it as the communicating part of an organism.

For our purposes, in the context of a data processing environment, a portal is a means for an external system to access resources and functions of some other system. In this specific case, that other system is an MCP-based application. A portal is also the collection of technology and conventions that allow those two systems to communicate in a client-server relationship.

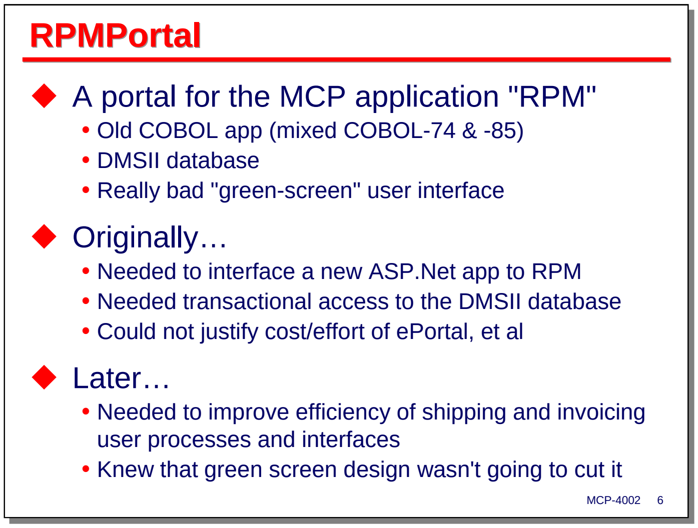


With a portal to software-based applications, you generally have the following high-level components:

- First, there are two sets of application software – the software that runs on the external system (the client), and the software that runs on the internal system (the server, our MCP system).
- Second, there is some sort of network connection between the two. This can by any type of communications mechanism, but these days it would typically be TCP/IP.
- Next we need a protocol to govern how the two sets of software will communicate. A protocol is simply a set of rules that describe how each end of the connection should act. This protocol runs on top of the basic network mechanism, so it would be typically something running over a TCP/IP connection.
- The purpose of the protocol is to enable exchange of data, so we also need some rules for how that data should be formatted.
- Finally, we will need some software and conventions that implement the protocol and data exchange format and allow the two sets of application software to use them.

A Specific Portal Implementation

With that very brief introduction, I'll now discuss the specific portal mechanism we've implemented.

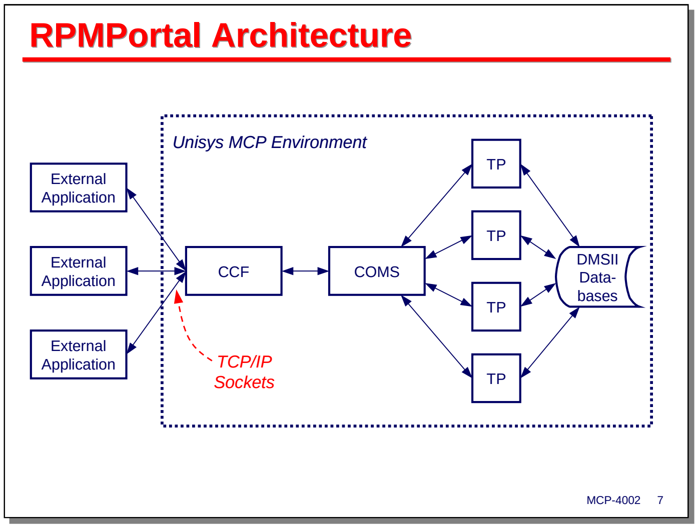


This portal was built for an MCP-based application named RPM, so we quite ingeniously called it "RPMPortal." RPM is an old COBOL application from the late '70s and early '80s. It's currently a mixture of COBOL-74 and -85, although lately we have been compiling everything with the COBOL-85 compiler. The application has a DMSII database. It also has one of the poorest green-screen user interfaces I've ever seen. Unfortunately, fixing the user interface would require a thorough rewrite of the on-line application programs, so we live with it.

The original impetus for RPMPortal was a new ASP.Net application that needed to exchange data with RPM. We had been using file transfer with extract and import programs to do this sort of thing for a long time, but the new application required a more responsive interface. In particular, it needed to bounce transactions off the DMSII database in real time.

We initially looked at a number of ways to implement this capability, and considered some purchased solutions, such as ePortal. In the end, we decided that what we needed was fairly simple, and we could not justify the cost and effort to acquire and implement something like ePortal, so we built our own.

After that initial project proved successful, we later found the need to radically improve the functionality and efficiency of our shipping and invoicing processes. The user procedures needed substantial improvement, and we knew that the existing green-screen mechanism just was not going to be adequate. We ended up building customer interfaces in ASP.Net and using RPMPortal to connect those user interfaces to new business logic and database structures running in the MCP environment.



The basic idea of RPMPortal is quite simple, and leverages a lot from standard MCP facilities. The key to this design is CCF, the Custom Connect Facility. CCF provides a way to connect a number of network protocols to COMS. For RPMPortal, we chose to use plain old TCP/IP sockets.

The external user interfaces (running as an ASP.Net web site on a Windows IIS box) use standard Windows sockets to connect to CCF in the MCP environment. CCF converts those socket connections to COMS pseudo-stations and establishes a separate COMS session for each one. The rest of the implementation on the MCP side consists of standard COMS Direct Window TP programs that access a DMSII database. The only thing unusual about these TPs is that they process portal-formatted messages instead of T27-style screen messages.

While to date we have only front-ended our portal with ASP.Net applications, in principle anything that can establish a TCP/IP connection could use the same portal mechanism to talk to MCP-based applications. There is nothing particularly new or innovative about this design – it's a straightforward application of CCF. Most of the interoperability is a matter of mutually-agreed conventions between the external and MCP-based application environments. The thing that really gives this approach value is the inter-application protocol and data exchange format we are using.

This basic design scales easily to handle multiple simultaneous clients for one application, multiple sockets for multiple applications, test vs. production environments, and even different protocols and data exchange formats.

Advantages of this Approach

- Inexpensive (cheap, in fact)
- Very fast initial availability (a weekend)
- Very easy to construct
  - Used standard MCP network facilities
  - Already had MCP library to handle a message format
  - Needed to write corresponding VB.Net classes
- Has just the capability we needed
  - Almost no learning curve
  - Can extend functionality as needed
- It works really well – for us

MCP-4002 8

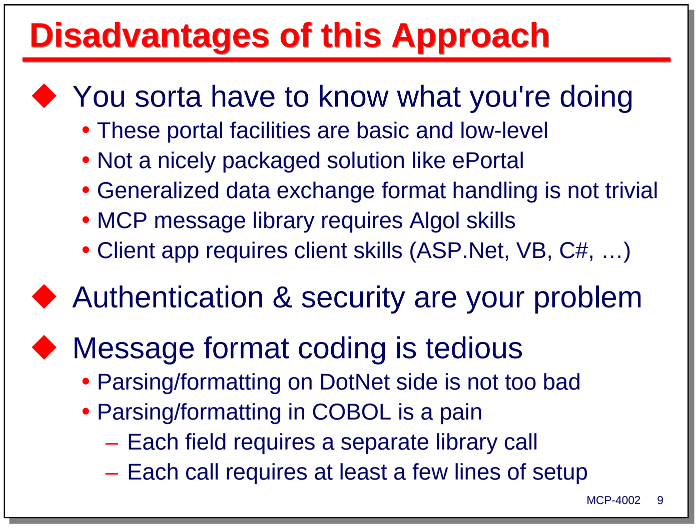
There are a number of advantages to this particular approach. The first is that it is inexpensive. In terms of capital outlay, it was essentially free. The customer already had the necessary network and Windows IIS environment in place, and we used standard MCP facilities that are part of the ClearPath IOE.

We were able to put together an initial prototype very quickly. I put together a simple demo user interface and MCP application together over a weekend. We have since enhanced the portal infrastructure quite a bit, and the production applications took significant resources to design and build, but the basic idea – and a fair amount of that first weekend's coding – have survived intact.

I had to write a couple of classes in VB.Net to support the Windows side of the inter-application protocol and data exchange format. From an MCP project several years ago, I had an Algol library that supported a data exchange format that seemed suitable, so having this library in hand considerably eased the job of getting an initial prototype in place and shortened the amount of time it took.

One nice aspect of this approach is that it gives us just the capability we need. There has been almost no learning curve involved for the portal itself. We can also extend the existing portal functionality as needed, and have already done so as new development has presented us with new requirements that needed to be supported.

In short, this approach works really well for us. It's not a comprehensive solution that anyone else might find suitable, but it was never intended to be.



This approach has some disadvantages, too. The first is that you need a fair amount of knowledge to put it together and make it work. The portal facilities themselves are very basic and operate at a fairly low level. It is not a nicely packaged solution like ePortal or some of the other third-party interface modernization tools.

In particular, managing a generalized data exchange format is not trivial. COBOL is not well suited to this task, so we are using an Algol library to parse and format the portal messages. Maintenance of this library requires Algol programming skills. Similarly, the user interfaces on the client side are custom-built applications, so implementation and maintenance of those require corresponding client-side skills – in our case ASP.Net and VB.Net programming.

The portal mechanism provides no user authentication, security, or encryption. If you need those, you have to implement them yourself. We decided that we did not need any protection between the IIS box and the MCP server, because they are a few feet from each other in the same locked room. User authentication and privileges are determined by standard ASP.Net security and some MCP application user tables in the database.

The parsing and formatting of the data exchange messages has turned out to be a lot of work and much more tedious than I originally expected it to be. On the DotNet side, it's not too bad because we have an object-oriented programming environment with good parameterization and data abstraction capabilities. On the MCP side, it's frankly a pain. In COBOL, each field must be dealt with separately, and as we all know, COBOL's ability to build software abstractions is limited. The parsing and formatting of every field requires at least a few lines of coding to set up calls to the library, and I have not found a reasonably-maintainable way of doing that except by using lots of linear code.

The Big Disadvantage

- *This is not an interface modernization tool*
  - Can't just slap a web interface on an existing MCP app
  - MCP side requires extensive coding for message parsing and formatting
- Best if used for new features/functions
- Good GUI/web interfaces usually require a new back-end design, anyway
  - WYSIWYG requires a lot of back-end support
  - Most modern user interfaces are highly reactive
  - Most legacy app designs are too imperative to make a good base for WYSIWYG

MCP-4002 10

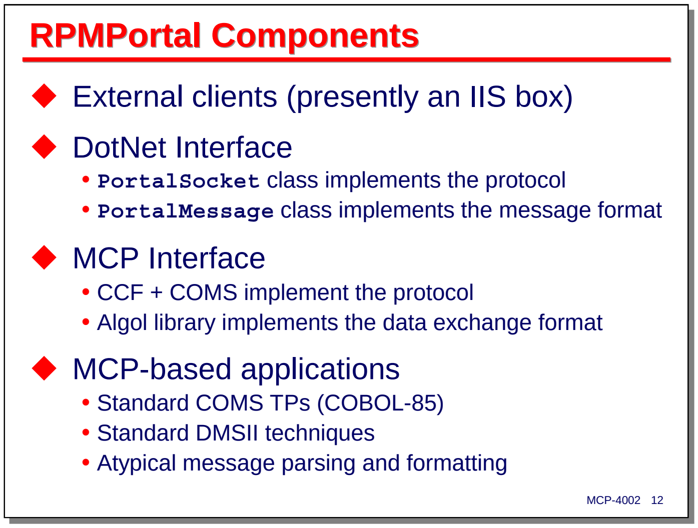
Perhaps the biggest disadvantage to this portal approach is that it is not a good tool for modernizing an existing interface. You cannot use it to just slap a new web interface on an existing MCP application. Because of the way the data exchange format works, the MCP side requires extensive coding to parse and format fields, and thus existing applications cannot be easily adapted to this approach.

Therefore, this portal approach works best if it is used to implement new features and functions in an application. That turns out to be what we wanted to do, so the need to code field processing in an entirely different way on the MCP side was not a problem for us. We were going to write all new code, anyway.

There is another point to be made here, though. Good GUI/web user interfaces usually require a new back-end design and implementation, anyway. A WYSIWYG interface requires a lot of back-end support. Most modern user interface designs are highly reactive to the user – you present the user with choices, and they pick one. Most legacy application designs are very imperative – instead of WYSIWYG (What You See Is What You Get), they are YAFIYGI (You Asked For It, You Got It). They just don't present data to the user interface in ways that are either useful or appropriate for a modern user interface design.

RPMPortal Components

In this next section, I'll discuss in some detail the components that make up the RPMPortal implementation and the APIs that it makes available to the application software.



RPMPortal currently consists of the following components:

- External clients. From the MCP's perspective, the client is presently a single IIS box. The end-user clients are web browsers communicating with an ASP.Net web site.
- Two interface classes for DotNet applications, written in VB.Net:
  - **`PortalSocket`** is a class that implements the inter-application protocol
  - **`PortalMessage`** is a class that implements the inter-application data exchange format.
- On the MCP side, CCF and COMS implement the protocol, and the Algol library implements the data exchange format.
- The MCP-based applications are standard COMS TPs written in COBOL-85. They use standard COMS send/receive techniques and standard DMSII query and update techniques. The atypical thing about them is that they deal with messages in the data exchange format rather than as T27 screens.

Inter-Application Protocol

- Request-response mechanism
- Standard TCP/IP socket connection
  - MCP and IIS box are assumed to be trusted
  - Sit on same rack in a locked server room
  - No authentication or encryption between them
- Message framing convention
  - Use CCF **`FRAMING=STANDARD`** format
  - 6-byte binary header on each message block
    - Header signature (hex **`ABCD`**)
    - 16-bit binary message sequence number
    - 16-bit binary message length

MCP-4002 13

The inter-application protocol is very basic. It is simply a request-response mechanism. All communications are initiated by the client with a request. The MCP side sends a response to each request.

Communications between client and server take place over a standard TCP/IP connection. As mentioned earlier, there is no security in place between the IIS box and MCP – that connection is assumed to be trusted. Both servers sit in the same rack in a locked server room, so this is an adequate assumption for our purposes.

TCP is a stream-based communication protocol. This means that it does not implement any message boundaries – it just delivers streams of bytes. We need to delimit the hunks of bytes that represent requests and responses, so we chose the CCF **`STANDARD`** framing scheme. This is handled entirely by CCF and is transparent to the MCP applications. The **`STANDARD`** frame consists of a six-byte header followed by the bytes of the message itself. The contents of the header are:

- A two-byte signature that helps identify the beginning of a frame. The value of the 16 bits in the signature is hex **`ABCD`**. All values in the header are in network (big-endian) order.
- A two-byte (16-bit) message sequence number. The numbering sequence starts at one and wraps at 65535 back to one.
- A two-byte (16-bit) message length. This specifies the length of the message data following the header. It does not include the length of the header.

This framing format provides good integrity for message sequencing and delimiting. Running on top of TCP/IP (which provides pretty good data integrity itself), it's probably more than we need. We chose it because it's built in to CCF and provides for transparent message handling – there are no reserved byte values for delimiting the messages, as there are with other framing methods.

Inter-Application Data Format

- Evaluated a number of potential formats
  - Positional: fixed-length fields, CSV, tab-delimited
  - Non-positional: URL-encoded, XML, JSON
- Ended up using one from a former project
  - "MDC" format
  - Better than URL-encoding, not as good as JSON
  - Already had an Algol library to parse/format messages
- Basically a two-level name/value scheme
  - Good for master/detail data structures, plus a little more
  - No data-type information – everything is a string
  - Types established by convention, based on field names

MCP-4002 14

For the inter-application data exchange format we evaluated a number of potential formats. We looked at some simple positional ones, such as fixed-length fields (i.e., COBOL-style records), comma-separated values, and tab-delimited values. We decided these did not offer enough support for data structures, and do not handle on-going changes to message layouts well. We also looked at some non-positional formats, including URL-encoded strings, XML, and JSON. URL-encoding is a basic name/value scheme, which again did not offer enough support for data structures. XML can be used to represent just about anything, but it can be really complex, and at the time did not have any MCP support – that has since changed.

I really thought hard about using JSON (Javascript Object Notation, http://www.json.org), but in the end decided to use something from a former project, which I call "MDC" format. This is a name/value scheme, better than URL-encoding, but not as robust as JSON. The main reason I chose it was that I already had (from the former project) an Algol library that COBOL programs could call to parse and format the messages.

MDC format is a two-level set of name/value lists. It's good for master/detail structures, which are common in business applications, plus a little more. One drawback is that it carries no data type information – all values are strings. We ended up establishing data types by convention between the clients and server – the field name implied its type, and both ends had to agree on what that was.

MDC Message Structure

- Variable-length ASCII text string
  - Actually, it's ISO-8859-1 (Latin-1)
  - Uses ASCII control characters for delimiters
  - [ ] ⇒ optional element, { } ⇒ repeating element
- Message:
  SOH *header* **STX** [*body*] **ETX** *checksum* **EOT**
- Checksum:
  - Optional, may be binary zero if not used
  - Otherwise,
    - Binary sum of ASCII codes, **SOH** thru **ETX**, inclusive
    - High-order bit of sum set to 1
  - Typically not used with TCP/IP networks

MCP-4002 15

I think the MDC message structure is sort of interesting, so I'll describe it in some detail over the next couple of slides. An MDC message is a variable-length ASCII text string. Actually, the character set is ISO-8859-1, also known as ANSI or Latin-1. It is a delimited format, and uses several ASCII control characters as the delimiters. In the diagrams on the slide, square brackets indicate elements of the format syntax that are optional and curly braces indicate elements that are repeating (zero or more times).

An MDC message begins with an **SOH** (Start of Header, hex 01) character and is followed by a *header* consisting of one or two fields. After the header is an **STX** (Start of Text, hex 02) character, which signals the beginning of the message *body*. An **ETX** (End of Text, hex 03) character signals the end of the body. Following the **ETX** is a *checksum* byte, and following that is an **EOT** (End of Transmission, hex 04) character that terminates the message.

The checksum is always present, but isn't really necessary on modern networks. The MDC format was originally designed to work over RS-232 circuits, which is why it includes a checksum capability. If used, it is the binary sum of all of the characters from **SOH** through **ETX**, inclusive, with the high-order bit of the result forced to 1. This insures that the checksum is never an ASCII control character. If the checksum is not used, is has the value binary zero, which is the ASCII **NUL** character. We chose not to use the checksum.

MDC Structure, continued

- Header:
  trancode **FS** [*msg-token*]
- Body:
  field-list [**GS** *record-list*]
- Field-list:
  { *field-name* [= [*field-value*]] **FS** }
- Record-list:
  { *field-list* **RS** }

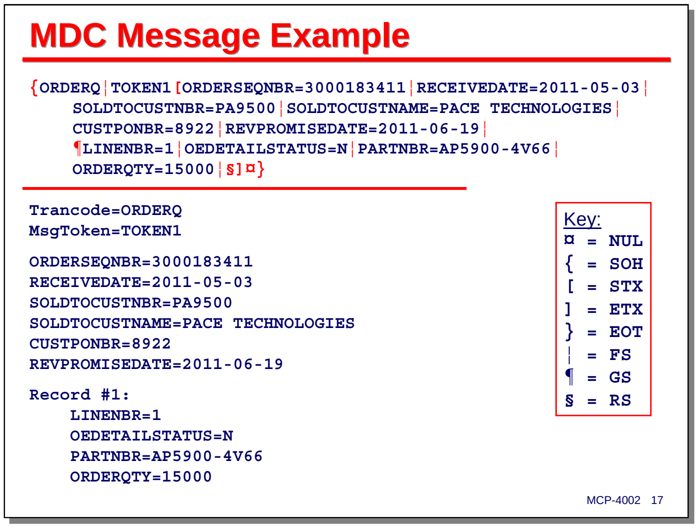
MCP-4002 16

The MDC *header* between the **SOH** and **STX** delimiters consists of one or two fields, separated by an ASCII **FS** (Field Separator, hex 1C) character.

- The *trancode* field is a string of up to 30 characters. It is intended to identify the type of message and how it should be processed.
- The *message token* is also a string of up to 30 characters, but can be empty. It was originally intended to indicate the source of the message, but we use it as an opaque request token. The MCP applications always echo the token from the request message in any replies to that message. This potentially allows a multi-threaded client to sort out which reply messages go with which requests.

The body of the message between the **STX** and **ETX** characters consists of a *field-list*, optionally followed by a **GS** (Group Separator, hex 1D) character and something called a *record-list* (also known as a *form-list*).

- A *field-list* is just a sequence of name/value pairs.
  - The field name is simply a string of up to 30 characters.
  - The field value is optional, but if present must be preceded by an ASCII "equals" sign (hex 3D).
  - The name/value pairs in the list are delimited by **FS** characters. An **FS** after the last name/value pair in the list is optional.
- A *record-list* is just a sequence of *field-lists* delimited by **RS** (Record Separator, hex 1E) characters. An **RS** after the last *field-list* is optional. The *record-list* is what gives the MDC format its two-level nature. The first *field-list* in the body is used for the master fields; the *record-list* can represent an arbitrary number of detail entries. Interestingly, the detail entries do not all have to have the same structure. Each detail "record" potentially can have different fields from the other records. Note that the entire *record-list* is optional, so a simple MDC message is little more than a string of name/value pairs with a header on the front.



This slide shows a sample MDC message with the ASCII delimiter characters represented by the printable graphics highlighted in red. This message has a *header*, a main *field-list* in the body, and a *record-list* consisting of a single *field-list*.

MDC Conventions for RPMPortal

- Final delimiter in a field- or record-list is optional
- Client app *may* prefix **STX** by a *COMS Trancode* value to route the message
- Checksum is always zero
- MCP app *may* segment long outputs
  - Each COMS **SEND** results in one CCF frame
  - **PortalSocket** accumulates frames until it sees **EOT**
  - MCP inputs currently *must* be only one frame
  - Allows for 1-in, many-out transactions

MCP-4002 18

As mentioned, the final delimiter in the *field-* and *record-lists* is optional.

Originally I thought it would be possible to do COMS Trancode routing based on the *trancode* field in the header. That turned out to be more trouble than it was worth, so by convention, the portal clients may optionally prefix the MDC message with a COMS Trancode. We're currently using the COMS Trancode to route the message to the appropriate TP program, and the MDC *trancode* in the message *header* to route the message for processing within the TP.

Also as previously mentioned, we are not using the checksum, so its value is always zero.

Since we were trying to build a nice user interface, one of the things we needed to be able to do was retrieve lists of things from the MCP database and display them to the user. Usually the user can click on something in these lists to activate a function or drill down into a more detailed display for whatever entity that list entry represents. The MDC-formatted data for some of those lists got long – much longer than the 64K byte limit of the CCF frame header or the limit on the size of a COMS message.

Our solution to this was to allow the COMS TPs to segment their responses. They will retrieve and format data into a message area until it reaches a certain size threshold, then do a COMS **SEND** on the accumulated message, and then continue formatting from the beginning of that message area. Thus a single response can consist of multiple COMS **SEND**s.

Each COMS **SEND** results in one CCF frame being sent over the network to the ASP.Net client. This means that an MDC message sent by the MCP could span multiple CCF frames. The **PortalSocket** class in the DotNet environment is the one that handles the CCF frame, so we simply modified it to accumulate frames until it saw one with an **EOT** in it. It then strips the frame headers and concatenates the frame payloads to reconstruct the MDC message in one piece. Fortunately, dynamically allocating and processing multi-megabyte messages is not a problem for the DotNet runtime environment, especially since lifetime of the memory allocation for these messages is fairly short.

This segmentation technique may be a bit of a kludge, but it was easy to do and works quite nicely. Request messages to the MCP are still limited to one CCF frame, but responses can be arbitrarily long (up to the 2GB limit of a DotNet String object), and the segmentation is only exposed on the DotNet side within the **PortalSocket** class.

DotNet Interface

The next subject to discuss is the DotNet interface in detail – handling of the protocol, CCF frames, and MDC message format.

DotNet PortalSocket Class

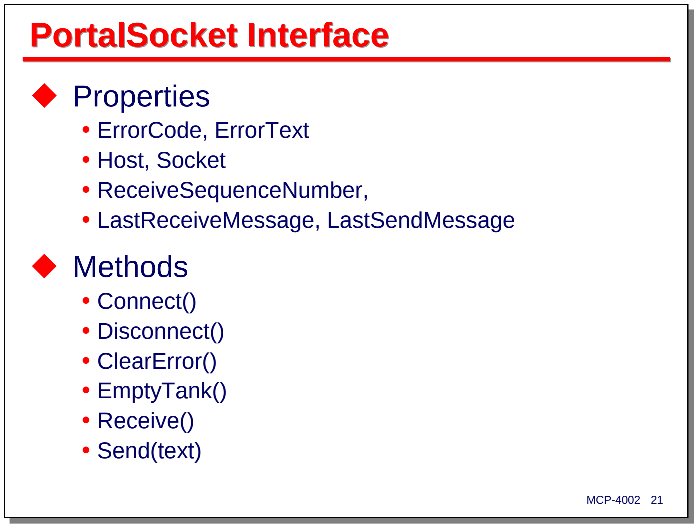
- VB.Net class to manage protocol for client
  - Uses **`System.Net.Sockets`** **`TCPClient`** and **`NetworkStream`** for TCP/IP socket
  - Each socket creates one COMS session
  - Uses synchronous (blocking) I/O
- On input:
  - Validates and strips CCF frame header
  - Accumulates frames to form one MDC message
  - Translates MDC text from ISO-8859-1 to Unicode
- On output:
  - Translates MDC text from Unicode to ISO-8859-1
  - Prepends CCF frame header

MCP-4002 20

On the DotNet side, the **`PortalSocket`** class handles protocol and CCF framing for the ASP.Net application. It uses the **`TCPClient`** and **`NetworkStream`** classes from the DotNet **`System.Net.Sockets`** namespace to open a connection to the MCP server and send and receive data. Each socket connection causes CCF to create a separate COMS session. The network communications operate in a synchronous (blocking) fashion, but since we are implementing a request/response mechanism, that is not a problem.

When receiving a message from the MCP, this class validates and strips the CCF frame header. It also accumulates frame payloads as necessary until it sees one with an EOT character in it, signaling the end of the MDC-formatted message. It concatenates all of the frame payloads and translates the resulting string from the ISO-8859-1 encoding used over the wire to Unicode for use within the DotNet environment.

When sending a message to the MCP, this class reverses the process used on input. It translates the message text from Unicode to ISO-8859-1, constructs and prepends the CCF frame header, and then transmits the data.



The API for **`PortalSocket`** is very straightforward. There are two properties to establish the MCP server's IP address and socket (port) number, two properties for reporting errors back to the caller, and some debugging properties that allow us to examine the message sequence number from the CCF header and the text of the last messages that were sent or received.

The class has methods to connect to and disconnect from the MCP server, clear the error properties, empty the internal memory area ("tank") used to receive and accumulate frames, and to do the actual receives and sends for the connection.

There is a *Technical Guide* document in the materials accompanying this presentation that describes the API in more detail.

DotNet PortalMessage Class

- VB.Net class to support the MDC format
  - Parses fields from incoming messages
  - Formats fields for outgoing messages
  - Converts field and record lists to/from collection objects
  - Provides conversion between strings and DotNet types:
    - Integer, Long, Decimal
    - Boolean
    - Date, Time
- Interfaces with **`PortalSocket`** class
  - Receives a complete message from **`PortalSocket`**
  - Generates a complete message for **`PortalSocket`** to send

MCP-4002 22

The **`PortalMessage`** class handles the parsing and formatting of MDC-format messages for the DotNet environment. It converts the *field*- and *record-lists* in MDC messages to internal DotNet collections. The applications construct an output message by populating the internal collections with values and then calling a method of the class to convert those collections to an MDC-format message. The class also provides methods to convert between the purely string values represented in the MDC-format messages and common DotNet types – Integer, Long, Decimal, Boolean, Date, and Time.

`PortalMessage` interfaces with the **`PortalSocket`** class. **`PortalSocket`** accumulates a complete MDC message on input, which **`PortalMessage`** parses and converts to the internal collection objects. **`PortalMessage`** formats a complete MDC message on output and passes it to **`PortalSocket`** for conversion to a CCF frame and transmission to the MCP server.

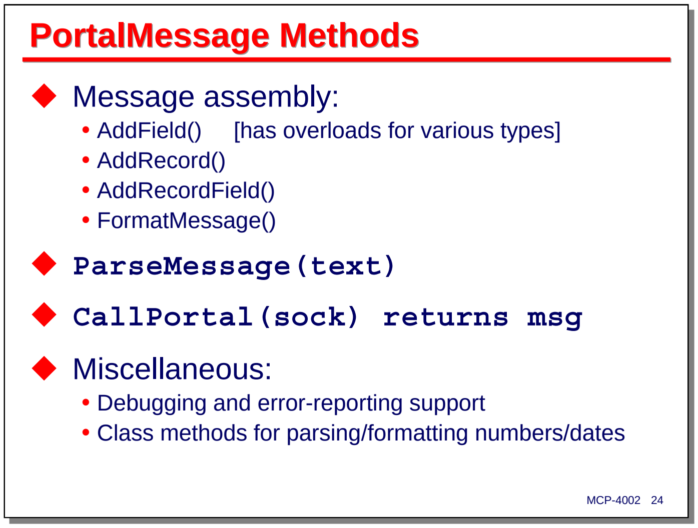
PortalMessage Properties

- Message envelope:
  - RoutingCode (COMS Trancode, output only)
  - Trancode, MsgToken
  - Checksum
- ErrorCode, ErrorText
- Fields collection
  - `FieldList = Generic.Dictionary(Of String, String)`
  - Holds parsed name/value pairs from main field-list
- Records collection
  - `RecordList = Generic.SortedDictionary(Of Integer, FieldList)`
  - Indexed collection of field-lists in the record-list

MCP-4002 23

`PortalMessage` has a larger API than **`PortalSocket`**, largely due to the number of type-conversion methods it supports. It has a number of properties as well:

- There are properties to interrogate and set fields in the message envelope – the optional COMS Trancode that prefixes the MDC-formatted data, the *trancode* and *msg-token* fields of the MDC *header*, and the *checksum*, which is read-only.
- There are properties to report errors back to the caller, similar to those for **`PortalSocket`**.
- The principal property is the **`Fields`** collection, which holds the name/value pairs for the main *field-list* in the MDC message. It has a type of **`FieldList`**, which is a public, internal class, and is built on the DotNet **`Generic.Dictionary(Of String, String)`** class.
- If the MDC message contains a *record-list*, the parsed version of that is in the **`Records`** collection, which is based on the **`Generic.SortedDictionary(Of Integer, FieldList)`** class. This class behaves something like an array of *field-lists*, which in turn use the same class as the main *field-list*.



PortalMessage has a sizable number of methods, as described over the next two slides.

Data for an output message is assembled through a series of "add" methods:

- **`AddField()`** appends a name/value pair to the main *field-list* collection. It has overloads to convert a number of DotNet types to the string values that are actually stored in the collection.
- **`AddRecord()`** appends a new, empty record to the *record-list* collection.
- **`AddRecordField()`** appends a name/value pair to a specified record in the *record-list* collection. The record is specified by its index value. If that record does not exist, it is first created.
- **`FormatMessage()`** converts the data in the **`Fields`** and **`Records`** collections to MDC format and returns the complete message as a String value.

`ParseMessage()` takes a complete MDC-format message as a parameter, validates it, parses the fields, and loads them into the **`Fields`** and **`Records`** collections of its PortalMessage instance.

`CallPortal()` implements a single request/response interchange. We found we were writing the same calls to **`PortalSocket`** send and receive methods over and over, so we implemented a standard request/response method in **`PortalMessage`** to standardize that common function. The application will execute **`CallPortal()`** on a **`PortalMessage`** instance, passing a **`PortalSocket`** object. The method will execute **`FormatMessage()`** on that message instance to generate the MDC-format message and send it using the supplied socket object. It then uses the socket object to receive a complete MDC-format response, instantiates a new **`PortalMessage`** object, executes **`ParseMessage()`** on that new instance to load the response data into the instance's collections, and returns the new **`PortalMessage`** instance to the caller. It also does general error checking for the send and receive processes.

There are some miscellaneous methods for debugging and support of error reporting. There are also several class (i.e., static, or what VB.Net terms "shared") methods to do utility parsing and formatting of numbers and dates.

PortalMessage FieldList Methods

- FieldList assembly:
  - Add()   [has overloads for various types]
- FieldList field extraction & conversion:
  - GetBoolean()
  - GetDate()
  - GetDecimal()
  - GetInteger()
  - GetLong()
  - GetString()
- Conditional field extraction & conversion:
  - TryGet…() for each of the types above

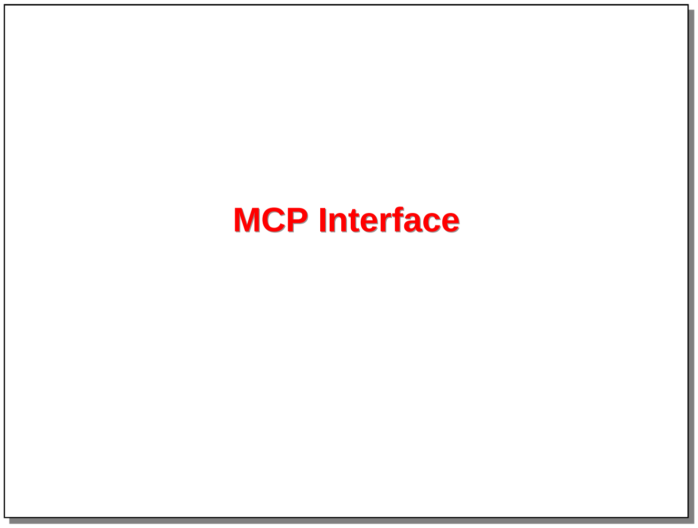
MCP-4002 25

There are a number of PortalMessage methods for appending new name/value pairs to *field-lists* and converting the string values from *field-list* entries into native DotNet types.

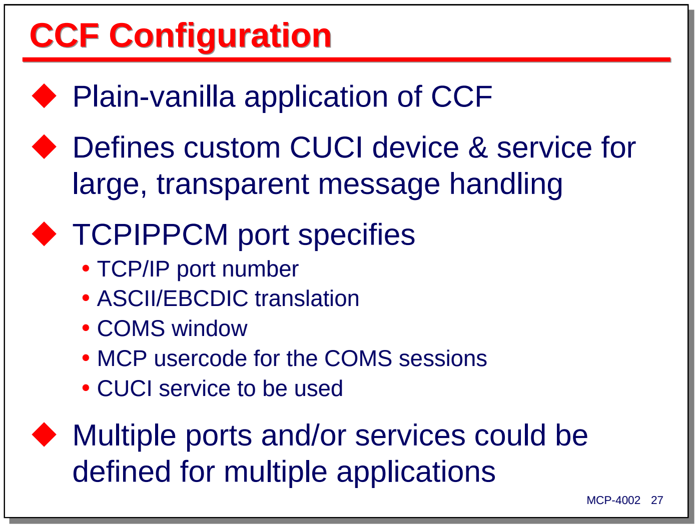
`Add()` will append a name/value pair to its instance. It had a number of overloads to format various DotNet types as string values.

There are a series of **`Get...()`** methods to select a list entry and convert its value to the designated DotNet type. If the name does not exist in the list, or the value is not valid for the requested type, these methods return a default value, e.g., zero for numeric types.

The **`TryGet...()`** methods are similar to the **`Get...()`** methods, but return a Boolean result that indicates whether the conversion was successful, rather than just returning a default value if it was not.



Our next subject is the configuration of the MCP interface and the API provided by the Algol library than handles the MDC message format.



RPMPortal is a plain-vanilla application of CCF. The configuration consists of three items:

- We defined a custom CUCI device to handle large, transparent messages. The primary purpose of this CCF device is to prohibit NDL-style input editing of the message.
- We also defined a custom CUCI service to route the messages into COMS.
- The actual network interface is defined by a TCPIPPCM port. It specifies the TCP port (socket) number on which the MCP will listen for connections, ASCII/EBCDIC translation, the COMS window to which incoming messages will be routed, a constant MCP usercode to be used for the COMS sessions (end-user authentication is handled between ASP.Net and some application-level coding on the MCP side), and the name of our custom CUCI service.

We currently have two CCF ports defined, one for production and one for our test environment. We could easily add more CCF ports and services to accommodate additional applications and/or multiple COMS windows.

Sample CCF CUCIPCM Config

```
ADD DEVICE RPMPORTALDEV
    CCENABLE =              FALSE,
    CONTROLCAPABLE =        FALSE,
    DEFAULTVT =             TRANSPARENT,
    LINEWIDTH =             0,
    MARCCAPABLE =           FALSE,
    MAXINPUT =              32767,
    MAXOUTPUT =             32767,
    MESSAGES =              NONE,
    PAGELENGTH =            0,
    SCREEN =                FALSE,
    SINGLEWINDOW =          TRUE,
    USAGE =                 IO;

ADD SERVICE RPMPORTALSVC
    CLOSEACTION =           3,          % NO WINDOW
    DEVICE =                RPMPORTALDEV,
    DYNAMIC =               TRUE,
    IDLEDISCONNECT =        TRUE,
    LOGOFFDISCONNECT =      TRUE,
    LOGONREQUIRED =         TRUE,
    DEFAULTWINDOW =         RPMPORTAL;
ENABLE SERVICE RPMPORTALSVC;
```

MCP-4002 28

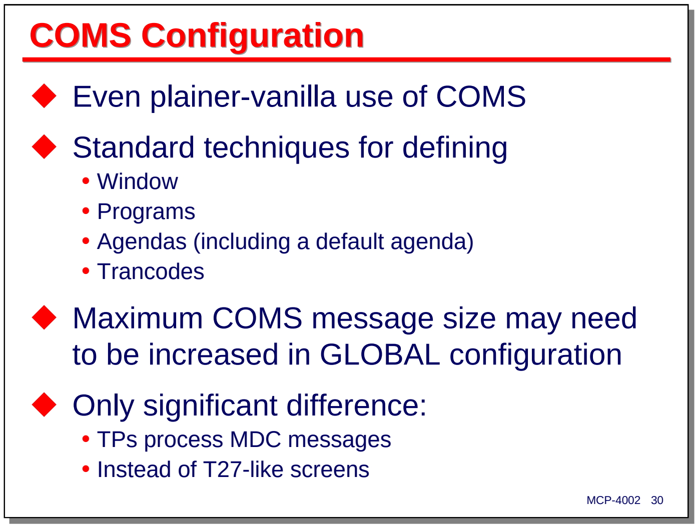
This slide shows the CCF CUCIPM configuration for the custom device and service we use with RPMPortal.

Sample CCF TCPIPPCM Config

```
ADD PORT RPMPORTAL
    BLOCKEDTIMEOUT =   30,         % MINUTES
    CHECKINTERVAL =    5,          % MINUTES
    FRAMING =          STANDARD,
    MAXINPUT =         32767,
    MAXOUTPUT =        32767,
    MAXOFFER =         2,          % TOTAL PASSIVES
    MAXSUBPORT =       32,         % TOTAL OPEN PORTS
    MINOFFER =         2,          % OFFERS AT A TIME
    PASSIVEOPEN =      TRUE,       % IT'S A SERVER
    SERVICE =          RPMPORTALSVC,
    SOCKET =           1036,
    STATIONNAME =      RPMPORTAL/$IPADDRESS/$YOURNAME,
    TRANSLATE =        TRUE,
    TRANSPORT =        TCPIP,
    USERCODE =         RPMPORTAL,
    WINDOW =           RPMPORTAL;
ENABLE PORT RPMPORTAL;
```

MCP-4002 29

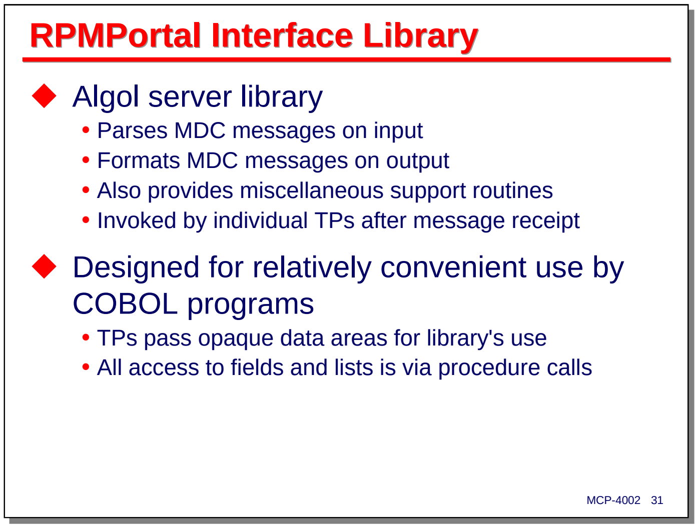
This slide shows the TCPIPPCM port configuration we use with RPMPortal.



The COMS configuration for RPMPortal is even more straightforward than that for CCF. We are using completely standard techniques for defining a Window and the elements for Direct Window TPs – Programs, Agendas, and Trancodes.

Since the portal uses larger messages than most green-screen applications, you may need to modify the maximum COMS message size in the **`GLOBAL`** section of the COMS configuration. We are currently using 32K, but could go up to 64K, the COMS limit.

The most significant difference for handing the portal in the COMS environment is that we are processing MDC messages instead of T27-style messages. This is primarily a TP application coding issue. Except for needing to support larger message sizes, it does not affect the COMS configuration.



The piece that really enables portal processing on the MCP side is the Algol library. It provides COBOL programs the ability to parse MDC messages on input, format MDC messages on output, and call some utility functions, such as character translation, string trimming, and substring search.

Handling of MDC messages is not automatic. The TPs receive data from COMS in MDC format, and must call the appropriate library procedures to parse the messages and extract individual fields one at a time. On output, the TPs must call the library to construct MDC messages one field at a time.

The whole purpose of this library is to standardize handling of the MDC format and to allow COBOL programs to work with this format relatively conveniently. The parsing mechanism builds some tables that make later field-level access more efficient. To persist these tables between library calls, the TP must pass some opaque data areas (COBOL 01 records) to the parsing and field extraction routines. The library manages these data areas internally, and the TPs need to leave this data alone, simply passing the areas as parameters when required. All access to the fields, field lists, and record lists of MDC messages is through library procedure calls – the COBOL programs never manipulate the message areas directly.

Library Message-Parsing Methods

- **MDC_ASSEMBLE_MSG**
  - Assembles a complete MDC message from COMS
  - Not strictly necessary when using CCF framing
- **MDC_PARSE_REPORT**
  - Parses header, main field-list, and record- (form) list
  - Returns header fields, field dict, and form dict
- **MDC_PARSE_FORM**
  - Parses fields from one record (form)
  - Returns dictionary for the field-list of the record
- **MDC_PARSE_FIELDS**
  - Parses one field-list from message to a dictionary
  - Not generally called directly

MCP-4002 32

When an MDC message is received by a COMS TP, it must first prepare it for processing. There are two library calls that are required for this, and a couple of optional ones.

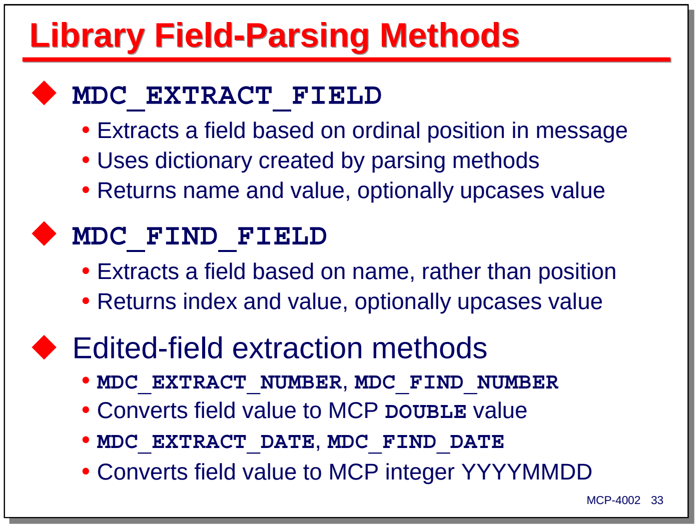
MDC_ASSEMBLE_MSG assembles a complete MDC message from COMS input messages. The library was originally written to support CCF **FRAMING=NONE**, which delivers buffers of TCP stream data as they arrive from the network. This means that COMS did not necessarily deliver complete MDC messages on each **RECEIVE** – the input buffer could contain an arbitrary collection of messages and fragments of messages. The purpose of message assembly is to collect complete MDC messages from this fragmented input and deliver them one at a time for processing. With CCF **FRAMING=STANDARD**, however, this process is no longer necessary – the collection of network data fragments into complete frames is handled by CCF, and therefore COMS should be delivering complete MDC messages to the TP. This routine has been retained in this implementation, however, since it validates the MDC message envelope, computes the checksum (if being used) and extracts the MDC header *trancode* field, which is useful for internal message routing.

MDC_PARSE_REPORT does the initial top-level parsing of the MDC message. A complete MDC message was originally called a "report." This routine scans the message and builds dictionary structures for the main *field-list*, and if second level "records" are present, a dictionary for the *record-list*. It does not parse the individual *field-lists* within records, however. The dictionaries are opaque data structures to the caller, and are simply passed in later calls to other library routines. It also extracts the two MDC *header* fields, *trancode* and *msg-token*, and returns their values to the caller.

MDC_PARSE_FORM parses the *field-list* for one record in the optional record-list. Records were originally called "forms," hence the name. The intention is that the TP will call **MDC_PARSE_REPORT** once for the message, and if a *record-list* is present, process the records one at a time, calling this routine once for each entry in the *record-list*. Its primary function is to build a dictionary structure for the field-level routines to use in extracting and converting field values.

MDC_PARSE_FIELDS is the routine that actually parses a *field-list* and builds the dictionary structure. It is called by **MDC_PARSE_REPORT** and **MDC_PARSE_FORM**, and is not normally called directly by the TP.

See the *Technical Guide* in the materials accompanying this presentation for a detailed description of each of these routines and the parameters they require.



Once the message or a record has been parsed, the TP can call field-level routines to extract and optionally convert the data for use. There are two main extraction routines, and a few others built on top of those.

`MDC_EXTRACT_FIELD` extracts a field from the message based on its ordinal position in the message. The dictionary structure built by the parsing procedure makes this an efficient operation. This routine is used when the TP needs to step through all of the fields in the message to see what is there. The routine returns the name and value for the field, optionally upcasing the value.

`MDC_FIND_FIELD` is typically used more often and **`MDC_EXTRACT_FIELD`**. It searches the message for the first field with a matching name. If one is found, it extracts the field value and returns it. It also returns the ordinal position of the field in the message, and optionally upcases the value.

Numbers and dates are common types of data elements, so the library has routines built on **`MDC_EXTRACT_FIELD`** and **`MDC_FIND_FIELD`** to convert the string values present in MDC messages to internal MCP data types. All numbers are returned to the caller as a double-precision floating value, which the caller can move to an integer, scaled value, or floating-point data item as necessary. Dates are represented in the MDC message in ISO 8601 format (**`yyyy-mm-dd`**), and are returned to the called as an eight-digit **`BINARY`** integer in **`yyyymmdd`** format.

That is the extent of support provided by the library for input message processing. If the TP requires additional data validation or conversion, it must extract the raw string values from the message and deal with them itself.

Library Output Message Formatting

- **MDC_FORMAT_HEADER**
  - Initializes output message area
  - Formats header trancode and msg-token fields
- **MDC_FORMAT_FIELD**
  - Appends field name and string value to message area
  - Trims spaces from string value, appends **FS**
  - Resizes message area as necessary
- **MDC_FORMAT_FIXED_FIELD**
  - Same as above, but no space trimming
- **MDC_FINISH_MSG**
  - Appends **ETX**, checksum, **EOT** to message area

MCP-4002 34

For output MDC messages, the library provides routines to build the message envelope and to append fields and records, one at a time, to the message data area.

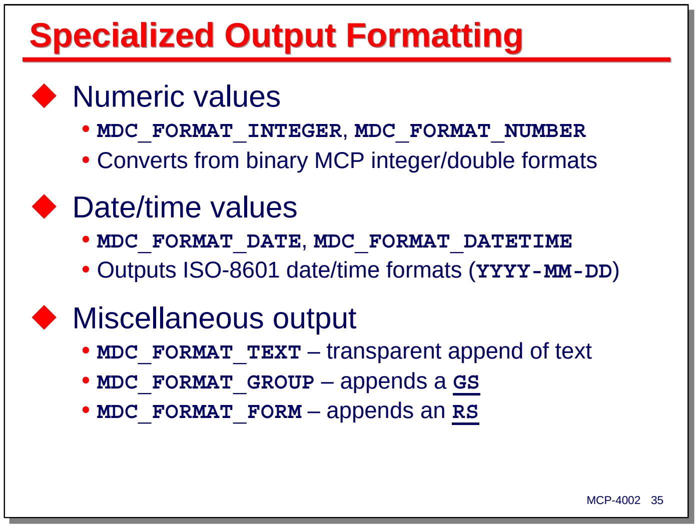
MDC_FORMAT_HEADER initializes the output message area and builds the MDC *header* from the *trancode* and *msg-token* fields. All of the output-formatting procedures use a zero-relative message offset parameter passed by the TP. This offset is updated with the current message length by each call.

MDC_FORMAT_FIELD appends a name/value pair for a field, along with the field's delimiting **FS** character. No conversion of the value is done, except that trailing spaces are trimmed. Each of the output formatting procedures checks the size of the output message area and resizes it if necessary to accommodate the new field.

MDC_FORMAT_FIXED_FIELD has the same function as **MDC_FORMAT_FIELD**, except that it does not trim trailing spaces from the value.

MDC_FINISH_MSG completes the formatting of the MDC message, appending the terminating delimiter and checksum characters. The offset it returns indicates the the final message length.

These procedures are the basis for output formatting. There are some specialized formatting procedures in addition to these, which are discussed on the next slide.



Since numbers and dates are common types of fields, the library has specialized output-formatting procedures to handle those types for the TP.

`MDC_FORMAT_INTEGER` converts a double-precision parameter to a signed string value and appends the field name and this formatted string to the output message.

`MDC_FORMAT_NUMBER` is similar to **`MDC_FORMAT_INTEGER`**, but formats the field value with a specified number of decimal places.

`MDC_FORMAT_DATE` converts an MCP **`BINARY`** parameter in **`yyyymmdd`** format to a string in ISO 8601 format and appends the field to the message.

`MDC_FORMAT_DATETIME` converts date and time parameters to an ISO 8601 timestamp (**`yyyy-mm-ddThh:mm:ss`**).

There are three miscellaneous output procedures which are used to handle less common situations.

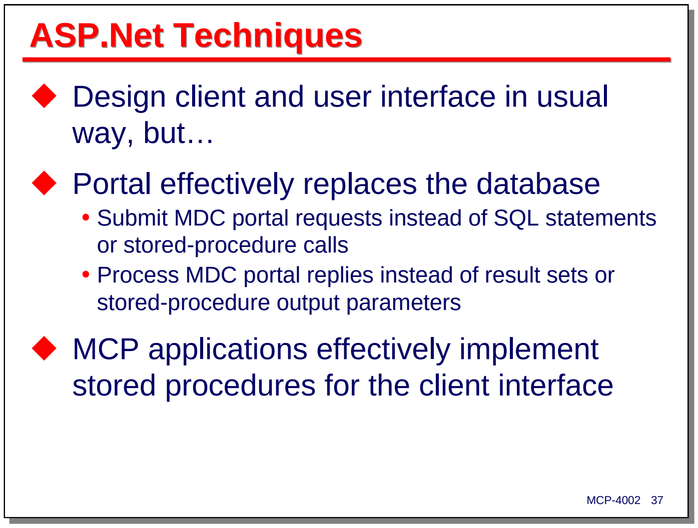
`MDC_FORMAT_TEXT` simply appends a string of text to the output message. It is oblivious to fields and delimiters, and appends exactly what the caller tells it. This is used for those rare cases when the standard output formatting functions are not sufficient.

`MDC_FORMAT_GROUP` simply appends the **`GS`** character that signals the start of a record-list in an MDC message.

`MDC_FORMAT_FORM` appends the **`RS`** character that signals the end of the field-list for a record in the record-list of a message.

Portal Application Coding Techniques

In this next section, I will try to describe some of the coding techniques used with portal applications.



On the client side, the applications are currently developed using ASP.Net technology. These are developed in a typical way – there is nothing special that needs to be done in terms of overall design or in the user interface to accommodate the portal.

What the portal does is effectively replace the access to a database, say a SQL Server database accessed through ADO.Net. Thus, instead of submitting SQL statements or stored-procedure calls to a database server, the ASP.Net application constructs MDC messages and sends them to the portal. Instead of receiving back a result set or stored-procedure output parameters, the application receives another MDC message.

One way to think of this is that the MCP side is effectively implementing stored procedures for the client interface. The difference is that these stored procedures are written in COBOL.

Submitting a Portal Request

```
Public Function GetShipToListByName(ByVal searchMode As String, _
                     ByVal nameFrag As String) As DataTable
   Dim request As New PortalMessage
   Dim reply As PortalMessage

   request.RoutingCode = String.Empty  '-- send to default agenda
   request.Trancode = "SHIPTOLIST.BYNAME"
   request.MsgToken = PortalMessage.FormatISODateTime(Now())
   request.AddField("USERID", userIDName)
   request.AddField("SEARCHMODE", searchMode)
   request.AddField("NAMEFRAG", nameFrag)

   reply = request.CallPortal(sock)
   If request.ErrorCode <> 0 Then
     Throw New Exception("Portal request error: " & _
                     request.ErrorText)
   ElseIf reply.Fields.GetInteger("*ERROR") <> 0 Then
     Throw New Exception("Portal reply error " & _
                     reply.Fields.GetInteger("*ERROR") & "=" & _
                     reply.Fields.GetString("*ERRMSG"))
   Else
     ... (process the reply message)
```

MCP-4002 38

This slide shows a fairly simple example of a VB.Net program instantiating a **`PortalMessage`** object, filling in some fields, sending it to the portal, and receiving a reply. The "**`sock`**" parameter to **`CallPortal()`** is a **`PortalSocket`** instance. In our current implementations, we allocate the socket object in the ASP.Net **`Session_Start()`** event (coded in **`Global.asax`**), and store it in the **`Session`** collection object. When the ASP.Net session terminates, the socket object is closed (disconnected from CCF) and deallocated.

Processing a Portal Reply

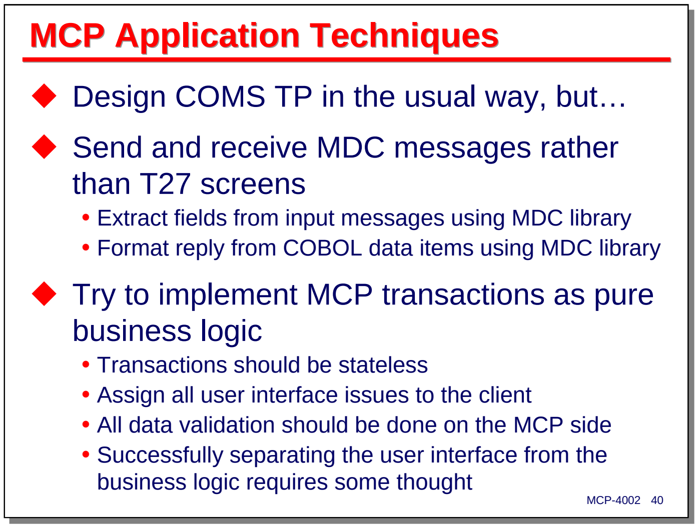
```
Dim reply As PortalMessage
Dim t As New DataTable("ShipToResultList")

t.Columns.Add("custNbr", GetType(String))
t.Columns.Add("custName", GetType(String))
t.Columns.Add("creditHold", GetType(Boolean))
t.Columns.Add("creditBal", GetType(Decimal))
t.Columns.Add("lastOrderDate", GetType(Date))
t.Columns.Add("city", GetType(String))
t.Columns.Add("postalCode", GetType(String))

For Each rec As PortalMessage.FieldList In reply.Records.Values
    t.Rows.Add(New Object() { _
        rec.GetString("CUSTNBR"), _
        rec.GetString("CUSTNAME"), _
        rec.GetBoolean("CREDITHOLDFLAG"), _
        rec.GetDecimal("CREDITBALANCE"), _
        rec.GetDate("LASTORDERDATE"), _
        rec.GetString("CITY"), _
        rec.GetString("POSTALCODE")})
Next
```

MCP-4002 39

This slide shows a simple example of processing a **`PortalMessage`** instance that is the result of a **`CallPortal()`** method. The method parses the MDC reply message, so at this point the **`Fields`** and **`Records`** collections in the **`PortalMessage`** instance have already been populated.

This example creates an ADO.Net **`DataTable`** object and then steps through the message's *record-list* collection to populate the table rows from the *field-lists* in each record of the *record-list*.

Building a **`DataTable`** object like this is a very useful thing to do, since that object can then be used by many of the ASP.Net data-aware controls, such as **`ListView`** and **`GridView`**.



In terms of application coding techniques on the MCP side, the COMS TPs can be designed in a typical way. As I've mentioned before, the big difference is that they will be processing MDC messages instead of T27 messages. That means that they will use the Algol MDC message-handling library to extract fields from input messages into COBOL data items, and then format fields in output messages from COBOL data items.

One of the challenges in designing portal applications, though, is to recognize that the user interface is elsewhere. The COMS TP needs to concentrate on business logic, and leave basic editing and presentation to the external client application. All transactions should be designed as stateless whenever possible. If conversational state needs to be maintained, that should be done on the client side as much as possible.

Regardless of what you do in terms of editing and validation on the client side, all validation critical to the functioning of the application must be done on the MCP side, even if that means redoing what the client has already done. The reason for this is that you never really know where the data is coming from and what kind of client you may be talking to. Data validation is a business-rule issue, and business rules need to stay on the MCP side.

Cleanly separating the user interface from the business logic is a new way of thinking for many traditional MCP on-line application programmers, and doing this successfully generally requires some thought – and not unusually a few false starts.

Receiving a Portal Request

```
RECEIVE CDI-COMS MESSAGE INTO FCM-COMS-MSG
MOVE SPACE TO W-TRANCODE, W-MESSAGE-ID
MOVE CDI-MSG-SIZE TO W-L
CALL "MDC_ASSEMBLE_MSG IN MDCLIB" USING
    FCM-COMS-MSG, W-L, W-TANK,
    W-MSG, W-MSG-SIZE, W-TRANCODE, W-RESULT
IF W-RESULT NOT = 1
  *> REPORT AN ERROR
ELSE
  CALL "MDC_PARSE_REPORT IN MDCLIB" USING
      W-MSG, W-MSG-SIZE, W-TRANCODE, W-MESSAGE-ID,
      W-FIELD-COUNT, W-FIELD-DICT,
      W-FORM-COUNT, W-FORM-DICT, W-RESULT
  IF W-RESULT NOT = ZERO
    *> REPORT AN ERROR
  ELSE
    EVALUATE W-TRANCODE
      WHEN "PING"
        PERFORM 1020-PORTAL-SEND-PONG THRU 1020-EXIT
      ...
    END-EVALUATE
    ...
```

MCP-4002 41

This slide shows the general logic involved in receiving a portal message and processing it. I have eliminated a lot of the checking of COMS header fields one would normally do here.

After receiving the message, we call **`MDC_ASSEMBLE_MSG`** to make sure we have a complete MDC message and validate the message envelope. As mentioned previously, much of this procedure's function is now superseded by CCF framing, but it is still useful to perform this step. If the result is not the value **`1`**, it means a complete MDC message was not received, which would normally be an error. We have yet to see that happen in our environment.

Next, the TP calls **`MDC_PARSE_MSG`** to extract the MDC *header* fields and build the main *field-list* and *record-list* dictionaries. A non-zero result here indicates the message is malformed.

The *trancode* field from the MDC *header* is typically used to identify the type of transaction, so a typical approach is to use that to select the specific routine within the TP that will process the message.

Extracting Request Fields

```
MOVE "SEARCHMODE" TO W-FIELD-NAME
COMPUTE W-L = FUNCTION LENGTH (WRQ-SEARCH-MODE)
CALL "MDC_FIND_FIELD IN MDCLIB" USING
    W-MSG, W-FIELD-NAME, W-FIELD-DICT, W-INDEX,
    WRQ-SEARCH-MODE, W-L, W-UPCASE-TEXT
IF W-INDEX = ZERO
  MOVE "P" TO WRQ-SEARCH-MODE
END-IF

MOVE "OEJOBSEQNBR" TO W-FIELD-NAME
CALL "MDC_FIND_NUMBER IN MDCLIB" USING
    W-MSG, W-FIELD-NAME, W-FIELD-DICT, W-INDEX,
    W-DOUBLE-VAL, W-RESULT
IF W-INDEX = ZERO
  MOVE ZERO TO OEFORDM-KEY-OEJOBSEQNBR
ELSE
  MOVE W-DOUBLE-VAL TO OEFORDM-KEY-OEJOBSEQNBR
  IF W-RESULT NOT = ZERO
    MOVE 201 TO W-PORTAL-ERROR-CODE
    MOVE "Job seq number not numeric" TO W-PORTAL-ERROR-TEXT
  END-IF
END-IF
```

MCP-4002 42

This slide shows a couple of typical examples that extract values from a message and place them in COBOL data items for the TP to use. I tend to go after specific fields by name, so use the **`MDC_FIND`**… versions of the input field routines a lot more often than the **`MDC_EXTRACT`**… versions.

Formatting a Response Message

```
CALL "MDC_FORMAT_HEADER IN MDCLIB" USING
    W-TRANCODE, W-MESSAGE-ID, FCR-COMS-REPLY, W-OUT-SIZE

MOVE "NAMEFRAG" TO W-FIELD-NAME
COMPUTE W-L = WRQ-NAME-FRAG-LENGTH
CALL "MDC_FORMAT_FIELD IN MDCLIB" USING
    W-FIELD-NAME, WRQ-NAME-FRAG, W-L,
    FCR-COMS-REPLY, W-OUT-SIZE

MOVE "CUSTORDERCOUNT" TO W-FIELD-NAME
MOVE WFU-ORDER-COUNT TO W-DOUBLE-VAL
CALL "MDC_FORMAT_INTEGER IN MDCLIB" USING
    W-FIELD-NAME, W-DOUBLE-VAL, FCR-COMS-REPLY, W-OUT-SIZE

CALL "MDC_FINISH_MSG IN MDCLIB" USING
    FCR-COMS-REPLY, W-OUT-SIZE
MOVE W-OUT-SIZE TO CDO-MSG-SIZE
MOVE ZERO TO CDO-DESTINATION
SEND CDO-COMS FROM FCR-COMS-REPLY WITH EMI BEFORE 0 LINES.
```

MCP-4002 43

This slide shows some simple examples of output message formatting. We always call **MDC_FORMAT_HEADER** to initialize the output are and message building process. **W-OUT-SIZE** is the current message offset, and is passed to each of the output-formatting procedures. At the end, we call **MDC_FINISH_MSG** to finalize the message envelope, and then do the COMS **SEND**.

Error Communication

- Errors detected in the business logic must be passed back to the client
- Many ways to do this – none convenient
- Decided on a two-level convention
- Global error for the transaction:
  - Field "**`*ERROR`**" for a numeric error code
  - Field "**`*ERRMSG`**" for a textual error message
- Field-level errors
  - Prefix field name with "**`*`**" for that field's error message
  - E.g., error text for "**`CUSTNBR`**" will be in "**`*CUSTNBR`**"
  - Error fields sent only for fields that have errors

MCP-4002 44

When the business logic on the MCP side detects errors, those errors must generally be passed back to the client. Not only do we want to say what the error was, it would be nice to identify the specified field(s) that are affected. With that information, the client application can do user-friendly highlighting of errors.

There are many ways to handle field-level error identification – all of them are a pain. I finally decided on a two-level convention.

There is a global error for the entire message, consisting of a numeric error code returned in the "**`*ERROR`**" field and a corresponding error message returned in the "**`*ERRMSG`**" field.

Field-level errors are indicated by appending an error-message field to the message. The convention is that if there is an error in the field "**`CUSTNBR`**", the error message will be in a field named "**`*CUSTNBR`**". These "**`*`**" fields are only sent if the corresponding field has an error. The client application looks for these error fields and uses their presence to highlight individual fields in the end-user interface.

Lessons Learned

In the final section of this presentation, I will try to summarize some of the things we've learned thus far from our experience with this portal implementation.

Lessons Learned

- The Portal was easy
- Portal-based apps are a lot harder
- MDC format is okay, but should have done a more robust one, e.g., JSON
- Extracting and formatting MDC fields in COBOL is really, really tedious
  - Lack of parameterization in COBOL is the problem
  - Almost makes the predefined, fixed-field record approach look worthwhile
  - Self-defining fields are still better for long-term maintainability, though

MCP-4002 46

The first lesson was that building the portal infrastructure was easy. Building applications to use the portal was a lot harder. This was true on both ends of the portal.

The MDC message format was adequate, and thus far has been able to handle everything we've needed. In retrospect, though, I now wish I had chosen a more robust format. If I were to do this over, I would implement a JSON parsing and formatting library for the MCP, especially since ASP.Net already has considerable JSON functionality built in.

ASP.Net has a significant learning curve, and building good user interfaces is challenging. Dealing with MDC fields and records on the DotNet side, however, was relatively easy and convenient.

The MCP side was not as challenging, but dealing with the MDC messages was a very different experience, and where the messages had a large number of fields, the coding to extract and format messages, even with the support of the Algol library, was really tedious. The problem is that COBOL simply does not have very good parameterization capabilities, and every field requires multiple lines of coding to extract or format, so the result was lots (and lots) of repetitive, linear coding.

All of that MDC message coding almost makes the use of fixed-length fields in fixed-format records look worthwhile, but fixed-format records require much closer synchronization of the client and server implementations, so I'm still convinced that flexible-format messages with named fields are better for long-term application maintainability.

Lessons Learned, continued

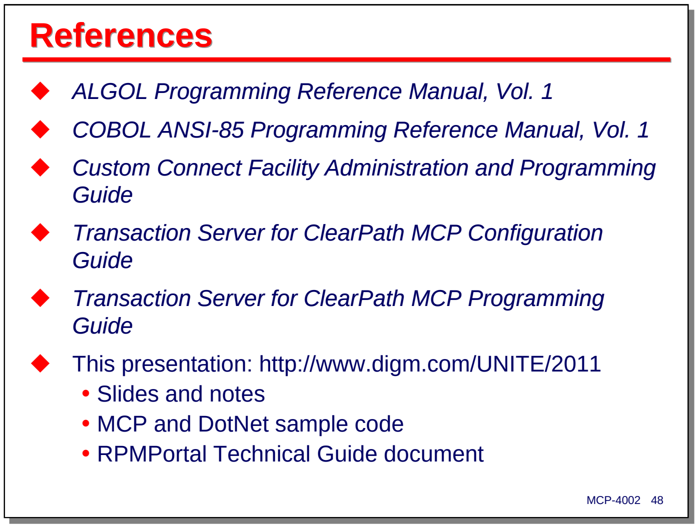
- **PortalSocket** and **PortalMessage** worked well on the DotNet side
- **TCPClient** does not tolerate errors well
  - Timeouts force you to close and reopen the connection
  - Fortunately this doesn't happen often
- Users love the richer web-based interfaces
  - But designing a good GUI is not easy
  - You really have to work at making it nice for the user
  - Not every programmer can do this well
- We probably won't be doing any new green-screen interfaces going forward

MCP-4002 47

The VB.Net classes for dealing with the CCF protocol and MDC message format worked very well on the DotNet side. The PortalSocket class could probably use some better network error handling, but the network path between the MCP and IIS systems is so short that we have not had any problems in this area.

One thing about the DotNet environment that proved to be a disappointment was that the **TCPClient** sockets do not tolerate errors well. If the socket timed out waiting for a reply from the MCP, there was no way to recover the connection – **TCPClient** refused any further attempts to receive on that connection. The only solution we've found is to close the connection and open a new one.

One very gratifying outcome of this experience was that the end users absolutely love the richer web-based interfaces we've built. After years of working with bad green-screen interfaces, they are just about beside themselves with pleasure at how the new interfaces work. It took a lot of effort to make them smile like this, though – designing good GUI/web interfaces is not easy. It takes an entirely different approach to do this well, and not every programmer can be good at it.



This slide lists the major Unisys documentation references that are relevant to our portal implementation.

In addition, all of the infrastructure software we build for the portal, the technical documentation for the portal APIs, and a couple of small sample applications are available in the materials accompanying this presentation. They can be downloaded from our web site at the URL shown on the slide.

End

You Portal

2011 UNITE Conference

Session MCP-4002