Engineering Test Software (ETS) Operation

This document provides final product specifications. VIA Telecom reserves the right to make changes to this product at any time without notice.

This document contains proprietary information of VIA Telecom, Inc. The information contained herein is not to be used by or disclosed to third parties without the express written permission of an officer of VIA Telecom, Inc.

This document describes VIA Telecom's Engineering Test Software and will remain the official reference source for all revisions/releases of this product until rescinded by an update.

VIA Telecom, Inc. reserves the right to make changes to any product herein at any time without notice. VIA Telecom does not assume any responsibility or liability arising out of the application or use of any product described herein, except as expressly agreed to in writing by VIA Telecom; nor does the purchase or use of a product from VIA Telecom convey a license under any patent rights, copyrights, trademark rights, or any other of the intellectual property rights of VIA Telecom or third parties.

The VIA Telecom logo design and VIA TELECOM are trademarks owned by the company. Microsoft, Windows, Windows NT, Visual Basic, and Visual C++ are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. LabVIEW is a trademark of National Instruments Corporation. CommLib is a trademark of sysFire LLC. All other brand and product names may be trademarks of their respective companies.

VIA Telecom products are not intended for use in life-support appliances, devices, or systems. Use of any VIA Telecom product in such applications without written consent of the appropriate VIA Telecom officer is prohibited.

VIA Telecom is certified to the ISO 9001:2000 quality management system international standard.

VIA Telecom, Inc.
3390 Carmel Mountain Road
San Diego, CA 92121, U.S.A.
Tel: 858.350.5560
Internet: www.via-telecom.com

January 2006

Engineering Test Software Operation

Revision:	32
Status:	Draft
Author:	Multiple
Function:	VIA Telecom, Inc.
Path and Filename:	C:\CM\CBPTOOLS\Docs\ctm_ets_overview.doc
File Type:	Microsoft Word 2000
Version 32 Release Date:	May 10, 2007

Revision History

Rev	Status*	Date	Author	Reason for change/description
32	Draft	5/10/2007	C. Lewis	Corrected the description of non-bitpacked bit fields.
31	Draft	1/24/2007	C. Lewis	Added string and numeric field password description.
30	Draft	2/21/2006	C.Lewis	Added the Download.dll documentation.
29	Draft	1/19/2006	C. Lewis M. Liu D. Orloff	Added ViaHub description, updated the EtsMain command line options, added the ViaBinary interface to the ViaIf.dll and the EtsIf.ocx interface. Added BINLOG and GETDEF sections. Added conditionally and invisible definition support descriptions.
28	Approved	9/19/2005	Roy Davis, Bryan Yang	Rev 26_D approved for release as Rev 28_A
27	Draft	09/02/2005	C. Lewis M. Liu	Added ViaIf.dll description and added explanations to the Get Command, Error and Response interface functions. Added USB support
26	Approved	5/24/2005	Roy Davis, Bryan Yang	Rev 25_P approved for release as Rev 26_A
25	Pending	05/02/2005	M. Liu	Added the description of ViaCrypt utility and binary EtsMsg.bin file
24	Approved	1/13/2005	R. Smith J. Miller	Rev 23_P approved for release as Rev 24_A.
23	Pending	01/06/2005	Chuck Lewis D. Orloff	Made generic to all CBP products Added section on DBM Templates and AutoRspQ interface Removed ETS Visual Basic Interface Updated sections for adding Commands, Traces, Spies, and Faults Miscellaneous minor updates
22	Draft	10/06/2004	H. Nguyen	Added Register OCX and updated External EXEs
21	Draft	08/17/2004	H. Nguyen	Added Auto Config Loading when phone is detected.
20	Draft	06/18/2004	H. Nguyen	Added External EXEs to ETS GUI
19	Draft	2/5/2004	C. Lewis	Added the Lock Config File command line option
18	Draft	10/13/2003	D. Schaeffer	Convert to customer template
17	Approved	7/15/2003	P. LaBarbera R. Smith S. O'Leary	Version 16_D approved for release as version 17_A
16	Draft	6/12/03	C. Lewis	Added the description of "Default ETS Options" section.
15	Draft	5/28/03	C. Lewis	Added descriptions of the following new options. Disabled commands, hidden fields, read only fields and startup command options. Updated with changes as recommended in review. Added Fault Log section description. Moved section 5 to the end.
14	Draft	4/14/03	C. Lewis	Updated the command line options and added the COUNT_OF option to expression definitions.
12	Draft	3/10/03	C. Lewis	Added bitpacked arrays. Updated documentation for Lite message definitions. Added description of the "Save

Rev	Status*	Date	Author	Reason for change/description
32	Draft	5/10/2007	C. Lewis	Corrected the description of non-bitpacked bit fields.
31	Draft	1/24/2007	C. Lewis	Added string and numeric field password description.
30	Draft	2/21/2006	C.Lewis	Added the Download.dll documentation.
29	Draft	1/19/2006	C. Lewis M. Liu D. Orloff	Added ViaHub description, updated the EtsMain command line options, added the ViaBinary interface to the ViaIf.dll and the EtsIf.ocx interface. Added BINLOG and GETDEF sections. Added conditionally and invisible definition support descriptions.
28	Approved	9/19/2005	Roy Davis, Bryan Yang	Rev 26_D approved for release as Rev 28_A
27	Draft	09/02/2005	C. Lewis M. Liu	Added ViaIf.dll description and added explanations to the Get Command, Error and Response interface functions. Added USB support
26	Approved	5/24/2005	Roy Davis, Bryan Yang	Rev 25_P approved for release as Rev 26_A
25	Pending	05/02/2005	M. Liu	Added the description of ViaCrypt utility and binary EtsMsg.bin file
				Definitions" menu item. Added Decode CDMA Message description. Added Reset Board implementation. Added Binary logging feature.
11	Draft	2/14/03	D. Schaeffer	Made file "read-only recommended"
10	Draft	1/06/03	D. Schaeffer	Converted to new VIA Telecom template
9	Draft	12/20/02	W. Tang	Added globalVar section and #pragma section.
8	Draft	10/24/02	H. Nguyen	Added Command Function section.
7	Draft	10/10/02	W. Tang	Updated padding field usage and other minor changes.
6	Draft	9/17/02	S.Rodenbaugh	Major updates of all message definition sections.
5	Draft	8/02/02	A. Monteiro	Added a Section on Commands, Spies, Traces, Faults. This section was ported from a stand alone document written by Victor Nguyen for CBP3
4	Draft	6/22/01	H.Nguyen	Updated ActiveX section with functions.
3	Draft	5/29/01	E. Tarolli	Updated options and command line entries
2	Draft	7/8/00	K. Kim	Changed document name from tm_etssdd_cbp3.doc to tm_ets_overview_cbp3.doc.
1	Draft	12/29/99	K. Kim	Original version.

* Statuses include: D: Draft, P: Pending review/frozen, U: Update needed, A: Approved

Ver 28_A Approved by:	Roy Davis, Bryan Yang	**Date:**	Sept 19, 2005
Ver 26_A Approved by:	Roy Davis, Bryan Yang	**Date:**	May 24, 2005
Ver 24_A Approved by:	R. Smith, J. Miller	**Date:**	January 13, 2005
Ver 16_A Approved by:	P. LaBarbera, R. Smith, S. O'Leary	**Date:**	Sept 15, 2003

Table of Contents

1 Introduction 1
1.1 Acronyms 1
2 Interfaces 2
2.1 Communication Interface 2
3 Message Definition Files 3
3.1 File Usage Syntax 3
3.1.1 Include Files 3
3.1.2 Define Statements 3
3.1.3 Syntax Checking Pragma 3
3.1.3.1 Syntax_dupname 4
3.1.3.2 #pragma push(syntax_dupname) 4
3.1.3.3 #pragma pop(syntax_dupname) 4
3.1.3.4 #pragma disable(syntax_dupname) 4
3.1.3.5 Definitions 4
3.2 Message and Record Definitions 5
3.2.1 Message Naming Conventions 5
3.2.1.1 Naming of Submessages in ETS Base Message 5
3.2.1.2 Naming of Submessages in ETS Named Submessages 6
3.2.1.3 Naming of Submessages in Non-ETS Named Submessages 7
3.2.2 Message and Record Parameters 8
3.2.2.1 BitPack 8
3.2.2.2 PrintOnly 8
3.2.2.3 TruncateBits=# 8
3.2.3 BitPack Message and Record Format 8
3.2.4 Message Examples 9
3.2.4.1 Example of Command/Response Message 9
3.2.4.2 Example of Response Only Message 9
3.2.4.3 Example of Command/Response Sub-Message 9
3.2.5 Record Definitions and Examples 10
3.2.6 GlobalVar Definition 11
3.2.6.1 Associating a field with a global variable 11
3.2.6.2 Using “exist” Condition Based on a Global Variable for a Field 11
3.2.6.3 Using “exist” Condition Based on a Global Variable for a Group of Fields 11
3.2.7 Conditional Definitions 11
3.2.8 Invisible Definitions 12
3.2.9 Startup Commands Definition 12
3.2.10 Extra Log Window Definition 13
3.2.11 Fault Log Definition 13
3.2.12 Default ETS Options 13
3.3 Field Definitions 14
3.3.1 Bit Fields 14

3.3.1.1 BitPack Bit Fields ... 14
3.3.1.2 Non-BitPack Bit Fields ... 14
3.3.1.3 Bit Field Parameter Differences ... 15
3.3.2 Numeric Fields ... 15
3.3.2.1 Password parameter ... 15
3.3.3 Padding Fields ... 16
3.3.4 Array Fields ... 16
3.3.5 Record Fields ... 17
3.3.6 String Fields ... 18
3.3.6.1 Fixed Length String ... 18
3.3.6.2 Variable Length String ... 18
3.3.6.3 Password parameter ... 19
3.3.7 Union Fields ... 19
3.3.7.1 Union Field Identifier ... 21
3.3.7.2 Union Field Examples ... 22
3.3.8 Enumerations ... 23
3.3.8.1 Enumerations: Short Form ... 23
3.3.8.2 Long Form ... 23
3.3.8.3 Enumeration For Unions ... 24
3.3.9 Field Parameters ... 25
3.3.9.1 Field Parameter: base ... 26
3.3.9.2 Field Parameter: byteorder ... 27
3.3.9.3 Field Parameter: bytes ... 27
3.3.9.4 Field Parameter: columns ... 27
3.3.9.5 Field Parameter: ConsumeBits ... 27
3.3.9.6 Field Parameter: ConsumeBytes ... 28
3.3.9.7 Field Parameter: count ... 28
3.3.9.8 Field Parameter: default ... 28
3.3.9.9 Field Parameter: enum ... 29
3.3.9.10 Field Parameter: exist ... 29
3.3.9.11 Field Parameter: fields ... 30
3.3.9.12 Field Parameter: Format ... 30
3.3.9.13 Field Parameter: indent ... 30
3.3.9.14 Field Parameter: length ... 30
3.3.9.15 Field Parameter: log10 ... 31
3.3.9.16 Field Parameter: logScale ... 31
3.3.9.17 Field Parameter: maxsize ... 31
3.3.9.18 Field Parameter: min and max ... 31
3.3.9.19 Field Parameter: minbytes ... 32
3.3.9.20 Field Parameter: noexist ... 32
3.3.9.21 Field Parameter: offset ... 32
3.3.9.22 Field Parameter: pos ... 33
3.3.9.23 Field Parameter: rows ... 33
3.3.9.24 Field Parameter: scale ... 34
3.3.9.25 Field Parameter: set ... 34

3.3.9.26 Field Parameter: size ... 34
3.3.9.27 Field Parameter: type ... 35
3.3.9.28 Field Parameter: Hidden ... 36
3.3.9.29 Field Parameter: ReadOnly ... 36
3.3.10 Expression ... 36
3.3.10.1 Field Names as Entities ... 37
3.3.10.2 COUNT_OF ... 37
4 Utilities ... 38
4.1 Include File Generator ... 38
4.1.1 Enumerated List Generation ... 38
4.1.2 String List Generation ... 38
4.2 User Interface ... 39
4.2.1 Ets Options Dialog ... 39
4.2.2 Logging ... 42
4.2.2.1 Extra Log Windows ... 42
4.2.3 Menus and Dialogs ... 42
4.3 Script Utility ... 42
4.4 OLE Automation Interface ... 43
4.5 Knobs, Dials, Graphs ... 43
4.6 ViaCrypt Utility ... 43
5 Modules ... 45
5.1 ETS Main ... 45
5.1.1 Command Line Options ... 46
5.1.1.1 Configuration File ... 46
5.1.1.2 Syntax Check ... 46
5.1.1.3 Communication Port ... 47
5.1.1.4 Window Mode ... 47
5.1.1.5 Defines ... 47
5.1.1.6 Save Definitions ... 47
5.1.1.7 Lock Configuration ... 47
5.1.2 Lite Mode Of Operation ... 47
5.1.3 Save Definitions ... 48
5.1.4 Message Control Library ... 48
5.1.4.1 Message Definition Processing ... 48
5.1.4.2 Message Object ... 48
5.1.4.3 Menu Handler ... 48
5.1.4.4 Log Processing ... 49
5.1.4.5 Trace Management ... 49
5.1.5 Communications Interface ... 49
5.1.5.1 Serial Interface ... 50
5.1.5.2 Message Encapsulation ... 50
5.1.6 Message Router ... 50
5.1.7 OLE Automation Interface ... 50
5.1.8 Specialized Message Handlers ... 50

5.1.8.1 Loopback Test 50
5.1.8.2 Printf Message Handler 50
5.1.8.3 Trace Message Handler 50
5.1.8.4 Database Template Handling 51
5.1.9 Decode CDMA Message 51
5.1.10 Reset Board 51
5.1.11 Binary Logging and Playback 52
5.1.12 Flash Download 52
5.1.13 External Application Support 52
5.1.13.1 External EXE Definition and Register OCX 52
5.1.13.2 External Application Interface 53
5.1.14 Auto Configuration Loading 53
5.2 Script Utility 53
5.3 Control Panels 53
5.4 Test Drivers 53
5.5 ETS ActiveX Interface 54
5.5.1 Overview 54
5.5.2 Initialization 54
5.5.3 Auto Script Support 54
5.5.4 Received Message Flow 55
5.5.4.1 Asynchronous Message Flow Functions 55
5.5.4.2 Message Event Functions 55
5.5.5 Field Data Functions 56
5.5.6 Array Field Functions 56
5.5.7 File Functions 57
5.5.8 Command Function 58
5.5.9 Serial Communication Functions 58
5.5.10 ETS Control Functions 58
5.5.11 AutoRspQ 58
5.5.11.1 Function Create(pFilter As String, pName As String) As Long 58
5.5.11.2 Function Destroy() As Boolean 58
5.5.11.3 Function Enable(Enable As Boolean) As Boolean 59
5.5.11.4 Function GetAllResponses() As String 59
5.5.11.5 Function GetOverRunCount() As Long 59
5.5.11.6 Function GetResponse() As String 59
5.5.11.7 Sub SetSize(Size As Long) 59
5.6 VIAIF Dynamic Link Library 59
5.6.1 ViaSetup Interface 59
5.6.1.1 LoadDefinitions 59
5.6.1.2 OpenCommInterface 59
5.6.1.3 UnloadDefinitions 60
5.6.1.4 CloseCommInterface 60
5.6.1.5 GetCommStatus 60
5.6.1.6 GetCommStatistics 60
5.6.1.7 ResetCommStatistics 60

5.6.1.8 GetAutoRspQ 60
5.6.1.9 RawHex 60
5.6.1.10 ResetPolarity 61
5.6.1.11 ResetBoard 61
5.6.1.12 SendDeferredCommand 61
5.6.2 ViaBinary Interface 61
5.6.2.1 SendMessage 61
5.6.2.2 GetMessage 61
5.6.2.3 Clear Filter 61
5.6.2.4 AddToFilter 61
5.6.2.5 ClearRxQueue 62
5.7 ViaHub 62
5.7.1 USB Connection Management 62
5.7.2 Application Access 62
5.7.3 Command Line Options 62
5.7.3.1 /ClearUsbPorts 62
5.7.3.2 /Delay 62
5.8 ETS Message Commands 62
5.8.1 BINLOG Messages 62
5.8.1.1 Create Binary Log File 62
5.8.1.2 Close Binary Log File 63
5.8.1.3 Get Status of Binary Log File 63
5.8.1.4 Playback Binary Log File 63
5.8.2 GETDEF 64
5.8.2.1 Get Definition Entry Count 64
5.8.2.2 Get Definition Entry Value by Entry Number 64
5.8.2.3 Get Definition Entry Value by Entry Name 65
5.9 Download DLL 65
5.9.1 IDownloader interface 66
5.9.1.1 Methods 66
5.9.1.2 Properties 68
5.9.1.3 IDownloaderEvents Interface 68
6 Adding Commands, Traces, Spies, Faults 70
6.1 ETS Commands 70
6.1.1 Adding an ETS Command 70
6.1.1.1 Modify ETS Message Id and Message Body Definition Text Files 70
6.1.1.2 Modify CBP CP Software Files 70
6.1.2 Executing an ETS Command 71
6.2 ETS Traces 71
6.2.1 Adding an ETS Trace 71
6.2.1.1 Modify ETS Message Id and Message Body Definition Text files 72
6.2.1.2 Modify CBP CP Software Files 72
6.2.2 Enabling an ETS Trace 73
6.3 ETS Spies 73
6.3.1 Adding an ETS Spy 73

6.3.1.1 Modify ETS Message Id and Message Body Definition Text Files ... 73
6.3.1.2 Modify CBP CP Software Files ... 73
6.3.2 Enabling an ETS Spy ... 74
6.4 ETS Faults ... 74
6.4.1 Adding an ETS Fault ... 74
6.4.1.1 Modify ETS File mmi/mmi_errs.txt ... 74
6.4.1.2 Modify CBP CP Files ... 74
6.4.2 Results of ETS Faults ... 75
7 Rev 27_D Review Information ... 76
8 Rev 25_D Review Information ... 77
9 Ver 23_P Design Review Information ... 78
10 Ver 14_D Design Review Information ... 79

List of Figures

Figure 1. ETS Interfaces 2
Figure 2. Naming Conventions for ETS Base Message 6
Figure 3. Naming Conventions for ETS Named Submessages 7
Figure 4. Naming Convention: Non-ETS Named Submessages 8
Figure 5. Message with Union Field 20
Figure 6. Message with Union Submessage Substitution 21
Figure 7. Options Dialog 40
Figure 8. ETS Sub-Module Interfaces 45
Figure 9. ETS Main Sub-Modules 46
Figure 10. Communications Interface 49

List of Tables

Table 1. Differences in Bit Field Parameters 15
Table 2. Field Types Versus Field Parameters 25
Table 3. Default Values for Min and Max Field Parameters 32
Table 4. Ets Options Dialog Selections 41

1 Introduction

This document describes the operation of the VIA Telecom Engineering Test Software (ETS).

ETS is a Windows®-based PC software tool that allows designers to observe, control, and record in real-time the behavior of CMDA handsets that use all versions of the VIA Telecom CDMA Baseband Processor (CBP) products. Designers can use ETS in all phases of handset design, including development, field test, and production. Features of ETS include real-time memory peak/poke functions for the CP and DSPs and the ability to capture detailed software interaction information in real-time or in a log file.

ETS supports a Windows-COM interface that allows further customer PC applications to be developed on top of ETS.

1.1 Acronyms

The following acronyms are used in this manual:

CBP	CDMA Baseband Processor
CO	Command Only
CP	Control Processor
CR	Command Response
CRM	Command Response Message
CT	Command Trace
DSP	Digital Signal Processor
DTR	Data Terminal Ready
ETS	Engineering Test Set
ICD	Interface Control Document
LSB	Least Significant Bit
MCL	Message Control Library
MMI	Man-Machine Interface
OAI	OLE Automation Interface
UART	Universal Asynchronous Receiver Transmitter
VB	Visual Basic®

2 Interfaces

ETS is a Windows-based PC program that provides designers with the capability to observe, control, and record the behavior of the CBP Control Processor (CP) and Digital Signal Processors (DSP).

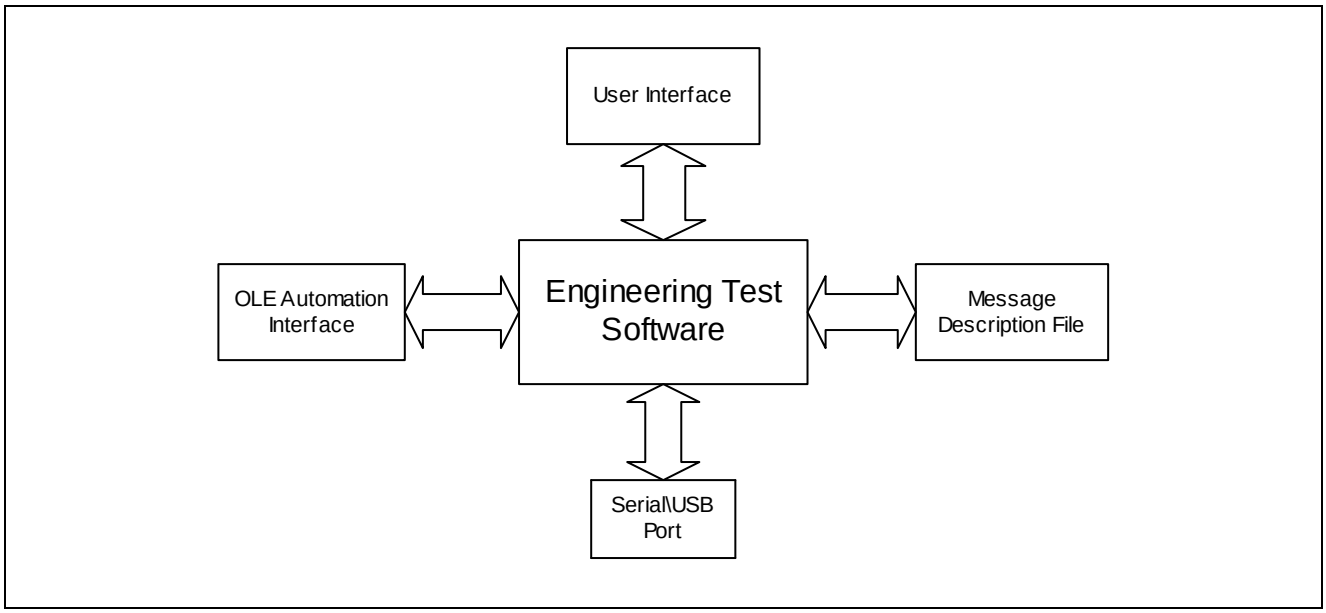


Figure 1. ETS Interfaces

2.1 Communication Interface

The ETS communicates with the CP through a serial interface. Message exchanges fall into the following four categories:

1. **Command Response** - A command message is sent from the ETS to the CP, and the CP immediately returns a response message to the ETS.
2. **Command Trace** - A Trace command message is sent from the ETS to the CP to enable or disable the transmission of a specific Trace message. When enabled, Trace messages are sent from the CP to the ETS when the CP software determines a need. The command messages of this category all have the same format, which is the message ID followed by a single boolean byte to enable or disable the trace.
3. **Command Only** - A command message is sent from the ETS to the CP. No acknowledgment is returned.
4. **Response** - A response message is sent from the CP to the ETS without any solicitation from the ETS.

Notes:

1. A dialog window is provided to configure the serial interface.
2. Multiple instances of the ETS can be running concurrently on several different communication ports.
3. There is no high level flow control.

3 Message Definition Files

When ETS is loaded it reads the EtsMsg.txt file and all of the referenced include files. It contains descriptions of each of the messages to be exchanged with the CP as well as various configurations items. A fatal error occurs if an invalid message definition exists in the file. A list defining each message will be in the file, in addition to definitions of the fields within each message.

An encrypted form of the ETS message definition file can also be used. The filename is EtsMsg.bin. It is used in the same manner as EtsMsg.txt. Loading EtsMsg.bin will take precedence over EtsMsg.txt if they exist in the same directory. Refer to section 4.6, “ViaCrypt Utility,” for details on how to create an encrypted definition file.

3.1 File Usage Syntax

3.1.1 Include Files

Multiple message definition files can be incorporated to better organize the message definitions. Include files are specified outside of the definitions and cannot have circular references. The following format is used:

```
Include <filename>
```

3.1.2 Define Statements

A "C Language" like ifdef syntax is used to control the interpretation of the definitions. The definitions themselves are specified using the "Defines" dialog box from the *File/Defines…* menu item. Definition strings must not contain space or tab characters. The following statements are allowed:

```
#ifdef <parameters>| Comments|
#ifndef <parameters>| Comments|
#else| Comments|
#else ifdef <parameters>| Comments|
#else ifndef <parameters>| Comments|
#endif| Comments|
#define <parameter>
```

These can be nested in the same manner that is done in the C language. Care should be taken since the effect of the statements cross into and out of include files.

Parameters are either a single definition value or a boolean statement. For example:

```
#ifdef ADEFINITION This is a comment for a single definition.
```

Boolean statements must be surrounded by parentheses. Examples are:

```
#ifdef (FIRST & SECOND) This is a comment for a boolean statement
#ifdef (FIRST | SECOND)
#ifdef (~FIRST | SECOND)
#ifdef (FIRST | (SECOND & THIRD))
```

Comments can follow the parameters if desired. Either a space character or a trailing parenthesis will separate the parameters from the comment.

3.1.3 Syntax Checking Pragma

There is a mode to run ETS called full syntax check mode, with option –s or /s. Under full syntax checking mode, ETS will check for syntax errors in ETS text files. One of the syntax errors is field name duplication error.

Example:

```
Begin Message, Msg1
      Field1, uint8, indent = 2
      Field2, uint16, indent = 2
      Field1, uint8, indent = 2
End
```

In this message, *Field1* is used twice, and it will be reported as a syntax error. But due to legacy reasons, some of the AMPS messages and records have used duplicated field names, and it is hard or impossible to change them. The following syntax checking flag and pragmas are introduced to overcome this problem.

3.1.3.1 Syntax_dupname

This is a boolean flag used under full syntax checking. At the very beginning, it is set as true, which means ETS will check for duplicated field names, and if a duplicate is found, ETS will report it as a syntax error.

3.1.3.2 #pragma push(syntax_dupname)

This pragma stores the boolean flag *syntax_dupname* to a stack.

3.1.3.3 #pragma pop(syntax_dupname)

This pragma restores the boolean flag *syntax_dupname* from a stack.

3.1.3.4 #pragma disable(syntax_dupname)

This pragma sets the boolean flag *syntax_dupname* as false.

Example:

The following is an example of how to use the pragmas described above:

```
#pragma push(syntax_dupname)
#pragma disable(syntax_dupnam)


Begin Message, Msg1
      Field1, uint8, indent = 2
      Field2, uint16, indent = 2
      Field1, uint8, indent = 2
End
#pragma pop(syntax_dupname)
```

In this example, ETS will not report the duplicated field names as a syntax error for Message Msg1.

3.1.3.5 Definitions

The message description file contains sets of definitions. The following syntax rules apply:

- Leading and trailing spaces are ignored.
- Lines that begin with a semicolon are ignored.
- All definitions start with a *Begin* line and end with an *End* line.
- The *Begin* line specifies the type of definition. The valid types are "Message," "Enum," and "Record."
- Definitions can contain *Entry* and *Help* lines.

- Lines that start with the word *Help* contain one-line descriptions that are associated with either the definition as a whole or with a single entry. A Help line applies to the definition as a whole if it immediately follows the *Begin* line.
- A line that starts with PrintFormat and follows the begin statement before the first entry defines a print format for the definition as a whole.
- An Entry line contains a name and an optional set of parameters. Its format is "<name>, <parameters>". The Parameters are specified in a comma-separated list.

Note: Help strings and PrintFormat strings are optional.

Example:

```
Begin <Definition Type>, <Name>|, Option1….|
      Help <help string for definition>
      PrintFormat, <printf format string for definition>
      <Entry Name>|,parameters|
      help <help string for entry>
End
```

3.2 Message and Record Definitions

The following is an example of a message definition. It is also the base definition of the ETS message structure.

```
Begin Message, ETS
      Help ETS Message frame
      Id, uint16, base=enum:ETS Id
      help Message Id
      Message, union ,enum=Id
      Help Message Data
End
```

3.2.1 Message Naming Conventions

ETS message naming conventions have the following variations, and each variation is described in the following paragraphs:

- Submessages contained in the ETS Message ID Base Message
- Submessages contained in an ETS Named Submessage
- Submessages contained in a non-ETS Named Submessage

3.2.1.1 Naming of Submessages in ETS Base Message

ETS has a set of rules that govern message naming. All messages start from the “ETS Message ID Definition” message shown in Figure 2. The ETS message is composed of an ID field plus one of the many messages represented by the union (Message). The union is based on the “ETS Id” enumeration. As Figure 2 shows, there is an implied relationship between the ETS message, the ETS Id enumeration, and the name of the message.

The layout of the message names is as follows:

Form: “ETS“ + “ “ + Enumeration Value + “ “ + (“Cmd” or “Rsp” or nothing)

Rules:

1. All messages start with ETS.
2. The “Enumeration Value” is the actual enumeration item name found in the ETS Id enumeration.
3. The “Cmd” and “Rsp” extensions are governed by the message category parameter of the enumeration item.

In Figure 2, the message name is created as follows:

“ETS” + “ “ + “CP DB Clear” + “Cmd”	to form	**ETS CP DB Clear Cmd**
“ETS” + “ “ + “CP DB Clear” + “Rsp”	to form	**ETS CP DB Clear Rsp**

3.2.1.2 Naming of Submessages in ETS Named Submessages

In Figure 3, the “ETS CP Trace” message represents a message generated from the ETS Base Message. The ETS CP Trace message is composed of an Id field plus one of the many messages represented by the union (TraceMsg). The union is based on the “CP TraceIds” enumeration. As Figure 3 shows, there is an implied relationship between the “ETS CP Trace” message, the “CP TraceIds” enumeration, and the name of message.

The layout of the message names is:

Form: (See notes 1 and 2) + “ “ + Enumeration Value + “ “ + (“Cmd” or “Rsp” or nothing)

Rules:

1. All message names start with the Base Message Name excluding the “ETS.”
2. If the Base Message Name ends in “Cmd” or “Rsp,” the extensions are excluded.
3. The “Enumeration Value” is the actual enumeration item name found in the enumeration.
4. The “Cmd” and “Rsp” extensions are governed by the message category parameter of the enumeration item.

In Figure 3, the message name is created as follows:

“CP Trace” + “ “ + “CP Mon Msg Trace” + “Rsp”	to form	**CP Trace CP Mon Msg Trace Rsp**

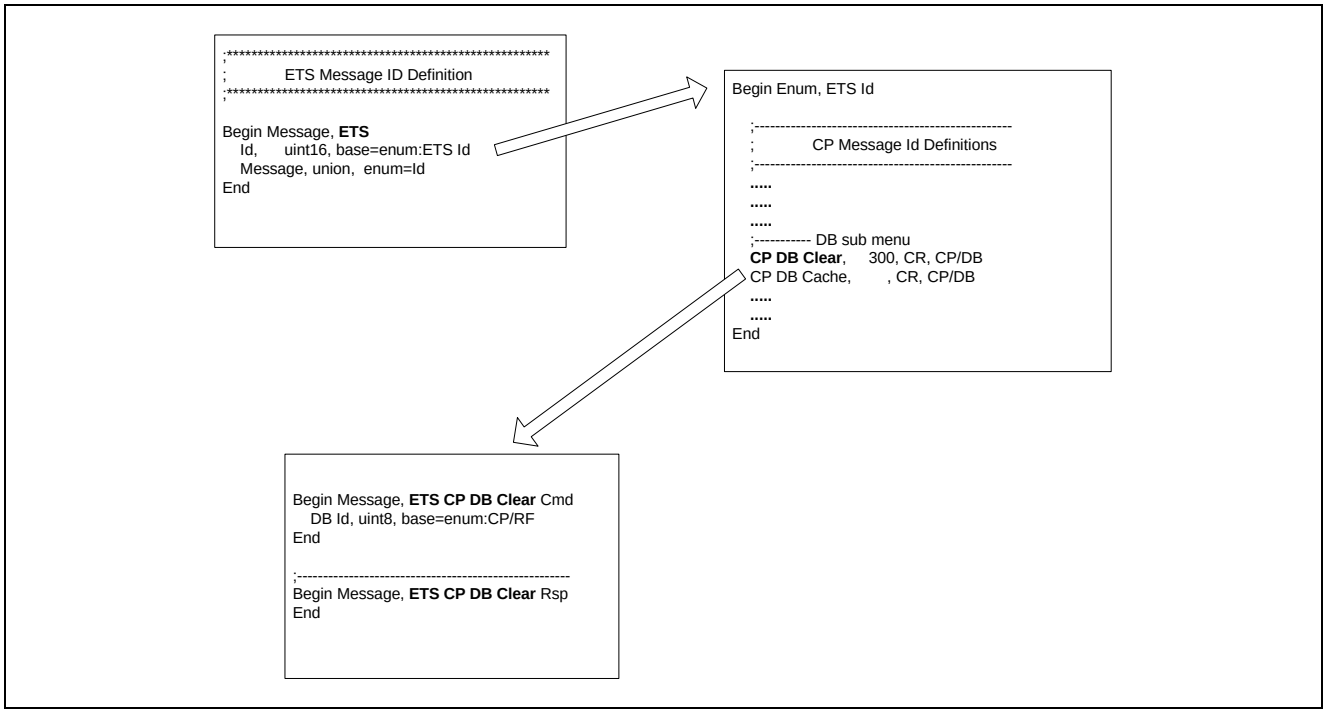


Figure 2. Naming Conventions for ETS Base Message

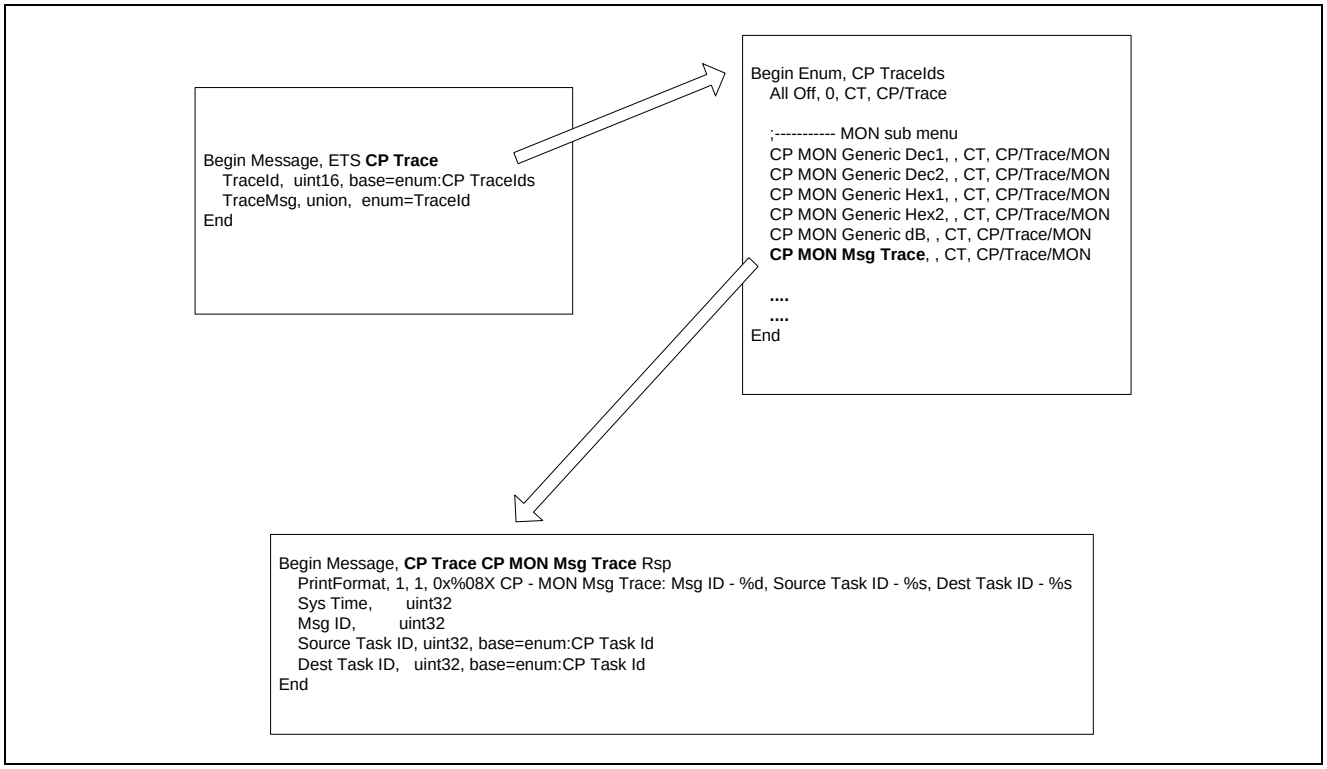


Figure 3. Naming Conventions for ETS Named Submessages

3.2.1.3 Naming of Submessages in Non-ETS Named Submessages

In Figure 4, the “CP Trace CP LMD Markov Trace Rsp” message represents a message generated from an ETS Named Submessage. The ““CP Trace CP LMD Markov Trace Rsp”” message is composed of multiple fields plus one of the many messages represented by the union (Data). The union is based on the “Markov Functions” enumeration. As Figure 4 shows, there is an implied relationship between the “CP Trace CP LMD Markov Trace Rsp” message, the “Markov Functions” enumeration, and the name of message. The layout of the message names is as follows:

Form: (See notes 1) + “ “ + Enumeration Value + “ “ + (“Cmd” or “Rsp” or nothing)

Rules:

1. If the Base Message Name ends in “Cmd” or “Rsp,” the extensions are excluded.
2. The “Enumeration Value” is the actual enumeration item name found in the enumeration.
3. The “Cmd” and “Rsp” extensions are governed by the message category parameter of the enumeration item.

In Figure 4, the message name is created as follows:

“CP Trace CP LMD Markov Trace” + “ “ + “Srv Opt Control” + “Rsp”

to form **CP Trace CP LMD Markov Trace Srv Opt Control Rsp**

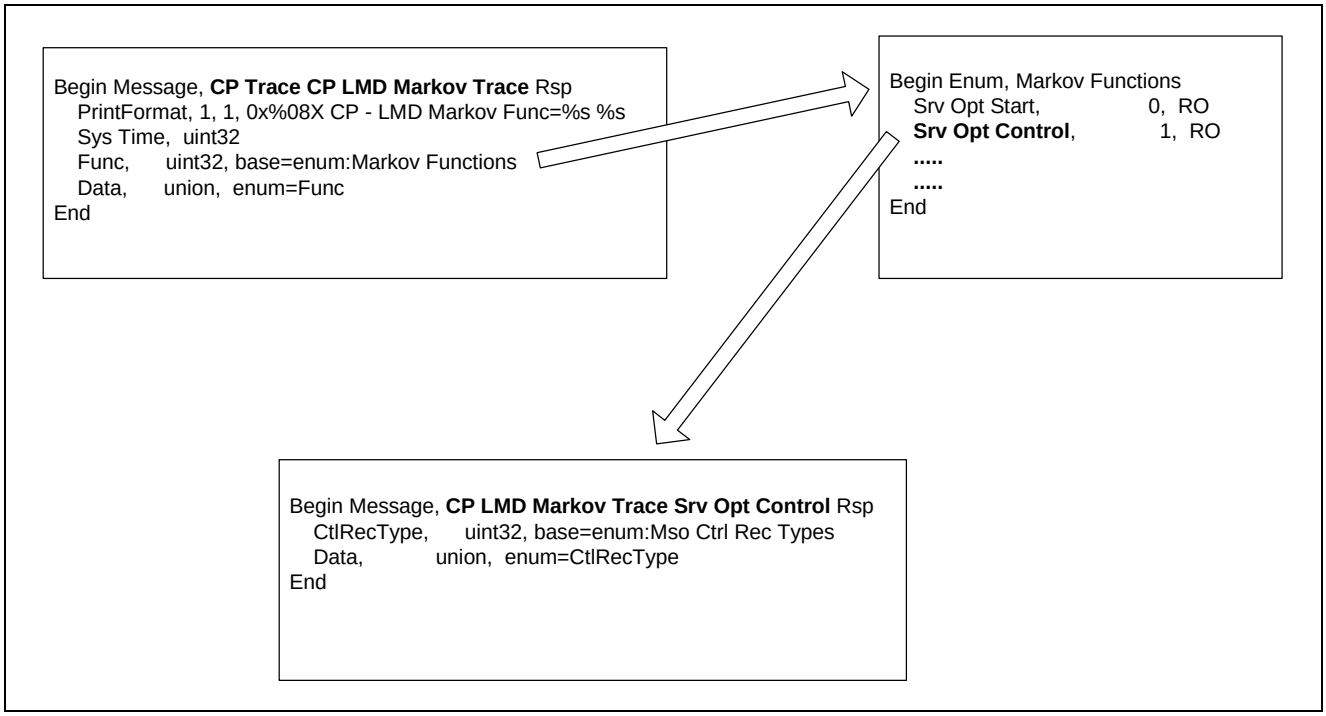


Figure 4. Naming Convention: Non-ETS Named Submessages

3.2.2 Message and Record Parameters

The following parameters can be placed on the *Begin* line of a message/record to provide extra control.

3.2.2.1 BitPack

The Bitpack parameter indicates that the message data can contain bit-fields and not necessarily be byte-aligned. Refer to the BitPack section for details.

3.2.2.2 PrintOnly

The PrintOnly flag is used for spy response messages and indicates that a spy dialog will not appear when the spy message is received. This causes the spy message to only be printed out.

3.2.2.3 TruncateBits=#

The TruncateBits option causes the specified number of bits to be truncated from the end of the message that is received. This is used where interpretation of the trailing data is not desired. The value of the parameter is the number of bits to truncate.

3.2.3 BitPack Message and Record Format

BitPack messages and records contain bit-aligned fields by default. Other messages and records contain byte-aligned fields by default.

Example:

```
Begin Message, MsgName, BitPack
      FieldName1, count=2
      FieldName2, count=2,
      base=enum:zero/one/two/three
```

```
      FieldName3, uint16
End

Begin Record RecName, BitPack
      Field1, count = 4
      Field2, uint32
End
```

BitPack Bit Fields are numeric. The format of the field definitions is:

<FieldName>, count=<bits> <, numeric field parameters>

In the example the first field (FieldName1) is displayed or entered as a 2-bit numeric value. The second field (FieldName2) is displayed or entered as an enumerated value. The third field (FieldName3) is an unsigned 16-bit integer.

The binary data for this example will be bit-aligned. There are a total of 20 bits packed as 2 two-bit fields followed by a single 16-bit field.

3.2.4 Message Examples

3.2.4.1 Example of Command/Response Message

The Loopback message has an Id equal to zero, being a command response type and having a menu item located under a Debug submenu. Also the descriptive text of the value is *Loopback Message* which would appear when the command menu item for the loopback message is selected. Based on this definition, the following two definitions for loopback command and response message can be made:

```
Begin Message, ETS CP Loopback Cmd
  Help ETS Loopback message command
  Data, array, type=uint8, maxsize=273, size=end, columns=60
End

Begin Message, ETS CP Loopback Rsp
  Help ETS Loopback message response
  Data, array,type=uint8, maxsize=273, size=end
End
```

3.2.4.2 Example of Response Only Message

The Fault message is a response only message and has an Id equal to 50. This allows for the associated response message to be defined as follows:

```
Begin Message, ETS CP Fault Rsp
  help Unit identifier
  Unit, uint8, base=enum:CP Unit
  Codes, union, enum=Unit
End
```

This case demonstrates that each message can itself contain a union and refer to other enumeration definitions. In this specific example there is a union member for each processor.

3.2.4.3 Example of Command/Response Sub-Message

The Spies message is a command response sub-message type and has an Id equal to 1-1. Because it is a CRM type it must refer to a message definition that contains a union. In this case the message definition is as follows:

```
Begin Message, ETS CP Spy
  SpyId,  uint16, base=enum:CP SpyIds
  SpyMsg, union,  enum=SpyId
End

Begin ENUM, ETS SpyIds
  All Off, 0, CT, CP/Spy, Lite
  CP Tx DSPM Mbox, , CT, CP/Spy/MON
  CP Tx DSPV Mbox, , CT, CP/Spy/MON
End

Begin Message, CP Spy Cmd
  Enable, uint8, base=bool
End

Begin Message, CP Spy All Off Rsp
End

Begin Record, DSP HW Mailbox
   Msg Id, uint16, base=enum:DSP HW Mailbox Ids
   Length, uint16
   Data,   array,  type=uint16, maxsize=40, size=Length
End

Begin Message, CP Spy CP Tx DSPM Mbox Rsp
   Sys Time, uint32
   Length,   uint16
   Num Msgs, uint16
   Data,     record, type=DSP HW Mailbox, maxsize=10, size=Num Msgs
End

Begin Message, CP Spy CP Tx DSPV Mbox Rsp
   Sys Time, uint32
   Length,   uint16
   Num Msgs, uint16
   Data,     record, type=DSP HW Mailbox, maxsize=10, size=Num Msgs
End
```

3.2.5 Record Definitions and Examples

Record definitions are sets of fields and are very similar to message definitions. The following is an example:

```
Begin Message, AMessage
    Data, record, type=ARecordName, maxsize=3
End

Begin Record, ARecordName
    Field1, uint8
    Field2, uint8
End
```

This defines *AMessage* which contains a fixed length 3-record array. The record *ARecordName* contains two unsigned 8-bit fields.

3.2.6 GlobalVar Definition

GlobalVar section defines the global variables in all ETS text definitions. Each entry in this section specifies the following:

- The name of the global variable
- The default value of this global variable

The following is an example:

```
Begin GlobalVar, Only One Global Var Block
  PREV, default = 6
  ; add more here
End
```

There are two ways to use a global variable, as described in the next two sections.

3.2.6.1 Associating a field with a global variable

To associate a field with a global variable, use GlobalVar attribute for a field. For example:

```
P REV, count = 8, GlobalVar=PREV, indent = 2
```

This means that if ETS is going to consume a message with this field, ETS will set the global variable PREV based on the value of this field. If ETS is going to produce a message with this field, ETS will use the value of global variable PREV to encode this field.

3.2.6.2 Using "exist" Condition Based on a Global Variable for a Field

For example:

```
CDMA Freq, count = 8, indent = 2, exist = (PREV > 5)
```

Here the PREV is a global variable specified in the GlobalVar section.

3.2.6.3 Using "exist" Condition Based on a Global Variable for a Group of Fields

For example:

```
Begin Message, CP Spy CP PE ENG_LAYER2_TR 13 15 Rsp, BitPack
  ;…
  Prat,  count = 2, indent = 2
   #existif (PREV >5)
        CDMA Freq, count = 11,  indent = 2
        Ext CDMA Freq, count = 11, indent = 2
   #endexistif
          ; …
End
```

The two fields, "CDMA Freq" and "Ext CDMA Freq", only exist when the global variable PREV is greater than 5.

3.2.7 Conditional Definitions

ETS Definitions can be defined conditional on global variables using #existif, #orexistif, and #endexistif. Field names must still be unique (no duplicated field allowed) even if they appear in different conditional statements.

The following is an example of the syntax for commands that are disabled based on a condition of a global variable:

```
CP AMPS Tune Radio1, 801, CO, RF/AMPS, disabled=GBL_RF_OPTION==1
CP AMPS Tune Radio2, 801, CO, RF/AMPS, disabled=GBL_RF_OPTION>1
CP AMPS Tune Radio3, 801, CO, RF/AMPS, disabled=GBL_RF_OPTION<3
```

Another example of using #existif is below. Depending on the GBL_RF_OPTION conditional statement the field "Use NVRAM data_FALSE" or "Use NVRAM data_TRUE" will display on the "CP AMPS Rx Cal Override" ETS command window dialog.

```
Begin Message, ETS CP AMPS Rx Cal Override Cmd
   #existif GBL_RF_OPTION==1
      Use NVRAM data_FALSE,     uint8, base=bool, default=False
   #orexistif GBL_RF_OPTION>1
      Use NVRAM data_TRUE,     uint8, base=bool, default=True
   #endexistif
      Pdm step/dB slope,  int16, base=16, default=0xcfa0
End
```

3.2.8 Invisible Definitions

ETS has an "invisible" definition feature that allows a command, database template or any field within a definition to be set "invisible" within EtsMain or ETS log data. This invisible type will cause the field not to be displayed at all depending on the conditional statement which will be evaluated (greater than, less than, equal, combination of greater (or less) than and equal, etc...), but it doesn't disturb the others fields information.

The syntax is similar to the “hidden” field option. The difference is that for "hidden" the field name is shown but the value is null bytes in the command or response; for "invisible" the field name won't show up at all in the dialog box or in the response message.

The following is an example of the syntax for invisible commands based on a global variable:

```
AMPSGoodFrameCt, uint16, base=10, Invisible=GBL_AMPS_SUPPORTED==0

AMPSBadFrameCt, uint16, base=10, hidden=GBL_AMPS_SUPPORTED==0
```

The next example shows use within the NAM1 DBTemplate:

```
SlotCycleIndex,    uint8, base=10, Invisible=1
```

A final example shows how individual fields can be made invisible within a command:

```
Begin Message, ETS CP Rev Channel Data Cmd

   ChnlDataRdy, uint16, base=16

   FundRate,    uint16, base=10, Invisible=1

   Words,       uint16, min=1, max=200

   Data,        array, type=uint16, maxsize=200, size=end, columns=60

End
```

3.2.9 Startup Commands Definition

The startup commands section causes the messages listed to be sent when ETS starts up. Also there is a menu selection “Debug/Send Startup Commands” that will send these messages when selected. The following example will send the CP version command and a loopback command when ETS starts up.

```
Begin Record, Startup Commands
    ETS, Id=CP Version
    ETS, ID=CP Loopback
End
```

3.2.10 Extra Log Window Definition

Extra log windows display selected messages. This definition file entry specifies the following:

- The title that will appear at the top of the window.
- The log entries that will be displayed.
- The menu selection of any spy messages that need to be turned off.

The format of the definition is as follows:

```
Begin LogExtra, <Window Title>
  <Leading text of Log entry to match>
  Menu, <Menu definition string for Spy or trace message>
End
```

An example that is used to display Spy Paging Messages is:

```
Begin LogExtra, Spy Paging Messages
  ETS,  Id=CP Spy, SpyId=CP PE ENG_LAYER2_TR 13 13
  Menu, CP/Spy/Protocol Stack/Engine/ENG_LAYER2_TR/CP PE ENG_LAYER2_TR 13 13
End
```

3.2.11 Fault Log Definition

The messages that will be added to the fault log are listed in this section. It has the following format.

```
Begin Record, Fault Log
    ETS, CP Fault
    ETS, DSPM Fault
    ETS, DSPV Fault
    ETS, CP Exe Fault
    ETS, CP File Info
End
```

3.2.12 Default ETS Options

This section defines the default values for the ETS Options dialog. This dialog allows customers to define their own set of defaults base on the characteristics of their system.

```
Begin Record, Default ETS Options
    InitBaud                ,115200
    SimCommMode             ,0
    TxPreambleEnable        ,1
    UnderFlowEnable         ,1
    DisableStartupCommands  ,0
    DnldWinSize             ,4
    DtrEnable               ,0
    InitFlashTimeout        ,25000
    LogWinSize              ,1000
    MaxMenuEntries          ,40
    XmtRate                 ,0
    PopupCommStatus         ,0
    PopupFaultLog           ,1
    TxPreambleEnable        ,12
    SpeechDnldAddr          ,0x2000000
    SpeechUpldAddr          ,0x2040000
    TraceLogNote            ,0
```

```
    TraceWinSize            ,1000
    UseCTypeForPrint        ,0
End
```

3.3 Field Definitions

The entries in the message definitions are field definitions. Each message definition must contain at least one field. The first parameter of a field entry is the field type. The remaining parameters are an optional list of comma-separated name value pairs with the format of (**,<name>=<value>**).

The allowable field types are shown below and are described in the following sections:

- **array** Array of Numbers
- **bit** Bit Field
- **numeric** Numeric Fields
- **record** Record Array
- **string** String Field
- **union** Union Field

3.3.1 Bit Fields

A bit field type is made up of one or more consecutive bits within a numeric field. Each bit field is itself a numeric field and most of the numeric field parameters apply. Bit fields fall into two categories: *BitPack Bit Fields* and *Non-BitPack Bit Fields*. The bit fields behave differently in each category.

3.3.1.1 BitPack Bit Fields

These bit fields are resident in a BitPack message or record. The fields in the message or record are not necessarily on byte boundaries. When a bit field is encountered, its bits are consumed or produced directly inline in the message. Depending upon the size of the bit field, the fields following may or may not be byte-aligned:

Example:

```
Begin Record RecName, BitPack
   Field1, count = 4
   Field2, bit, count = 3
   Field3, uint32
End
```

In this example, Field1 has a size of 4 bits, and Field2 has a size of 3 bits. Field3 starts immediately following Field2 and is therefore aligned at bit 8 of the record. The record contains 39 bits. Note that in Field1, the bit field type has not been specified because it is optional when the count field parameter is used.

3.3.1.2 Non-BitPack Bit Fields

These bit fields are found in non-BitPack messages and records (i.e., standard messages and records). Unlike BitPack bit fields, these bit fields operate on one to four data bytes. Multiple bit fields can be defined for a single set of data bytes. When the field is consumed or produced, all bits in the data bytes are used. Bits that are not referenced by a bit field are processed and then ignored.

The fields are defined by an offset, count, and position. The offset field parameter specifies the number of bits from the beginning of the data that the bit field starts in. The count field parameter specifies the number of bits in the bit field. The position parameter (pos) defines the relationship of each bit field with respect to the other bit fields within the same data bytes.

Note that the multiple byte fields will be interpreted in a big endian format. This is the same as the Motorola format. Also bit 0 is the most significant bit of the bit field.

Example:

```
Begin Record RecName
  Field1, bit, offset=15, count=1, bytes=2, pos=first, base=enum:Support Type
  Field2, bit, offset=0, count=1, bytes=2, pos=middle, base=enum:Support Type
  Field3, bit, offset=1, count=1, bytes=2, pos=last, base=enum:Support Type
  Field4, uint32
End
```

In this example, Fields 1 through 3 operate on the same 2-byte data block. The record contains 4 bytes.

3.3.1.3 Bit Field Parameter Differences

The two categories of bit fields have different requirements for the field parameters, as shown in Table 1.

Table 1. Differences in Bit Field Parameters

Parameter	BitPack Bit Fields	Non-BitPack Bit Fields
"bit" field type keyword	Optional when count parameter is present	Required
Bytes	Not Used	Required or assumed to be 1
Count	Required or assumed to be 1	Required or assumed to be 1
Offset	Not Used	Required
Pos	Optional	Required

3.3.2 Numeric Fields

The standard numeric types are:

- **int8** 8-bit signed integer
- **int16** 16-bit signed integer
- **int32** 32-bit signed integer
- **int64** 64-bit signed integer
- **uint8** 8-bit unsigned integer
- **uint16** 16-bit unsigned integer
- **uint32** 32-bit unsigned integer
- **uint64** 64-bit unsigned integer

By default unsigned numbers are printed and displayed in hexadecimal, and signed numbers are displayed in decimal (see *base* field parameter). The default minimum and maximum values are the range of the type.

Example:

```
Id, uint16
```

Bit sizes other than 8, 16, 32, and 64 can be defined using the *bytes* field parameter.

3.3.2.1 Password parameter

A password parameter can be added to a numeric field declaration. This will cause the characters of the field to be replaced with '*' characters when printed and when displayed in a command dialog window.

Example:

```
Name, uint16, password
```

3.3.3 Padding Fields

Some messages and records must end on a specific byte or bit boundary. If the message is of fixed size, a field can be added to insure the proper ending bit position. However, if the message has a variable length due to variable field arrays or fields that only exist under certain circumstances, then the padding must be dynamic.

The padding field type is used to dynamically insure that a message ends on a specific byte or bit boundary. The padding field uses the count parameter with negative value to define the required boundary. The following rules apply:

Rules:

1.	If count = -8	The message must end on a byte boundary.
2.	If size = -7 to –1	The message must end on a byte boundary plus the number of bits defined by the absolute value of count.
3.	In a message consume	At the end of message with a padding field, the message size is tested. If it matches the padding boundary, no action is taken. If not, the bits are skipped until the proper boundary is reached.
4.	In a message production	At the end of message with a padding field, the message size is tested. If it matches the padding boundary, no action is taken. If not, bits are added until the proper boundary is reached.

Example:

```
Begin Message, Test Cmd, BitPack
   Segment, uint8,  base=enum:Test Segments
   Offset,  uint16, base=10
   TestField, bit, count=3, exist= (Offset ==3)
   Reserved, count = -2
End
```

If TestField Does Not Exist:

The message size, before the padding field (Reserved), is 24 bits and on a byte boundary. The padding field indicates that the message must end 2 bits beyond a byte boundary, so 2 additional bits must be skipped on a consumption of the message, and added on a production of the message.

If TestField Does Exist:

The message size, before the padding field (Reserved), is 27 bits; 3 bits beyond a byte boundary. The padding field indicates that the message must end 2 bits beyond a byte boundary, so 7 additional bits must be skipped on a consumption of the message, and added on a production of the message.

3.3.4 Array Fields

The ***array*** field type represents a one-dimensional numeric array. The type field parameter is used to define the numeric type represented in the array. The allowable types are:

- int8
- int16
- int32
- uint8
- uint16
- uint32
- bit

Rules:

1. If the base field parameter is used, *enum* and *bool* are not allowed as values.
2. An **array field** is displayed as a list of space-delimited numbers.
3. A maxsize field parameter is required and defines the maximum number of elements in the array.
4. If just the *maxsize* parameter is specified (i.e., no size parameter) then the array is of fixed length.
5. A size field parameter is used to specify a variable length array field.
6. All other numeric field parameters apply to this field type with the exceptions of those noted in these rules.

Examples:

Fixed length array field with 253 entries

```
Data,array, type=uint8, maxsize=253
```

Fixed length array field whose size is the number of entries in the enumerated list "AnEnum."

```
Data, array, type=uint8, maxsize=SizeOfEnum:AnEnum
```

Fixed length array field displayed in 60 columns

```
Data, array, type=uint8, maxsize=253, columns=60
```

Fixed length array field displayed in 60 columns and 3 rows

```
Data, array, type=uint8, maxsize=253, columns=60, rows=3
```

Variable length array field to end of message with a maximum size of 253

```
Data, array, type=uint8, maxsize=253, size=end
```

Variable length array field with a length specified in the *Length* field.

```
Length, uint16
Data, array, type=uint8, maxsize=253, size=Length
```

Array of bitfields. This applies only to messages defined with the BitPack option.

```
Data, array, type=bit, count=2, maxsize=16
```

3.3.5 Record Fields

The record field type represents a one-dimensional arrays of records. The type field parameter is used to define the record to be used.

Rules:

1. The type parameter must be present and contain a valid record name.
2. A maxsize parameter is required and defines the maximum number of records in the array.
3. If just the *maxsize* parameter is specified (i.e., no size parameter) then the array has a fixed number of records.
4. A size parameter is used to specify a variable number of records in the array.
5. Record arrays can be used as fields of messages and records.

Examples:

Fixed length record array field with 10 "RecordA" records.

```
Data, record, type=RecordA, maxsize=10
```

Variable length record array field to end of message with a maximum of 20 "RecordB" records.

```
Data, record, type=RecordB, maxsize=20, size=end
```

Variable length record array field with the number of "*RecordC*" records defined by the *Length* field.

```
Length, uint16
Data, record, type=RecordC, maxsize=24, size=Length
```

3.3.6 String Fields

String fields can be of either fixed or variable length. By default, strings are variable length null terminated "C" style fields. If no field parameters are provided, the end of the field will be defined by the null terminator or the end of the message, whichever comes first. The types of allowable string formats are as follows:

3.3.6.1 Fixed Length String

A string field is defined as fixed length when the length field parameter is used. The following rules apply:

1. The field will always contain the number of bytes defined in the *length* parameter.
2. If a string is shorter than the length parameter, it will end in a null terminator. The rest of the bytes are undefined.
3. The size of the string can be equal to the value of the length parameter. When this occurs, the null terminator is not used.

Example:

```
Name,string, length=13
```

where the string will always be 13 characters (bytes) long.

3.3.6.2 Variable Length String

A variable length string field has no *length* parameter. The following rules apply:

1. The number of characters (bytes) in the field is normally defined by the size field parameter (see the size parameter for full details of usage).
2. If the *size* parameter is not defined, size=end is assumed.
3. If the *size* parameter is set to another field value or a field equation, the size of the field is fixed at that value. In this case, the field is treated the same as a fixed length string field.

Examples:

```
Name, string, size=end, columns=20
```

where the string ends with a null terminator or the end of the message, whichever comes first. It was displayed in 20 columns.

```
Name, string, size=end, columns=20, rows=3
```

where the string ends with a null terminator or the end of the message, whichever comes first. It was displayed in 20 columns and 3 rows.

```
Characters, uint8
Name, string, size=Characters
```

where the string has a fixed length defined by the *Characters* field.

3.3.6.3 Password parameter

A password parameter can be added to a string field declaration. This will cause the characters of the field to be replaced with '*' characters when printed and when displayed in a command dialog window.

Example:

```
Name, string, size=end, columns=20, password
```

3.3.7 Union Fields

The union field type is used to cluster submessages under a parent message. As Figure 5 below indicates, the union field represents multiple submessages that can be appended to the parent message. Each submessage takes the place of the union field to form the full message (see Figure 6 below). Messages with union fields have the following rules:

Rules:

1. The union field must contain an enum field parameter that is used to identify the ID field in the message that contains the submessage ID. In Figure 5, that field is the "Id" field.
2. The ID Field must contain a base=enum parameter. Short form enumerations are not allowed.
3. The enumeration has a special form for union fields that is explained in the Enumerations for Unions section.
4. A specific submessage naming convention is used and described in the Message Naming Conventions section.

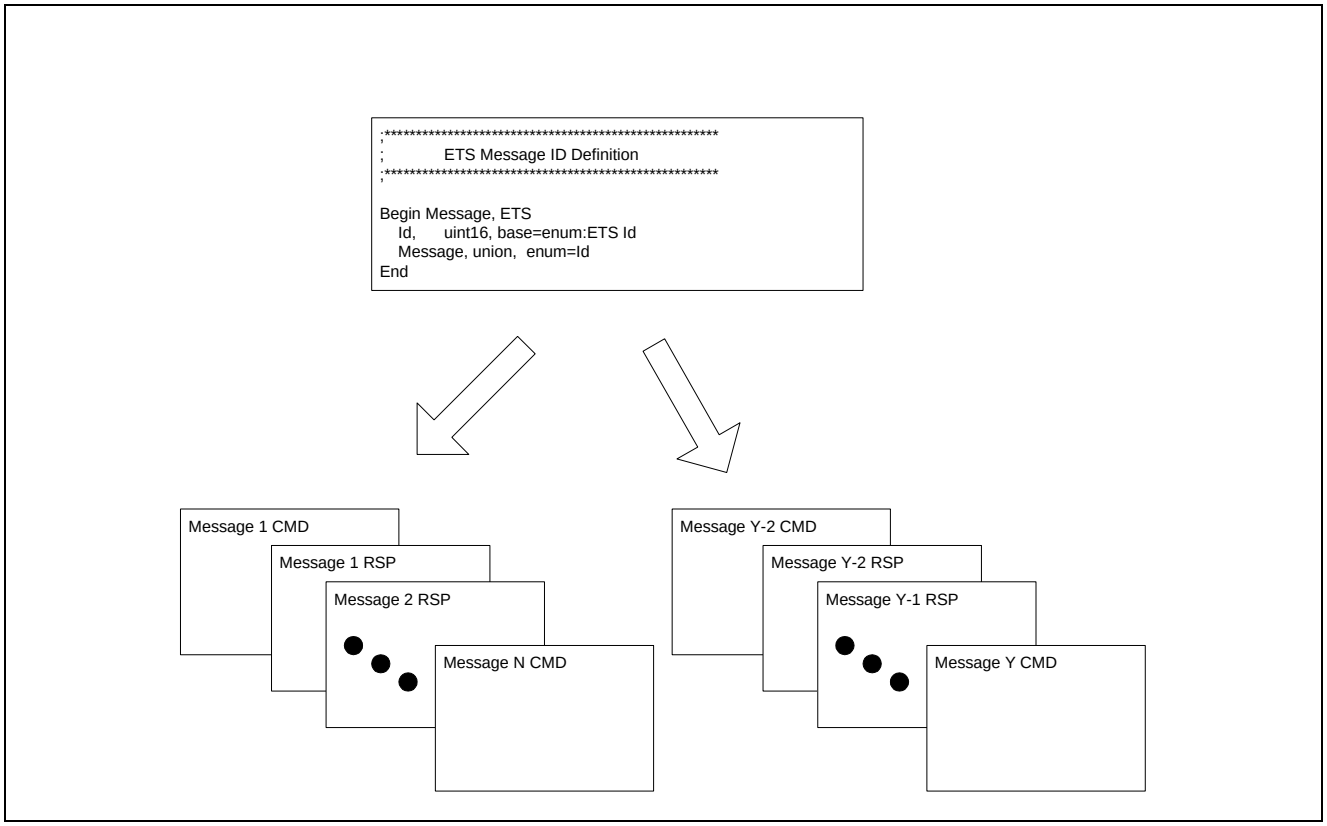


Figure 5. Message with Union Field

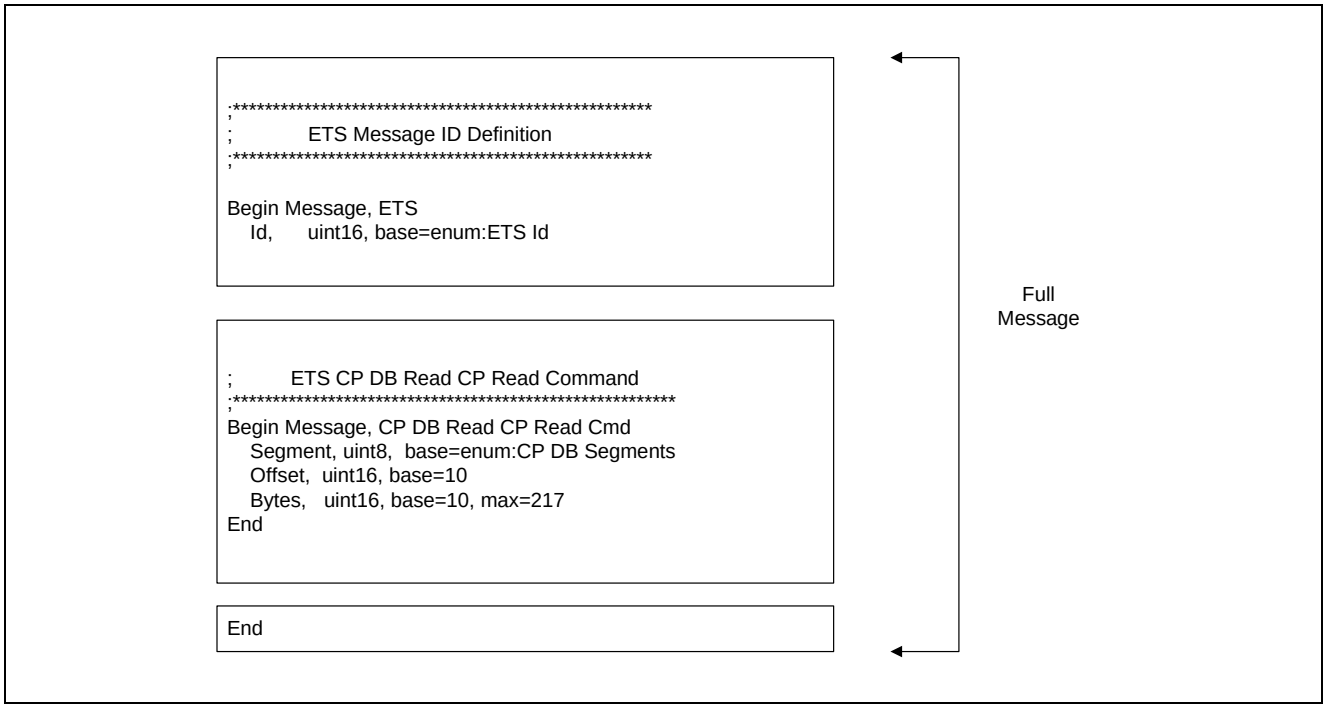


Figure 6. Message with Union Submessage Substitution

3.3.7.1 Union Field Identifier

The identifier field, for a union field, determines which union member is to be used. Normally the identifier field needs to be specified before the union field is defined. That is because union members are of variable length. As shown in the following definition, the union field is defined after its associated identifier field.

Example:

```
Begin Message, ETS
    Id, uint16, base=enum:ETS Id
    Message, union ,enum=Id
End
```

If the specific size of the union data is known it is possible to have a union identifier follow the union. This could either be because each of the union members are of the same length or because there is a field prior to the union field that determines the size of the union data. Only under these circumstances can the union identifier follow the union. For this to work the union field must contain either the *ConsumeBits* field parameter or the *ConsumeBytes* field parameter to define the length of the union submessage.

Example:

```
Begin Record, Overhead Message Record, Bitpack
   T1T2, count=2, indent=2
   DCC, count=2, indent=2
   Msg, union, enum=OHD, ConsumeBits=21
   OHD, count=3, base=enum:FOCC Overhead Message Enum
End
```

3.3.7.2 Union Field Examples

3.3.7.2.1 Example: Main Message

The following are excerpts from the main message definition identifier:

```
Begin ENUM, ETS Id
  help ETS Message Identification values.
  Loopback, 0, CR, Debug
  Help Loopback Messages
  MonFault, 1, RO
  Spies, 101, CRM, SpyTest
End
```

3.3.7.2.2 Example: Command Response Message

The Loopback message has an Id equal to zero, being a command response type and having a menu item located under a Debug submenu. Also the descriptive text of the value is *Loopback Messages* which would appear when the command menu item for the loopback message is selected.

Based on this definition the following two definitions for loopback command and response message can be made.

```
Begin Message, ETS Loopback Cmd
  Help ETS Loopback message command
  Data,array,type=uint8,maxsize=253,size=end,columns=60
End
Begin Message, ETS Loopback Rsp
  Help ETS Loopback message response
  Data,array,type=uint8,maxsize=253,size=end
End
```

3.3.7.2.3 Example: Response Only Message

The MonFault message is a response only message and has an Id equal to 1. This allows for the associated response message to be defined as follows:

```
Begin Message, ETS MonFault Rsp
  Proc, uint8, base=enum:ETS MonFault Processor
  help Processor identification
  Unit, union, enum=Proc
  help Unit identifier
End
```

This case demonstrates that each message can itself contain a union and refer to other enumeration definitions. In this specific example there is a union member for each processor.

3.3.7.2.4 Example: Command Response Sub-Message

The Spies message is a command response sub-message type and has an Id equal to 101. Because it is a Command Response Message (CRM) type it must refer to a message definition that contains a union. In this case the message definition is as follows:

```
Begin Message, ETS Spies
  SpyId, uint8, base=enum:ETS SpyIds
  SpyMsg, union ,enum=SpyId
End
Begin ENUM, ETS SpyIds
  Spy0, 0, CT, SpyTest
  Spy1, 1, CT, SpyTest
  Spy2, 2, CT, SpyTest
End
Begin Message, Spies Spy0 Rsp
  bArray, array, type=uint16, maxsize=126, size=end
End
```

This shows that there are three spy ids and also shows the message definition for Spy 0. Note that the name is *Spies Spy0 Rsp* **NOT** *ETS Spies Spy0 Rsp*.

3.3.8 Enumerations

Numeric fields can be constrained to a set of enumerated values. This option is useful for the following:

1. For entry of commands this limits the operator to a set of valid options and provides a textual list of those options in a drop down list box.
2. For display it provides a textual representation of the numeric value.
3. For union structures it provides a list of possible values to identify the union member in effect.
4. Received enumerated values are printed and displayed by their enumerated value when they match. If they do not match they are displayed as a number.
5. When dialogs are used to send commands containing an enumerated value, a drop down list is provided to select which of the values is desired. Values outside of the enumerated list are not provided as an option.

Enumerated lists are specified with either a short format or a long format.

3.3.8.1 Enumerations: Short Form

The short format allows for enumerated values that start at zero and increase by one for each enumeration. The short format cannot be used as a union member identifier. An example is:

Example:

```
Processor, uint8, base=enum:CP/DSPM/DSPV
```

In this example, the Processor field can have the following values:

CP = 0

DSPM=1

DSPV=2

3.3.8.2 Long Form

The long format allows the value of each enumeration to be specified individually. As in a C style enumeration if a enumerated items value is not specified its value is implicitly one more than the prior enumerated value. The enumeration is specified in an Enumeration Definition as follows.

Example:

```
Begin Message, CP DB Read CP Read Cmd
   Segment, uint8,  base=enum:CP DB Segments
   Offset,  uint16, base=10
   Bytes,   uint16, base=10, max=217
End

Begin Enum, CP DB Segments
   DB PSW NAM,           0x00
   DB PSW,               0x01
   DB PSW MS CAP DB,     0x02
   DB MMI MISC,          0x03
End
```

In this example, the Segment field is defined by the CP DB Segments enumeration.

3.3.8.3 Enumeration For Unions

A special type of enumeration definition is used to support a union field. The format of the enumeration is extended to the following set of comma-separated tokens.

- Name — The name of the enumerated value
- Value — The value assigned to the name.
- Message category — Message category (see below).
- Sub menu — Menu specification (see below).
- Lite — The word “lite” if this is part of the lite message set.
- Disabled — Identifies conditionally disabled command messages (see below).

The following is an example of a enum declaration for a union message.

```
A Uim Message, 5, CR, CP/UIM, lite, disabled=GBL_UIM==1
```

3.3.8.3.1 Message Category

The *Message Category* is one of the following values:

CR - Command Response
Represents a command and a response id. In its simplest form a CR message could be a command response pair that contains nothing but the message id. In which case there would be no associated union member definition.

CT - Command Trace
Represents a command and a trace id. A command message of a trace message is defined only once. Each trace response message needs to be defined.

CO - Command Only
Represents a command id only. No response will be returned for this message being sent.

RO - Response Only
Represents a response only message.

CRM - Command Response Message
Represents a command response message type. This type can only be used within a CR type. It means that the union member identifies a set of sub-messages. The current example of this is the spy messages.

3.3.8.3.2 Sub-Menu

The sub-menu parameter is optional and defines the menu structure before the name of the message appears in the menu. For example three messages could be used to define the mode of the CP. So the sub-menu for each of the messages might be defined as *Mode*. If a sub menu were not defined for the message then if it were to have a menu at all it would be a main menu item. Each command type category message will have an associated menu item.

Each level of the menu tree is specified by a slash. For example “Debug/Comm” would specify a Comm sub menu under the Debug menu.

3.3.8.3.3 Lite

If a message is to be a part of the lite message set, this field will be set to “lite”; otherwise, it is left blank.

3.3.8.3.4 Disabled

Some commands are applicable to a subset of the possible phone configurations. For example the set of messages dealing with GPS are only useful if the GPS feature is installed on the phone being tested. Global variables are used to identify the configuration.

This field has the format “disabled=<expr>”. <expr> is an expression that uses constants and global variables to determine If the message is to be disabled or not.

3.3.9 Field Parameters

Table 2 shows the relationship of fields types to field parameters. Each of the field parameters is described in the following sections.

Table 2. Field Types Versus Field Parameters

	Field						
Field Parameters	Array	Bit	Bit in BitPack	Numeric	Record	String	Union
Base	*						
Byteorder	*	*	*	*			
Bytes	*	*	*	*			
Columns	*	*	*	*	*	*	*
consumebits	*						*
consumebytes							*
Count		*	*				
Default	*	*	*	*		*	
Enum							*
Exist	*	*	*	*	*	*	*
Fields	*	*	*	*			
Format	*	*	*	*			
Indent	*	*	*	*	*	*	*
Length						*	
log10	*	*	*	*			
logscale	*	*	*	*			
max	*	*	*	*			
maxsize	*				*		
min	*	*	*	*			
minbytes					*		
noexist	*	*	*	*	*	*	*
offset	*	*		*			

	Field						
Field Parameters	Array	Bit	Bit in BitPack	Numeric	Record	String	Union
pos		*	*				
rows	*	*	*	*	*	*	
scale	*	*	*	*			
set	*	*	*	*			
size					*	*	
type	*				*		

3.3.9.1 Field Parameter: base

The base parameter allows entry and display of a numeric field to be controlled. The options for the base parameter are 10, 16, bool, and enum.

3.3.9.1.1 base: 10, 16

The numbers 10 and 16 specify decimal or hexadecimal as the display format. By default unsigned numbers are hexadecimal and signed numbers are decimal. These defaults can be overridden with the base parameter. When data is entered into a numeric field it can be entered in either hexadecimal or decimal regardless of the base parameter setting. An example is:

```
Id, uint16, base=10
```

3.3.9.1.2 base: bool

The bool option allows the field to be treated as a boolean. The representation in a control dialog will be a checkbox. The printout or display in a trace window will be either a zero or a one. Received values will be interpreted as zero for false and non-zero for true. The leading character of the default or script parameter could be T (true), Y (yes), F (false) or N (no). Also a numeric value is interpreted as zero for false and non-zero for true. A boolean true is the numeric value 1.

```
Id, uint16, base=bool
```

3.3.9.1.3 base: enum

In addition, an enumerated value can be specified. For more information, see the section, “Enumerated Values.”

3.3.9.1.4 base: Default Values

If the minimum or maximum parameters are not defined, the following default base rules apply:

1. > 8 Bytes — hexadecimal
2. If base defined: — it is used.
3. If base not defined:
   - int8 — decimal
   - int16 — decimal
   - int32 — decimal
   - int64 — decimal
   - uint8 — hexadecimal
   - uint16 — hexadecimal
   - uint32 — hexadecimal
   - uint64 — hexadecimal

When a bit field is defined without a base, it is treated like an integer so the intX rules apply.

3.3.9.2 Field Parameter: byteorder

The target system will have one of two types of byte ordering for 16-bit, 32-bit, and 64-bit integers. The options for this parameter are:

motorola Big Endian order, which means that 16-bit and 32-bit integers will have the Most Significant Byte placed first in the message stream. As an example:

Example: *0xF8AB4489* will result in a byte stream of: 0xF8 0xAB 0x44 0x89

intel Little Endian order, which means that 16-bit and 32-bit integers will have the Least Significant Byte placed first in the message stream. As an example:

Example: *0xF8AB4489* will result in a byte stream of: 0x89 0x44 0xAB 0xF8

If the *byteorder* parameter is not specified, "**intel**" is assumed.

3.3.9.3 Field Parameter: bytes

There are cases in which the number of bytes, in a numeric value, will not be 1, 2, 4, or 8. For those cases, the **bytes** parameter is used to define the number of bytes in the field. The following rules apply:

Rules:

1. For byte counts greater than 8, the *byteorder* field parameter is not used. The first byte of the number is presumed to be the high order byte and is always presented first.
2. The **bytes** parameter is used to define number of bytes referenced by a non-BitPack bit fields.
3. The **bytes** parameter has a maximum value of 4 for non-BitPack bit fields.

Examples:

A 3-byte (24-bit) value is defined for the A3ByteNumber field:

```
A3ByteNumber, uint8, bytes=3
```

A 7-byte (56-bit) value is defined for the A7ByteNumber field:

```
A7ByteNumber, uint8, bytes=7
```

The abc bit field is contained in 2 bytes:

```
abc, bit, offset=4, count=3,  bytes=2, pos=middle, base=enum:Support
Type
```

3.3.9.4 Field Parameter: columns

The columns parameter defines the number of characters to be displayed in a window. If not specified, the default is 1.

3.3.9.5 Field Parameter: ConsumeBits

The *ConsumeBits* field parameter is only used by the union field type or record field type. The *ConsumeBits* parameter defines the specific number of bits that a field will consume when a message is received. It can be a constant or an expression.

Example:

```
Begin Record, Overhead Message Record, Bitpack
    T1T2, count=2, indent=2
    DCC, count=2, indent=2
```

```
    LEN, count = 8, indent = 2
    Msg, union, enum=OHD, ConsumeBits= LEN + 1
    OHD, count=3, base=enum:FOCC Overhead Message Enum
End
```

In the example, the Msg union field has exactly 21 bits.

3.3.9.6 Field Parameter: ConsumeBytes

The *ConsumeBytes* field parameter is only used by the union field type or record field type. The *ConsumeBytes* parameter defines the specific number of bytes that a field will consume when a message is received. It can be a constant or an expression.

Example:

```
Begin Record, Overhead Message Record, Bitpack
   T1T2, count=2, indent=2
   DCC, count=2, indent=2
   Msg, union, enum=OHD, ConsumeBytes=21
   OHD, count=3, base=enum:FOCC Overhead Message Enum
End
```

In the example, the Msg union field has exactly 21 bytes.

3.3.9.7 Field Parameter: count

The count field parameter is used with bit fields to define the number of bits assigned to that field. Normally, the count parameter is set to a fixed numeric value. However, there are messages in which the count parameter must be set dynamically, based on some criteria. In these cases, the parameter can be set to a *field name* or a *field equation*. Each time that the message is consumed or produced, the value of the count parameter is calculated from the field value or the field equation.

Example:

```
Parity, count=4,  indent=2  count is fixed at 4 bits
Test1, bit, count=Parity    count is dynamically set to the value of the Parity field
Test2, count=(Test1 * 2)    count is dynamically set to the value of the equation
```

Example:

```
reserved, count=-8, indent=2   make the message byte aligned
reserved1, bit, count=-2       make the message byte aligned + 2 bits
```

3.3.9.8 Field Parameter: default

The default parameter is used to define a default value for a field. The values that can be used are field-type dependent and described in the following sections.

3.3.9.8.1 Default: String Fields

The default parameter allows an initial string value to be specified for command messages. The string provided cannot contain a comma because that would be treated as a parameter delimiter. If a default parameter is not specified, the string field is initialized to a Null String.

Example:

```
 Name, string, length=10, default=ten
```

3.3.9.8.2 Default: Numeric Fields

The value of the default parameter is dependent upon the base field parameter for the field.

10 or 16	values must be numeric or hexadecimal (i.e., 0x1234)
bool	values must be True, False, Yes, No
enum	values must be one of the allowable enumeration items

Example:

```
Band, uint8, base=enum:PCS/AMPS/Cellular/T53, default=PCS
NumWords, uint16, default=47
IsActive, uint8, base=bool, default=true
```

3.3.9.9 Field Parameter: enum

Any message can contain a single union field. Union Fields contain an *enum* field parameter that identifies the prior numeric field that is associated with an enumeration definition. Each entry in the enumerated list represents a union structure type identified by the name of the union field combined with the mnemonic for the enumerated value (see the Message Naming Conventions section).

Example:

```
Begin Message, ETS
   Help ETS Message frame
   Id, uint16ETSHELP.NF_001, baseETSMAIN.NB_001 =enum:ETS Id
   help Message Id
   Message, union ,enum=Id
   Help Message Data
End
```

This example defines the union field named *Message* whose members are identified by the *Id* field and listed in the *Ets Id* enumeration definition.

3.3.9.10 Field Parameter: exist

Normally, when a field is defined in a message, that field is always present. However, there are message fields that are only present under certain conditions such as another field having a specific value. The *exist* parameter is used to define criteria that must be true in order for the field to exist in the current message. When the *exist* parameter is used, it is set to a field equation. Each time that the message is consumed or produced, the equation is evaluated, and the result is tested for zero (False – does not exist) or nonzero (True – field exists).

Example:

`Parity, count=4, exist=(Field1 != 2)` The field exists only if *Field1* is not equal to 2.

3.3.9.10.1 ExistIf Statement

If the existence of a group of fields is dependant on the same parameter they can have the *#ExistIf* statement applied to them as a group.

Example:

```
Begin Message, AMessage
   F1, uint8
#existif F1==2
```

```
   F2, uint8
   F3, uint8
#endexistif

   F4, uint8
End
```

In this example fields F2 and F3 would have the existence parameter applied to them **exist=(F1==2).**

If the *#endexistif* statement is left out then *#existif* will be applied to all fields from the *#existif* until the *end* statement for the message.

3.3.9.11 Field Parameter: fields

The *fields* parameter function is used with large numeric fields. Large is defined as numeric field with a *bytes* parameter set to more than 8 bytes. The fields parameter is used to establish multiple subfields that the primary field should be divided into. This parameter is set to multiple space-separated numbers that identify the quantity of bytes to be applied to each subfield. The number of entries defines the number of subfields to create. The summation of the subfield byte counts must equal the value of the bytes parameter of the primary field.

The name of each subfield is created by appending a numeric offset, starting at zero, to the primary field name.

Example:

```
TestField, uint8, bytes=7, fields= 2 1 4
```

The example results in the following subfields:

TestField0	2 bytes
TestField1	1 bytes
TestField2	4 bytes

3.3.9.12 Field Parameter: Format

The displayed format of a numeric can be specified. This is intended to be used primarily with scaled values in order to get the floating point value correct.

Example:

```
Num, uint16, scale=Q3, format=%g”
```

3.3.9.13 Field Parameter: indent

When a message is displayed in a log window, the entire message will be displayed on a single line unless the *indent* parameter is specified. When the *indent* parameter is used, in a field definition, it specifies that the field will be printed on the next line down with the specified number of tabs before it. Each tab represents about one inch. An example of the *indent* field being used is:

Example:

```
Afield, uint8, indent=2
```

3.3.9.14 Field Parameter: length

The *length* parameter is used to specify the number of characters (bytes) used in a fixed-length string field.

Example:

```
Name,string, length=13
```

where the string will always be 13 characters (bytes) long.

3.3.9.15 Field Parameter: log10

A log10 display can occur for numeric fields. This displays received values by taking the Log10. Values entered are translated using the inverse of the Log10 or ten to the power of the entered value. Log10 numeric fields are limited to a value greater than zero.

The format is:

```
Num, uint16, Log10=true
```

Internally these are processed as scaled values. If scale and offset parameters are also specified they are applied prior to the Log10.

3.3.9.16 Field Parameter: logScale

This parameter allows the result of a log10 to be scaled. The format is.

```
Num, uint16, Log10=true, LogScale=10
```

3.3.9.17 Field Parameter: maxsize

The *maxsize* parameter is used to indicate the maximum number of entities represented by a field type. The entity types are field-type dependent.

Numeric Field	maxsize represent the maximum number of array elements
Record Array Fields	maxsize represents the maximum number of records

The *maxsize* parameter can take two forms:

1. A number from 1 to N
2. *SizeOfEnum:* <name of enumeration>.
   This form allows the maxsize to be set to the number of elements in the enumeration.

Examples:

Fixed length array field whose size is the number of entries in the enumerated list "AnEnum".

```
Data, array, type=uint8, maxsize=SizeOfEnum:AnEnum
```

Fixed length record array field with a maximum size of 25.

```
Data, record, type=ARecordDef, maxsize=25
```

3.3.9.18 Field Parameter: min and max

The minimum and maximum field parameters are used to define the limits of the numeric field values that can be entered by a user. They have no effect on the values that can be received.

Examples:

```
Id, uint16 min=2, max=10
Id, int16 min=-5, max=5
```

If a minimum or maximum parameter is not defined, the defaults shown in Table 3 apply.

Table 3. Default Values for Min and Max Field Parameters

Numeric Type	Minimum	Maximum
int8	-128	127
int16	-32768	32767
int32	-2,147,483,648	2,147,483,647
int64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
uint8	0	255
uint16	0	65535
uint32	0	4294967295
uint64	0	18,446,744,073,709,551,615

3.3.9.19 Field Parameter: minbytes

The *minbytes* parameter is used in a record array field. If the size of the record array is defined as size=end (i.e., to end of message), the *minbytes* parameter can be used to define the minimum number of remaining bytes that must be in the message data in order to attempt to interpret it as a record in the array. If the *minbytes* parameter is not specified, the default is one byte.

As an example, if the minimum record size were 4 bytes, and there was the potential for unused bytes to be at the end of the message, the *minbytes* parameter should be set to four.

Example:

Variable length record array field to end of message with a maximum size of 20. The *minbytes* parameter used to set minimum record size is 4 bytes.

```
Data, record, type=ARecordDef, maxsize=20, size=end, minbytes=4
```

3.3.9.20 Field Parameter: noexist

In several messages, when there is no data received for a parameter, there is a specific meaning. The *noexist* parameter allows a value to be printed out for a field that was not received. Currently this parameter is only implemented for numeric fields. When the numeric field is an identifier for a union, the union field interprets the non-existent field as an indicator that the union field does not exist as well.

Example:

```
>>> Msg Id,   uint8, base=enum:Paging Msg Enum, NoExist=Null, indent=1
```

3.3.9.21 Field Parameter: offset

The *offset* parameter function is dependent upon the field type.

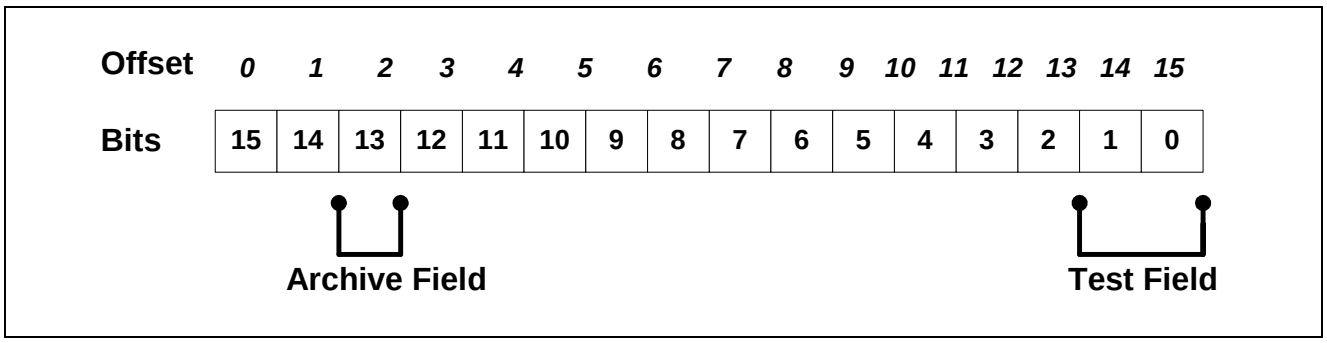
3.3.9.21.1 offset: Bit Fields

This parameter is used with non-bitpack bit fields. The offset parameter represents the bit start location, of the field, in the integer containing the field. The *offset* is referenced from the most significant bit (MSB) of the integer. An *offset* of zero refers to the MSB of the integer. The following example shows how this parameter works in this context:

Example:

```
Test, bit, count=2, offset=14, byte=2, pos=first, indent=1
Archive, bit, count=1, offset=2, byte=2, pos=last, base=enum:clear/set,
indent=1
```

The example has two fields that are contained in a two-byte (int16) integer. The bit layout is:



3.3.9.21.2 offset: Numeric Fields

In this context, the *offset* parameter is used as a numeric offset in the translation of an integer to a floating point number. See the scale for a full description of how the *offset* parameter is used.

3.3.9.22 Field Parameter: pos

The position parameter is used to define the boundaries of a cluster of bit fields or non-bitpacked fields. The *pos* parameter is ignored on bitpacked fields. The available values are:

first	first bit field in the cluster
last	last bit field in the cluster
middle	field is between first and last
only	only one bit field is in the cluster

Example:

```
Field1, bit, offset=15,  count=1,  bytes=2, pos=middle, base=enum:Support Type
Field2, bit, offset=0,   count=1,  bytes=2, pos=middle, base=enum:Support Type
Field3, bit, offset=1,   count=1,  bytes=2, pos=middle, base=enum:Support Type
Field4, bit, offset=2,   count=1,  bytes=2, pos=middle, base=enum:Support Type
Field5, bit, offset=3,   count=1,  bytes=2, pos=middle, base=enum:Support Type
Field6, bit, offset=4,   count=1,  bytes=2, pos=middle, base=enum:Support Type
Field7, bit, offset=5,   count=1,  bytes=2, pos=middle, base=enum:Support Type
Field8, bit, offset=6,   count=1,  bytes=2, pos=middle, base=enum:Support Type
Field9, bit, offset=7,   count=1,  bytes=2, pos=last,   base=enum:Support Type
Field10, bit, offset=5,  count=3,  bytes=1, pos=only
```

Fields 1 through 9 are part of the same bit field cluster, as defined by the *first*, *middle*, and *last pos* parameters. These fields apply to the same 2-byte integer. *Field10* is a single field cluster as indicated by the *pos* parameter value of *only*.

If the *exist* field parameter is set for a bit field, the *pos* must be set to *only*.

3.3.9.23 Field Parameter: rows

The *rows* parameter defines the number of rows to display in a window. If not specified, the default is 1.

3.3.9.24 Field Parameter: scale

The *scale* parameter is used with the *offset* parameters to allow a numeric value to be displayed and entered as a floating point value. The information that is moved across the serial interface is still an integer of some type. The formula that is used to convert from the integer to a float is:

Float = (scale * int) + offset

To convert from the floating point value to an integer:

Int = (float- offset) / scale

The *scale* parameter can be specified as either a floating point multiplier or as a Q#. The Q format indicates the number of bits that a binary decimal point should be moved to the left. If the scale parameter is Q3 then the scale value would be 2.0 to the minus 3 power which is 0.125. Negative values are allowed in the Q notation.

The offset parameter cannot be used without the scale parameter. If only an offset is desired the scale parameter must be set to 1.0.

The minimum and maximum values should be specified as floating point values if they are used at all.

Examples of *scale* and *offset* parameters are:

```
Num, uint16, scale=0.125
Num, uint16, scale=1.0, offset=3.224
Num, uint16, scale=Q3
```

3.3.9.25 Field Parameter: set

Normally, the value of fields is set by the user. However, some field values are based on other information, found in the message, such as the number of bytes occupied by another field. The *set* parameter is used to define a field equation that is used as the value for the field.

Example:

```
Begin Message, MsgName
  Field1, uint8, set=(NumOfBytes(Field3))
  Id, uint16, base=enum:ETS Id
  Field3,union, enum=Id
 End
```

In the example, the value of Field1 is dynamically set to the number of bytes contained in *Field3* when the message is produced.

3.3.9.26 Field Parameter: size

A size parameter is used to specify a variable length field. The following sections describe how the size parameter is interpreted based on its value.

3.3.9.26.1 size: end

With this setting, the field extends to the end of the message. The exceptions are:

1. For string fields, the field extends to the null-terminator or the end of the message, depending upon which comes first.
2. When a maxsize field parameter is used, the field extends to the end of the message or maxsize, depending upon which comes first.

Example:

```
Phrase, record, type=Vrec Vocabulary, size=end, maxsize=16
Args, array, type=uint32, maxsize=10, size=end
Name, string, size=end
```

3.3.9.26.2 size: Field Name

When the size parameter is set to a field name, then the value of that field is used as the value of size. The field must be defined prior to the current field or a syntax error will occur.

Example:

```
Characters, uint8
Name, string, size=Characters
```

3.3.9.26.3 size: Field Equation

When the size parameter is set to a field equation, then the result of the equation is used as the value of size.

Example:

```
Characters, uint8
Name, string, size=(Characters * 4)
```

3.3.9.27 Field Parameter: type

The *type* parameter function is dependent upon the field type.

numeric arrays The *type* parameter is set to the numeric type to be applied to the array entities. The allowable values are the same as those found in numeric fields.

Example: `Data, array, type=uint8, maxsize=355, size=Bytes`

record arrays When the *type* parameter is used in a record field, it is set to the name of the record to be applied to the field.

Example: `Hyst Low Gain State Coords, record, type=RxPwr Pts Low, maxsize=8`

Multiple field parameters can be set dynamically using expressions. An example of a field equation is:

count = **((Field1 + 2) * NumOfBytes (Field2))**

3.3.9.28 Field Parameter: Hidden

The hidden field parameter is used to hide the values of a field from the user. The format of this parameter is “hidden=<expr>”. When the expression is nonzero the field will be hidden.

3.3.9.29 Field Parameter: ReadOnly

The readonly field parameter is used prevent the user from modifying a field. The format of this parameter is “readonly=<expr>”. When the expression is nonzero the field can only be viewed by the user and cannot be modified.

Note: This option is only relevant to command messages. Fields within response messages are implicitly readonly.

3.3.10 Expression

Expression has the following rules:

1. **Expression operators**

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulation
!	Not
&	Bit And
|	Bit Or
^	Bit Xor
&&	Logical And
||	Logical Or
==	Equal To
!=	Not Equal To
>	Greater Than
<	Less Than
>=	Greater Than or Equal To
<=	Less Than or Equal To
?	Question Mark
:	Selection

2. **Special Functions**

   Count_of({path[]} expression) Refer to paragraph 3.3.10.2 COUNT_OF.

3. A selector expression is supported. It takes the form of:

   expression ? TrueResult : FalseResult

4. If an expression does not match any of the special functions and is not numeric, it is assumed to be a field name.

5. Operator precedence is observed based on conventional C language.

3.3.10.1 Field Names as Entities

Field names are used as entities in expressions. When a message consumption or production is in progress, the field, associated with the name, is queried for its value or size. In most cases, the queried field is present in the same message as the field that contains the field (see the *Length* and *Data* fields in the example). There are some message definitions that have dependencies to fields that are not in same message level or record. In the example, the existence of the *TimeValue* field, of the *Test Record*, is dependent upon the *DataSize* field in the message level that contains the record. To create a path to the *DataSize* field, DOS style notation of "..\" is used. The notation can be used to continue up as many levels as necessary (i.e., ..\..\.. and so on).

Example:

```
Begin Message, ETS CP FSI Block Read Rsp
      DataSize, uint16, base=10
      Length,   uint16, base=10
      Pad, record, type=Test Record, maxsize=1
      Data, array, type=uint8, maxsize=512, size=Length, base=16, indent=1
End

Begin Record Test Record
      TimeValue,  uint16, exist= (..\DataSize > 3)
End
```

3.3.10.2 COUNT_OF

Count_of({path[]} expression)

This was defined to handle an odd case in the IS2000 Rev. C specification. It requires that the number of occurrences of a field being set to 1 with in a set of records need to be counted in order to determine the size of another record array.

This function returns a count of the number of times that the expression evaluates to non-zero for each case of the path being applied to the fields within the expression. The path must be a sequence of field specifiers that identify a record array. The expression will contain fields. In this context the field name is first checked to see if it exists in the current message, then if it is a global variable and last if it is a member of the record identified by the path.

The following is an example.

```
Afield, array,type=int8,maxsize=20,size=count_of({..\FirstRec[]\LowerRec[]} AfieldInLowerRec==1)
```

4 Utilities

4.1 Include File Generator

A set of include files can be generated from the message definition files. The names of the files and the contents of the files are specified in a definition file that is selected using the Log/Make Include… menu item.

Two types of “C” definitions can be created using this utility. The first is an enumerated list, and the other is an initialized list of strings.

4.1.1 Enumerated List Generation

Each definition causes an include file to be generated. In the following example the file is *EtsId.h*. A single command file could contain multiple definitions, thereby generating multiple files.

Files which contain default header and footer information for the target include file can be specified. The header file is copied to the target file at the start of the process, and at the end of the process the footer file is appended to the target file.

The *ENUM* statements specify an enumeration definition that will be converted into a *typedef enum..* In the example the "ETS Id" enumeration from the etsmsg.txt file would be selected. The *NAME* parameter is optional and specifies the name of the typedef. The *PRE* parameter is optional and specifies a string that is to be attached to the beginning of the name of each enumerated value. The *POST* parameter is optional and specifies a string that is appended to the name of each enumerated value.

The following is an example include definition and its resulting typedef:

Include definition

```
Begin Include, EtsId.h
    Header, EtsIdPre.h
    Footer, EtsIdPost.h
    enum, ETS Id, name=EtsIdT, pre=MON_, post=_SYS
End
```

Enum definition from the message definition file

```
Begin ENUM, ETS Id
    Loopback, 0, CR, Debug
    Fault, 1, RO, Debug
End
```

Generated enumerated list

```
typedef enum
{
    MON_LOOPBACK_SYS=0,
    MON_FAULT_SYS=1
} EtsIdT;
```

4.1.2 String List Generation

The string list generator is designed to support Trace message testing. It extracts the help information that is associated with a list of enumerated values specified in the message definition file. As in the include file definition the name of the file can be specified. The only parameter used for the *ENUM* statements is the *NAME*.

The following example shows the string definition and the resulting string array that is generated.

Trace String definition:

```
Begin Message, ETS CP Trace
      TraceId,  uint16, base=enum:CP TraceIds
      TraceMsg, union,  enum=TraceId
End
```

Trace ID definitions:

```
Begin Enum, CP TraceIds
    All Off, 0, CT, CP/Trace, Lite
    CP MON Generic Dec1, , CT, CP/Trace/MON
    CP MON Generic Dec2, , CT, CP/Trace/MON
End
```

Trace Response definitions:

```
Begin Message, CP Trace CP MON Generic Dec1 Rsp
    PrintFormat, 1, 1, 0x%08X CP - MON Generic Dec1: %d, %d, %d, %d, %d, %d, %d,
%d, %d, %d
    Sys Time,    uint32
    Args,        array, type=int32, maxsize=10, size=end
End

Begin Message, CP Trace CP MON Generic Dec2 Rsp
    PrintFormat, 1, 1, 0x%08X CP - MON Generic Dec2: %d, %d, %d, %d, %d, %d, %d,
%d, %d, %d
    Sys Time,    uint32
    Args,        array, type=int32, maxsize=10, size=end
End
```

4.2 User Interface

The ETS provides complete visibility, logging, and control of all messages exchanged with the CP.

4.2.1 Ets Options Dialog

The Ets options dialog, accessed from the File menu, is shown in Figure 7.

Table 4 describes each option.

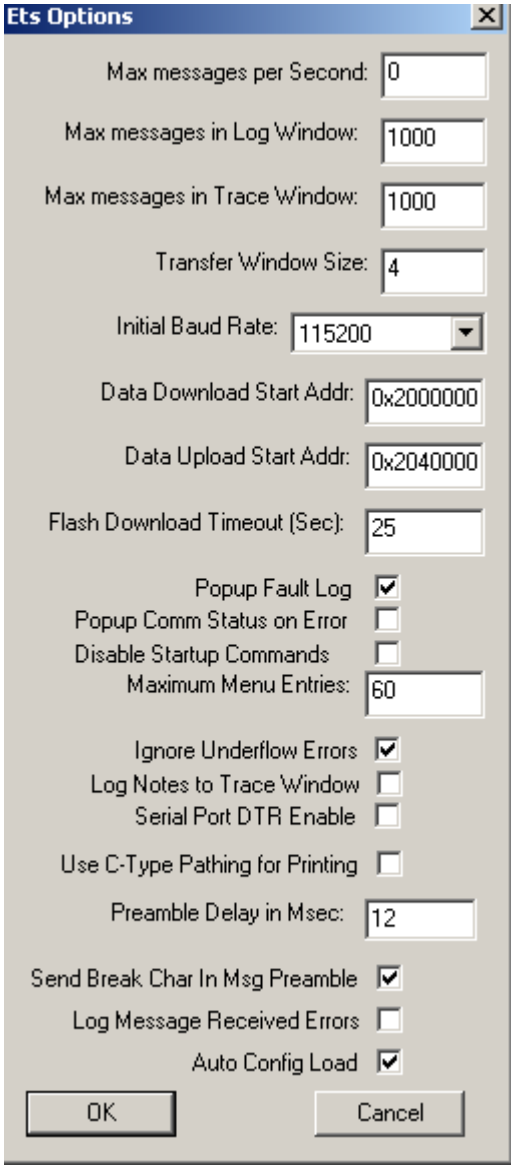


Figure 7. Options Dialog

Table 4. Ets Options Dialog Selections

Option	Description
Max messages per Second	Limits the maximum number of messages that ETS will send to the CP in a second. When this is set to 0, there is no limit. This option is initialized to 0.
Max messages in Log Window	This sets the maximum number of messages that will be displayed in the main log window. This option is initialized to 1000 messages.
Max messages in Trace Window	Sets the maximum number of messages to be displayed in the Trace window. This option is initialized to 1000 messages.
Transfer Window Size	When performing uploads, downloads, and loopback tests, multiple data transfer messages can be sent before an acknowledgement is received. The number of messages is called the "Transfer Window Size." This option is initialized to 4 and should not need to be changed.
Initial Baud Rate	This is the baud rate that ETS will use when started. The baud rate can be changed later by using the Serial Port Configuration Dialog from the Debug/Comm/Serial… menu item. This option is initialized to 115,200 Baud.
Data Download Start Addr	This is the default start address for the speech download buffer. This option is initialized to 0x2000000.
Data Upload Start Addr	This is the default start address for the speech upload buffer. This option is initialized to 0x2040000.
Flash Download Timeout (Sec)	The number of seconds that are allowed for the CP to acknowledge a flash download command. This option is initialized to 25000 ms (25 seconds).
Popup Fault Log	When checked, the fault log will be made the front window when an entry is made. This option is initially checked.
Popup Comm Status on Error	When checked, the Communications Status dialog pops up when an error occurs on the serial interface. This could be either a character error, such as parity or framing, or a message level error such as checksum for message framing. This option is initially unchecked.
Maximum Menu Entries	Sets the maximum number of menu entries that are allowed before the menu is replaced with a dialog box. This option is initialized to 40.
Ignore Underflow Errors	When checked, all underflow errors will be ignored. Underflow errors are generated when ETS receives less data than is defined for an entry in the definition text file. This option is initially checked.
Log Notes to Trace Window	When checked, notes that are entered from the COM interface are entered in both the main and the trace windows. This option is initially unchecked.
Use C-Type Pathing for Printing	When checked, all message printouts will use the C Language type pathing of xx.yy[3].rr. When not checked, no pathing will be applied to message fields except when records are present. When records are present, the pathing will be of the type fieldname.X where X is the record number. This option is initially checked.
Serial Port DTR Enable	When checked, enables the assertion of the DTR signal on the serial interface port. This option is initially unchecked.
Send Break Char In Msg Preamble	When checked, send a message preamble which contains a break character (0x00) followed by a multiple millisecond (see Preamble Delay option) pause followed by the start of message SYNC character. This preamble is sent before each message transmitted by ETS with the exception that if messages are clustered together, the preamble is only sent before the first message in the cluster. This preamble is used to wake up the CP from deep sleep. This option is initially checked.
Preamble Delay in Msec	This value is the delay used when the message preamble option is enabled. The option is initialized to 12 milliseconds.
Log Message Received Errors	Enables logging of errors in the received message framing.
Auto Config Load	When checked, the currently loaded Configuration file will be reloaded when a phone (MSUT) sends a “Configuration Info” message.

4.2.2 Logging

As messages are exchanged they are textually recorded in a Log Window and optionally appended to a Log file. In addition any ETS errors that occur are logged. Log entries are time stamped to the nearest tenth of a second. Entries fall into one of three classes:

1. Messages sent to the CP. Identified by a “<”.
2. Messages sent by the CP. Identified by a “>”.
3. ETS error messages. Identified by a “:”

ETS errors include serial interface errors as well as messages being improperly formatted.

Logging is controlled through the Log pull down menu, which contains the following selections.

- Clear - Clears the log window.
- Pause - Temporarily suspends logging. **Note:** this will prevent the OLE automation interface from working.
- Open - Opens a file to append log entries to.
- Close - Closes a log file that is open.

4.2.2.1 Extra Log Windows

Extra Log windows are used to separate specific messages from the main log window. Typically this will be used for Spy messages that are too complicated for display in a dialog window. Entries that are displayed in an extra log window can optionally be sent to the main log window.

4.2.3 Menus and Dialogs

For the Command Response, Command Trace, and Command Only message categories there will be menu items provided. For the Command Trace messages the menu item will be either checked or unchecked depending on the state of the trace. For the Command Response and Command Only messages a dialog window will be provided to allow the message to be sent. The command dialogs will have a checkbox that will allow them to either close after a command has been sent or remain visible.

4.3 Script Utility

The script utility provides the simplest method for automated processing. It is easy to use and requires minimal time to learn. It also allows command sequences to be recorded and played back. In addition, the following language elements are provided:

- END - Ends the script or returns from an include file. The “End of File” is interpreted the same as an END statement.
- PROMPT - Prompts the user. Curly braces can be used to continue the prompt text to the following line. If no text is provided then a default prompt is used asking the user to Press the OK button.
- WAIT - Waits the specified number of milliseconds.
- LOOP - Loops the specified number of times. If Zero is entered the loop is infinite. Only one level of loop is allowed. Loops cannot be nested.
- ENDLOOP - Marks the last statement in the loop.
- INCLUDE - Other script files can be included using this command.
- CONTINUATION LINES - A trailing backslash character "\" causes a line to be continued.
- LOGNOTE - Makes a note in the main log window.
- LOGTOFILE - Opens a log file that future main log entries will be copied to.
- LOGCLOSE - Closes a log file that is open.

- LOGTRACEFILE - Opens a trace window log file that future trace log entries will be copied to.
- LOGTRACECLOSE - Closes a trace window log file that is open.

Scripts can be saved to a file and reopened. Simple editing capabilities are provided.

4.4 OLE Automation Interface

All of the sophisticated display and control windows will use this interface to exchange messages with the CP.

The OLE automation interface provides the following.

1. Access to all response messages.
2. Access to all command messages.
3. Access to all error messages.
4. Generation of command messages.
5. Control and observation of system modes and functions *(to be determined)*.

This interface can be accessed from any Windows 98, Windows2000, or Windows XP programming environment that can use OLE automation. Some examples are Visual C++®, Visual Basic, Labview™, and Excel.

The interface members are:

- void SendCommand(LPCTSTR pCmd);
- BOOL m_getCommands;
- BSTR GetCommand();
- BOOL m_getResponses;
- BSTR GetResponse();
- BOOL m_getErrors;
- BSTR GetError();

The SendCommand function sends a command to EtsMain. The textual parameter is a command string that is exactly as it would appear in the Log Window.

The member variables m_getCommands, m_getResponses and m_getErrors determine if commands, responses, or errors are to be made available. As commands, responses, and error entries are made in the Log Window and if their corresponding flag is true, the string will be placed in a 100-entry queue for each of the commands, responses, or errors. They can then be retrieved using the GetCommand, GetResponse, or GetError functions. If any of these functions are called and data is not available, a null string will be returned.

4.5 Knobs, Dials, Graphs

For the screens that require more sophisticated user interfaces, specialized applications can be generated using the OLE Automation Interface to access the message interface.

4.6 ViaCrypt Utility

The ViaCrypt utility provides a way to encrypt etsmsg.txt. It is used at the command line in the following way:

```
ViaCrypt.exe etsmsg.txt etsmsg.bin
```

The new etsmsg.bin file is generated in the same directory as etsmsg.txt file. It is the user's option to encrypt the full, lite or verylite message definition set EtsMsg.txt file. See 5.1.1.6 Save Definition to generate different versions of etsmsg.txt.

Note: When using full message definitions the etsmsg.txt file cannot have any “include” statements. When using the full message definition set first consolidate the ETS Text files using the following command:

```
EtsMain.exe -s -savedefs=c:\temp\etsmsg.txt
```

The etsmsg.bin file should be put in the start directory when EtsMain.exe is run.

5 Modules

The ETS is composed of four types of executables: the ETS Main, Script Utility, Control Panels, and Test Drivers.

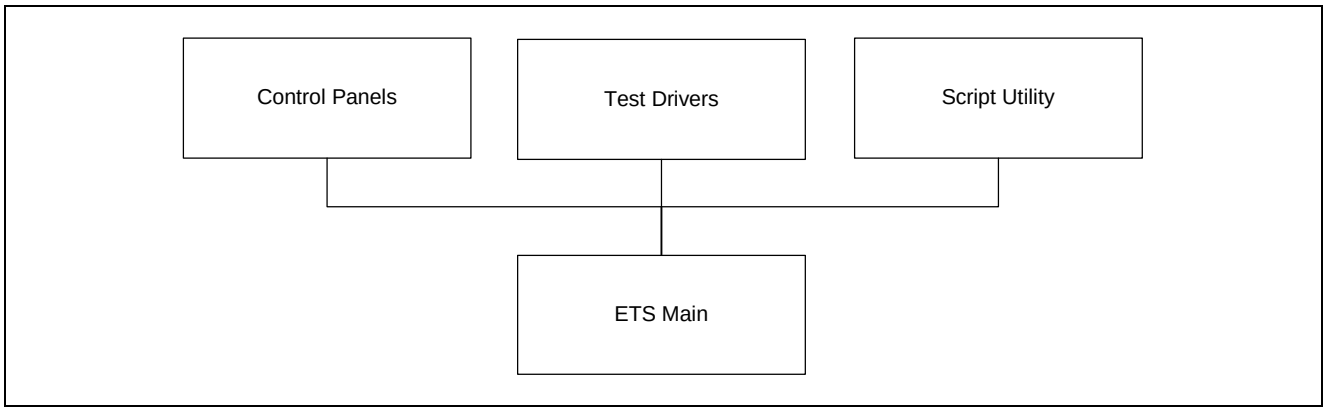


Figure 8. ETS Sub-Module Interfaces

The ETS Main module interfaces to the Script Utility, Control Panels, and Test Drivers through the OLE Automation interface.

5.1 ETS Main

The ETSMAIN.EXE file is the executable that contains all of the communications and message handling functions.

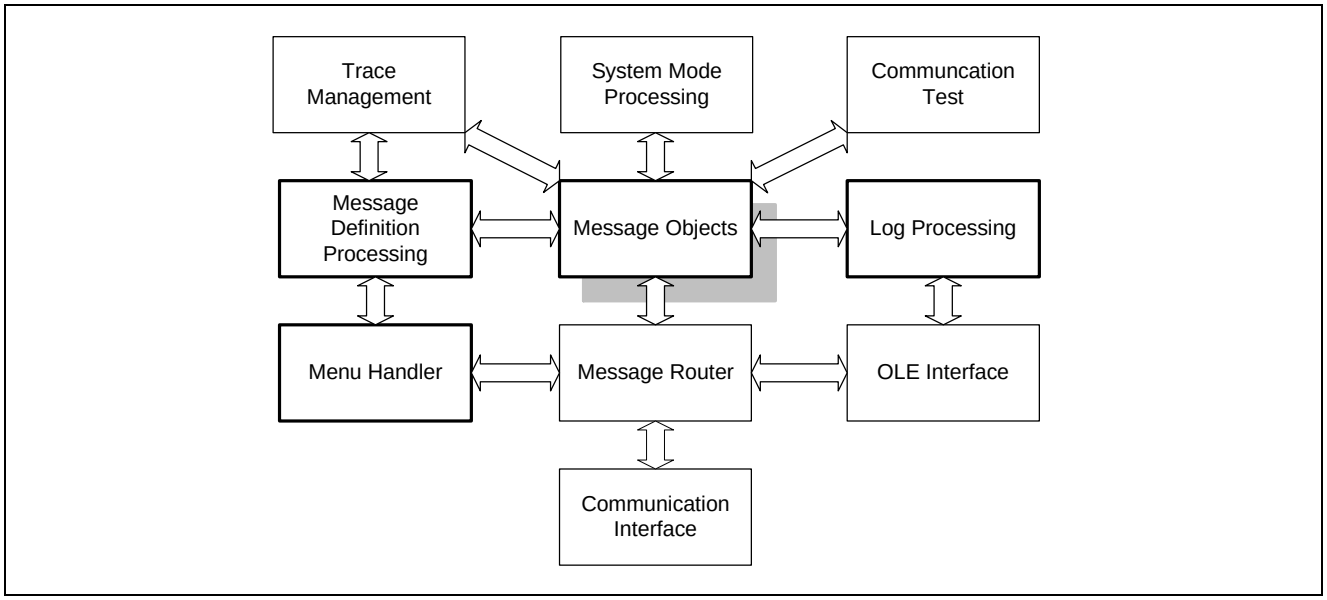


Figure 9. ETS Main Sub-Modules

5.1.1 Command Line Options

ETSMAIN.EXE has several command line options to support various applications. The following is a list of those options.

<filename>	Specify an initial configuration file to be loaded.
/S	Do a full syntax check of the ETS message definitions.
/Port=#	Specify the communication port to connect to.
/WinMode=[norm, min, hide]	Specify the initial display mode of ETS,
/D=<define>	Add a define from the command line.
/SaveDefs=<filename>	Saves the definitions to a file and then terminates ETS.
/LockConfig	Causes the config file to be locked.

```
ETSMAIN.EXE <pathname and config file name> /Port=<port specifier>
            /WinMode=<mode specifier>
```

5.1.1.1 Configuration File

<filename>

Configuration files contain ETS window position information as well as default parameters for command messages. ETS can be manually set up in a specific way and then have that setup saved in a configuration file. Later it can be reloaded to restore the previous setup. When a configuration file is loaded the required spy or trace commands are sent to the CP. Users should NEVER need to modify these configuration files manually.

5.1.1.2 Syntax Check

/S

This option causes ETS to perform a detailed syntax check of the message definitions. Normally ETS will do a limited amount of syntax checking when it loads in order to minimize the amount of time it takes to start ETS.

5.1.1.3 Communication Port

Multiple EtsMains can run concurrently as long as they are connected to different communication ports including Serial and USB. This option overrides the port specification in the system configuration for etsmain.

The port specifier identifies the communication port to be used. The serial ports COM1 through COM12 and USB can be used. The acceptable formats are.

/Port=2

/Port=COM2

/Port=USB2

5.1.1.4 Window Mode

The ETS program can be opened in three different Window modes: Normal, Minimize, and Hidden.

/WinMode=Norm	This is the default
/WinMode=Min	ETS appears in a minimized state.
/WinMode=Hide	ETS will not be visible at all.

5.1.1.5 Defines

/D=<define>

This option is used to start specific ETS configurations from the command line. The define text is used as though it were specified in the Defines dialog. The define will not appear in the Defines dialog nor will it be saved in the registry as would the normal defines. The following example would cause ETS to be loaded in the lite mode without the signaling messages being decoded.

```
EtsMain.exe -d=lite -d=DONT_DECODE_OTA_MSGS
```

5.1.1.6 Save Definitions

/SaveDefs=<filename>

This option saves the definitions to a file and then terminates ETS. It is intended to support automatic generation of specific message definition sets. The following example would cause ETS to be loaded in the lite mode without the signaling messages defined, doing a full syntax check, generate a new definition file and then terminate.

```
EtsMain.exe –s -d=lite -d=DONT_DECODE_OTA_MSGS -savedefs=c:\temp\etsmsg.txt
```

5.1.1.7 Lock Configuration

/LockConfig

Prevents the user from selecting a different configuration file.

5.1.2 Lite Mode Of Operation

ETS program can be run in Full or Lite Mode. Full mode loads all commands, spies, and traces. Lite mode loads only the specified commands, spies, and traces in the text files. The lite mode is enabled in the configuration defines under “Lite Message Set”. This will cause a “LITE” define to occur.

To have a message be included in the Lite mode, add “lite” at the 5th field of the message id definition in the following enuerations:

ETS Id

CP SpyIds

CP TraceIds

DSPM TraceIds

DSPV TraceIds

DSPM SpyIds

DSPV SpyIds

Example:

To enable CP Loopback in Lite mode, the command message line looks like the following:

```
CP Loopback, 0, CR, Debug, Lite
```

5.1.3 Save Definitions

In normal operation ETS loads message definitions from multiple files and applies the definitions in a pre-process to creating the messages. The Save Definitions feature allows the preprocessed messages definitions to be saved in a single file. The single file can be loaded faster that the multiple files with preprocessing applied.

When this is combined with a lite mode of operation the definitions saved are only the lite messages.

The Lite definitions can be saved to a new definition file using the “Save Definitions” menu item under the File menu. Using the lite definition file reduces the ETS load time and resource consumption by a factor of 10 to 20.

One disadvantage of using a saved definition file is that defines are not applied to the definitions.

In order to enable the Save Definitions feature ETS must be loaded with full syntax checking. (Etsmain.exe will have a “-s” switch on the command line.

5.1.4 Message Control Library

The message control library (MCL) contains a set of C++ classes that support message handling. The modules with a bold outline contain components of the MCL. The center box, “Message Objects,” is shaded to indicate there will be many message objects, each having the same interfaces.

5.1.4.1 Message Definition Processing

The processing of message definitions is performed by a combination of custom code and MCL code.

5.1.4.2 Message Object

Message Objects handle the following message-related functions:

- Creation of field objects based on message definitions.
- Validation of field definitions.
- Extracting fields from received messages.
- Generation of transmitted messages.
- Servicing of menu selections.
- Display of command dialog windows.
- Validation of user-entered parameters.
- Display of trace dialog windows.
- Generation of events upon message receipt.
- Generation of log entry strings.
- Interpretation of message transmission commands from the OLE interface.

5.1.4.3 Menu Handler

The Menu Handler is a module that creates menu items and assigns them to specific message objects.

5.1.4.4 Log Processing

The Log Processing module handles the following functions:

- Manages log window display.
- Updates to log file.
- Sends log entries to the OLE automation interface.

5.1.4.5 Trace Management

The trace management module tracks the activity states of trace messages. This module performs the issuing of trace commands.

When a user makes the menu selection to enable a specific trace, the menu item will become “checked” and the trace enable command will be sent.

If a trace message is received when it is not expected a command will be issued to turn the trace off.

When an expected trace message is received it will be decoded. It will then either be printed in the log and or displayed in a dialog. A system menu item will be provided to enable or disable the use of each. The default selection can be overridden as follows. If printing only is selected and the user wants to display the dialog then selecting the same menu item will cause the dialog window for that trace to appear. A trace dialog will have a menu selection to enable or disable printing of log entries.

5.1.4.5.1 Trace Id Zero

A trace command with an id of zero represents all of the traces. The field following the trace id will either be a single boolean value or a variable length bitmap depending on the specific trace message.

When a boolean value is used it either enables or disables all of the traces controlled by the message.

The CP Trace message uses the bitmap format. This is a variable length array of bytes. The first byte represents the number of bytes in the following byte array. Each bit within the byte array specifies the enabling or disabling of a trace. A bit is referenced based on its trace id. Bit zero is the least significant bit. The bit within the byte is the remainder of the trace id divided by 8. The byte offset is the trace id divided by 8.

5.1.5 Communications Interface

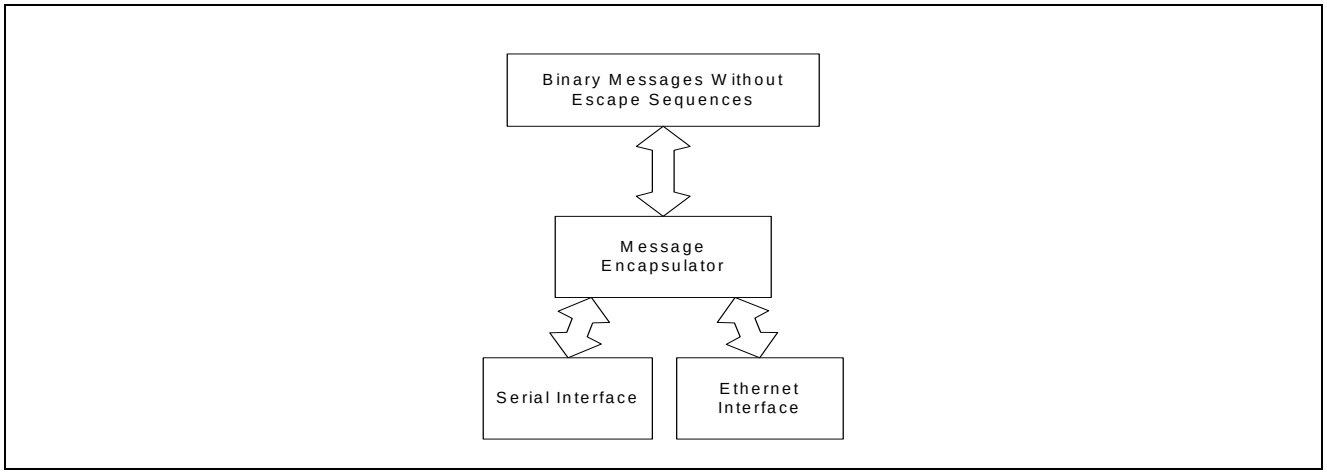


Figure 10. Communications Interface

The communications interface has an associated dialog window that indicates the activity on the active interface. In addition a dialog window will be provided to allow configuration of the communication interface. For

the purpose of automated processing the interface used can be specified from the command line of the ETS Main.

5.1.5.1 Serial Interface

The serial interface sub-module provides a bi-directional byte stream interface through which link layer data is sent.

5.1.5.2 Message Encapsulation

The message encapsulation sub-module translates between the byte stream to the serial interface and the message level. Messages that are to be sent are provided to the communication interface as a byte array. Messages that are received are provided as a byte array for decoding.

The escape sequences and checksums for framing messages are processed based on the direction of transfer.

5.1.6 Message Router

The message router decodes the following and routes the event or information to the correct message object:

- Message Id of received messages
- Menu events
- OLE Automation interface commands

5.1.7 OLE Automation Interface

As entries are made in the log the OLE Automation Interface (OAI) receives a copy of the entry. The OAI is composed of three outgoing queues for commands, responses, and errors. Log entries are placed in the appropriate queue when the queue is enabled.

A command interface is also provided. As commands are received they are sent to the Message Router for decoding and distribution.

System control functions will also be provided. At this time the system modes have not yet been defined.

5.1.8 Specialized Message Handlers

This section describes the operation of the specialized message handlers that are not supported by the MCL.

5.1.8.1 Loopback Test

For the purpose of testing the communication interface, a Loopback message test can be selected using the *Debug/Loopback…* menu item. This test uses the loopback message to send data to the unit and then receive the echoed data. It allows fixed and random data patterns. The test can be run continuously or one message at a time.

5.1.8.2 Printf Message Handler

This message handler processes the *MON_PRINTF_SYS* message in order to simulate the C printf function.

5.1.8.3 Trace Message Handler

The trace message handler processes the *MON_TRACECP_SYS*, *MON_TRACEDSPM_SYS* and *MON_TRACEDSPV_SYS* messages. These messages are declared as “MCL Trace” messages. For each of the trace ids for these messages there is a *PrintFormat* string in the message definition that specifies a Class, Level, and Format string in the following format:

```
<Class>,<Level>,<Format String>
```

Class can be a number or mnemonic. For example:

```
PrintFormat, "One","2", 0x%08X CP - A format string, %d, %d, %d
```

This associates a class of "One" and a level of “2” with a format string of “0x%08X CP - A format string, %d, %d, %d.

Note: The CP Trace messages always have the System Time as the first parameter. That is why the 0x%08x is the first field in the format string for the CP Trace messages.

The C standard printf format string is used. If a field is enumerated a %s must be used. If the field is a scaled numeric a double floating point format like %g must be used. Otherwise a %d or %x would be used. Note that a size modifier such as %ld cannot be used.

5.1.8.4 Database Template Handling

The databases maintained by the CP require a read, modify, write cycle for the user to go through. The database templates define the contents of each of the databases. In addition to a template definition, a list identifies each of the databases.

5.1.8.4.1 Template List

This section defines the list of all DBM templates. The Name of the database, the Id, Segment, offset, number of bytes and the menu to be associated with the database are specified.

```
Begin DbTemplate, Template List
CP DB NAM, DB Id=CP, DB Segment=MMI NAM, Offset=0, Bytes=138, Menu=CP/DB/DB
   &Templates/CP DB NAM
End
```

5.1.8.4.2 Template Definition

A DbTemplate definition needs to be specified for each entry in the Template List. The Template sections define the fields within a database.

```
Begin DbTemplate, CP DB NAM, Template
ESN, uint32
SCM, uint8
EndThe databases maintained by the CP require a read, modify,
```

5.1.9 Decode CDMA Message

The “Decode CDMA Message” menu item appears as a drop down menu when the right mouse button is clicked in a Log window. When Raw Rx signaling message is displayed and selected this menu item will display the details about how a signaling message was decoded.

5.1.10 Reset Board

The “Reset Board” menu item appears within the Debug menu. When it is selected the CDS board will be reset. This is done by toggling the RTS line for 500 ms.

5.1.11 Binary Logging and Playback

A binary log allows a comparatively compressed log to be made of the messages exchanged between the CP and ETS. At the start of the Log the version information of the CP is recorded. The Debug/BinLog menu contains the following menu items.

- Open BinFile…
  - An existing binary log file is opened and played back.
- Open TextFile…
  - An existing textual log file is opened and played back. This will work only if the log file contains Raw Rx and Raw Tx entries.
- Start Logging to BinFile…
  - Creates a binary log file that records all messages sent and received.
- Stop Logging to BinFile
  - Terminates the binary logging and closes the log file.

5.1.12 Flash Download

The flash download dialog is displayed when the “File/Flash Download…” menu item is selected. This is used to download the CPBoot, CP, DSPM , DSPV or SysRel images into flash memory.

5.1.13 External Application Support

An external executable program can be called from the ETSMAIN program by inserting its identified parameters in the ETS Definition text file.

5.1.13.1 External EXE Definition and Register OCX

Insert a Record to the Definition Text file as follows to add IMSIEditor.exe to ETS main menu in Debug pull down submenu”

```
Begin Record, External EXE
      <executable>, Menu=<menu specifier>, Title=<title>
End
```

Executable: Name of the executable file. The file must exist in the same directory that Etsmain.exe is in.

Menu Specifier: Specifies the menu item that will be used to launch the executable. Any menu item can be used. If the menu specifier is identical to one that is already defined for an ETS message it will overwrite the existing menu item.

Title: Specifies the contents of the window Title of the application when run.

Remark: This record can be affected in LITE definition messages by adding “LITE” at the end of the statement.

Example:

```
Begin Record, External EXE
      IMSIEditor.exe, Menu=Debug/External Exe/IMSIEditor, Title=IMSI Editor
End
```

To register an OCX (ActiveX component) like EtsIf.ocx when open ETSMAIN.EXE, Insert a Record to the Definition Text file as follows:

```
Begin Record, Register OCX
      File, <Component>
End
```

Component: The name of an OCX or an ActiveX component needed to be registered.

Example:

```
Begin Record, Register OCX
      File, EtsIf.ocx
End
```

5.1.13.2 External Application Interface

These applications can really be any windows application that interacts with Etsmain.exe through the Etsif.ocx or the autoscript interface.

When Etsmain.exe clears all of its log windows and any external applications are running, it will make the following entry the main log as a response message.

Function: Display=Clear

The external application designer may want to look for this message to clear the display of the external application.

5.1.14 Auto Configuration Loading

The previous loaded ETS Configuration can be automatic loading when a phone (MSUT) is detected and ETS "Auto Config Load" checked box of ETS option dialog was checked.

5.2 Script Utility

The script utility allows commands to be recorded and played back. Recorded sequences can be saved into and restored from a file. The Script commands available are Record, Run, Step, Pause, Continue, Stop and execution speed commands. Section 4.3 describes these operations in detail.

5.3 Control Panels

Control Panels are separate programs that interface to the ETS Main using the OLE automation interface. They provide the end user with a graphical set of buttons, knobs, and gauges that control specific aspects of the operation of the unit under test. Control panels may also interface to devices on a GPIB interface.

The current intent is that these programs be written in Visual Basic. Other languages could also be used.

5.4 Test Drivers

Test Drivers are separate programs that interface to the ETS Main using the OLE automation interface. They perform formal automated tests for design verification or system test. User interfaces for these programs tend to be simpler than that of the control panels and a log file is generated recording the results of the test. The output from these tests could be analyzed using Excel or other spreadsheets.

The current intent is that these programs be written in Visual Basic. Other languages could also be used.

5.5 ETS ActiveX Interface

5.5.1 Overview

This is an ActiveX control in the file ETSIF.OCX. It provides a set of functions that would commonly be needed to develop VB or other programs to interface to the ETS program. The following groups of functions are provided:

Auto Script Interface Layer

Message Flow Support

Get or set field data

Use configuration files and log files

Array field functions

System flag handler

5.5.2 Initialization

The connection to ETSMain is established using the Attach function.

Attach(Port as integer) as boolean.

The Port parameter allows more than one instance of ETSMain to be used at one time. If it is set to zero then any EtsMain that is running will be attached. If the port number is greater than zero then only an EtsMain that is connected to that Serial port will be attached.

GetPortList() as string

A list of the ports that EtsMain instances that are running is returned as a single string whose entries are separated by a new line.

AttachPort(Port as string) as boolean.

The Port parameter should be one of the items in the list provided by the GetPortList method. The port string is of one of two formats, COM# or USB#. Where # is a decimal value representing the port number. Also the “Any” string can be used to connect to any EtsMain instance running. Note that this is the only means of attaching to a EtsMain on a specific USB port.

IsAttached() as boolean

At any time the status of the ETSMAIN connection can be tested with the following member function:

5.5.3 Auto Script Support

All of the AutoScript interface functions are accessible from ETSIF.

The functions that provide input from ETS are as follows:

GetCommand() as string

Retrieves a string from the command queue in the same format as in the log window. If there is no command in the command queue an empty string is returned.

GetCommands as boolean

Enables and disables the command queue. When set to false the queue is emptied and disabled.

GetResponse() as string

Retrieves a string from the response queue in the same format as in the log window. If there is no response in the response queue an empty string is returned.

GetAllResponses() as string

Retrieves all of the strings in the response queue in the form of a list of strings. If there are no responses in the response queue an empty string is returned.

GetResponses as boolean

Enables and disables the response queue. When set to false the queue is emptied and disabled.

GetError() as string

Retrieves a string from the error queue in the same format as in the log window. If there is no error in the error queue an empty string is returned.

GetErrors as boolean

Enables and disables the error queue. When set to false the queue is emptied and disabled.

SendCommand(Cmd as String)

Sends the command string provided. The format is identical to the log window entry that gets made when the command is manually sent.

The received raw hex display of a message can be provided as a command string. These are lines that start with "Raw Rx:".

SetRspQSize(Size as Long)

Sets the size of the response queue. By default the queue size is set to 100 messages. By setting this value to a larger number, bursts of messages can be handled in a delayed manner.

5.5.4 Received Message Flow

Two methods are provided to support receiving messages from ETSMain. One involves waiting for a specific message. The other uses message events.

5.5.4.1 Asynchronous Message Flow Functions

WaitResponse(Seconds as integer) as string.

Waits until either the next response message is received or the number of seconds specified has elapsed. An empty string is returned under the timeout condition.

FindResponse(Seconds as integer, Pattern as string) as string.

Waits until a message is received that matches the specific response in Pattern or the timeout occurs. If a timeout occurs an empty string is returned. The Pattern is a string that matches the leading characters of the message string.

CancelFlag as boolean

This flag is used to prematurely abort a *WaitResponse* or *FindResponse* call. When set to true *WaitResponse* or *FindResponse* will complete immediately.

5.5.4.2 Message Event Functions

The message events are used to process messages with less latency than the other methods. Events can be used for Commands, Responses, and Errors. In addition the responses which generate events can be filtered.

The ETSIF ActiveX control has a Timer that it uses to poll ETSMain for messages during event generation. This means that the event threads are not part of ETSMain.

Members

EvFilterAdd(Message as string)

Call this function to add a string to the event filter list. Only those messages that are in the event filter list will generate events. The string that is provided must match the leading characters of the message text.

EnableEvents as boolean

Set this to true or false to enable or disable message event generation.

EvFilterClear()

Call this function to remove all entries from the event filter list. When the event filter is cleared all received messages will generate an event.

Events

OnCommand(Message as string)

OnResponse(Message as string)

OnError(Message as string)

Each of these event calls is made when the Command, Response, or Error occurs.

5.5.5 Field Data Functions

Recall that the name-value pair format is used for field and field content specification. An example is:

Aname1=Avalue1, Aname2=Avalue2, etc.

The functions *GetField* and *SetField* support this format.

GetField(Response as string, Field as string) as string

Use this function to extract field values from a response message. The Response parameter is the response as received from ETSMain. The Field parameter is the field name as it appears in the response message. The value of the selected field is returned.

SetField(Command as string, Field as string, Value as string) as string

The easiest way to send command messages is to manually generate the command, copy the command string, and save it in a string variable. Then use this function to modify the command string to set the fields to the specific values desired.

The returned string is the modified command string.

5.5.6 Array Field Functions

Arrays are logged as a space delimited list of numbers that are either in decimal, hex, or double-precision floating point format. The ArrayMgr is an ActiveX object within ETSIF. It contains the following functions to support VB in accessing and generating array string fields:

Load(Field as string)

LoadDouble(Field as string)

Internally an ArrayMgr object contains a list of numbers. This function interprets the array field value and translates it into the list of numeric values.

MakeField(Hex as boolean, ByteCount as integer, Signed as boolean) as string

MakeFieldDouble() as string

The function above generates an array field string from the internal numeric list. The Hex parameter determines if the list will be in hex or decimal format. The `ByteCount` parameter determines the number of bytes in each numeric value generated. When the Hex parameter is false the Signed parameter determines if the numeric values will be signed or unsigned decimal.

```
AddNum( Number as long )
AddDouble( Number as double )
```

Adds the number to the internal list of numeric values.

```
Clear()
```

Empties the internal list of numeric values.

```
GetNum( Index as integer ) as long
GetDouble( Index as integer ) as double
```

Returns a specific numeric value from the list.

```
GetSize() as long
```

Returns the number of entries in the numeric list.

5.5.7 File Functions

The functions described in this section allow access to the file interface functions of ETSMain.

```
OpenConfig( File as string )
```

The configuration file is loaded when this call is made.

```
OpenLog( File as string, OverWrite as boolean ) as boolean
```

This function opens a log file. If the file exists it can be overwritten or appended to.

```
CloseLog()
```

The log file is closed by this function call.

```
OpenTraceLog( File as string, OverWrite as boolean ) as boolean
```

This function opens a trace window log file. If the file exists it can be overwritten or appended to.

```
CloseTraceLog()
```

The trace window log file is closed by this function call.

```
LogNote( Note as string )
```

A note is entered in the log window.

```
LogNoteExt( Note as string, Main as boolean, Trace as boolean, Fault as
   boolean )
```

A note is entered in the log windows that are enabled in the parameter.

```
void EnableTracePrint(boolean Enable);
```

The entry of trace messages into the main log window can be enabled and disabled with this function. This operates in the same manner as the user selecting "Print to Main Log" in the popup menu of the trace log window.

```
void EnableExtraLogPrint(boolean Enable);
```

The entry of extra log window messages into the main log window can be enabled and disabled with this function. This operates in the same manner as the user selecting "Print to Main Log" in the popup menu of all of the extra log windows.

```
void PauseLogWindows( boolean Pause)
```

Pauses or removes pause from all log windows.

5.5.8 Command Function

This function is one of the ETS Interface methods that sends command message to the CBP board.

```
void  SendCommand(Cmd as string)
```

For example:

Send 0x55, one byte of data for loop back test:

```
Cmd  = "ETS, Id= CP Loopback, Data=0x55"
```

Check or uncheck Tx from ETS main Menu:

```
Cmd  = "Menu File /Raw /Tx"
```

5.5.9 Serial Communication Functions

These functions allow changing of the PC Serial Communication functionalities:

```
Boolean  SetBaudRate(Baud as integer) as boolean
```

The PC Serial Communication Baud Rate can be set by using the `SetBaudRate` function.

```
Boolean  SetCommPort(nCommPort as integer, bOnFlg as boolean) as boolean
```

The PC Serial Communication Port can be set by using the `SetCommPort` function.

5.5.10 ETS Control Functions

```
Void  CloseEts()
```

To close ETSMAIN program, use the CloseEts function.

5.5.11 AutoRspQ

The AutoRspQ provides a mechanism for multiple applications to received messages from ETS without interfering with each other. The primary example of this is the Virtual MMI application that is used concurrently with other applications that are receiving messages from ETS.

5.5.11.1 Function Create(pFilter As String, pName As String) As Long

The create function initializes a AutoRspQ by defining the messages that are to be delivered by the AutoRspQ.

pFilter – A newline separated list of ETS messages that are to be received.

pName – The unique name that is to be associated with the AutoRspQ.

5.5.11.2 Function Destroy() As Boolean

This function must be called when the use of the AutoRspQ is completed.

5.5.11.3 Function Enable(Enable As Boolean) As Boolean

This function enables or disables the delivery of messages in the Queue. Disabling the queue and then enabling it will flush all entries from the queue.

5.5.11.4 Function GetAllResponses() As String

Returns a new line separated list of all the responses currently in the queue. This causes all of the messages to be removed from the queue. Use this function when higher rates of messages are expected because the number of out of processes calls to ETS will be minimized.

5.5.11.5 Function GetOverRunCount() As Long

Returns the number of messages that could not be put in the queue because space was not available. The overrun count will be set to zero after this function is called.

5.5.11.6 Function GetResponse() As String

Returns the next message in the queue. The returned message is removed from the queue.

5.5.11.7 Sub SetSize(Size As Long)

Sets the maximum number of messages that the queue will contain. If this function is not called the maximum number of message allowed in the queue is 100.

5.6 VIAIF Dynamic Link Library

The ViaIf.dll provides the functionality of the EtsIf.ocx without the user interface of ETS. In addition the interface is provided as an in process COM connection because it is a DLL.

The main interface of the ViaIf.dll is identical to EtsIf.ocx. A test program ViaIfTest is provided in Visual Basic that demonstrates the compatibility between EtsIf.ocx and ViaIf.dll. Because the ViaIf.dll is not connected to ETS it requires a second interface to setup the communication with the phone as well as to select the message definitions to be used.

5.6.1 ViaSetup Interface

The ViaSetup interface allows the application to select definition files and and communications ports as well as other functions as listed below.

5.6.1.1 LoadDefinitions

```
LoadDefinitions([in] BSTR pPathName);
```

Loads the definition file specified in the pPathName parameter.

5.6.1.2 OpenCommInterface

```
OpenCommInterface([in] BSTR pInterfaceIdentifier, [in] BSTR pParameters);
```

Opens the selected Communications Interface.

pInterfaceIdentifier – “COM1, COM2,… or USB.

pParameters – Used only when COM# is selected. The string must contain the following:

Port identifier,

Baud Rate,

0 or 1 to enable or disable the TxPreamble

Preamble delay in milliseconds.

Example: “COM1, 115200, 1, 20” Open COM1 at 115200 baud, enable the TxPreamble with 20 milliseconds.

5.6.1.3 UnloadDefinitions

```
UnloadDefinitions();
```

Unloads the definitions that were loaded with the LoadDefinitions call.

5.6.1.4 CloseCommInterface

```
CloseCommInterface();
```

Closes the communications interface.

5.6.1.5 GetCommStatus

```
GetCommStatus([out, retval] BSTR * pStatus);
```

Returns the communications status. For serial ports this is in the form “COM1 115200 N,8,1”. For the USB port this is in the form "USB - NOT Connected" or “USB - Connected”.

5.6.1.6 GetCommStatistics

```
GetCommStatistics([out, retval] BSTR * pStats);
```

This provides the statistics since either a connection was established or since the last call the ResetCommStatistics. It provides the following information in a string.

```
“STATS, SampleTime=0, MsgsRcv=0, MsgsXmt=0, MsgRcvErrors=0, MsgXmtErrors=0,
   BytesRcv=0, BytesXmt=0, DrvRxErrors=0, DrvTxErrors=0, DrvRxHighWater=0,
   DrvTxHighWater=0, SerFramingErrors=0, SerOverrunErrors=0,
   SerParityErrors=0”
```

5.6.1.7 ResetCommStatistics

```
ResetCommStatistics();
```

Resets the statistics provided in the GetCommStatistics call.

5.6.1.8 GetAutoRspQ

```
GetAutoRspQ([out, retval] IAutoRspQ ** ppAutoRspQ);
```

Returns a reference to AutoRspQ object. This is identical to the AutoRspQ provided in the EtsIf.ocx interface.

5.6.1.9 RawHex

```
RawHex([in] BOOL TxHex, [in] BOOL RxHex);
```

Enables or disables logging of Raw Hex information when logging is active.

5.6.1.10 ResetPolarity

```
ResetPolarity([in] BOOL Positive);
```

Sets the ResetPolarity. If this call is not made the reset polarity will be negative which is compatible with the CDS4 boards.

5.6.1.11 ResetBoard

```
ResetBoard();
```

Performs a board reset using the CTS serial interface line. This will only have an effect if the active interface is a serial port. The CTS line is used in the CDS boards to allow an application to cause a hard reset of the board.

5.6.1.12 SendDeferredCommand

```
SendDeferredCommand([in] long Milliseconds, [in] BSTR pCommand);
```

This works the same as the SendCommand call in the main interface except that the command will not be sent until the number of milliseconds specified has elapsed.

5.6.2 ViaBinary Interface

The ViaBinary interface allows applications to bypass the translation of messages between textual and binary formats. Its purpose is to reduce the processor loading caused by the message translation process. This has proven very useful when downloading large quantities of data.

The design provides a filter mechanism to identify which received messages are to be put into a RxQueue. The RxQueue is read using the GetMessage function.

5.6.2.1 SendMessage

SendMessage([in] VARIANT * pSafeByteArray);

Sends a binary message to the CP.

5.6.2.2 GetMessage

GetMessage([in] long Milliseconds, [out, retval] VARIANT * pSafeByteArray);

Retrieves a message from the CP. The mount of time to wait is specified in Milliseconds. If this parameter is set to zero the function will return immediately if a message is not in the queue. When a message is returned by this function it is also deleted from the RxQueue.

5.6.2.3 Clear Filter

ClearFilter();

Clears the received message filter

5.6.2.4 AddToFilter

AddToFilter([in] VARIANT * pSafeByteArray);

Adds a message header to the Filter. Any received message that matches this binary header will be available through the GetMessage function.

5.6.2.5 ClearRxQueue

ClearRxQueue();

Removes all messages from the receive queue.

5.7 ViaHub

The ViaHub.exe program provides two functions. One is USB interface management the other is as a common connection point for applications using EtsMain.exe through EtsIf.ocx.

5.7.1 USB Connection Management

When a phone is connected to a USB port the ViaHub.exe associates it with a port number. If the physical USB port had not been used by a Via phone then the ViaHub.exe will assign the next port number to it. Otherwise the ViaHub will assign the USB port number that was first assigned.

5.7.2 Application Access

When an instance of EtsMain.exe is active it will register itself with the ViaHub. This lets the ViaHub know which port it is connected to. Through EtsIf.ocx applications can access this information. The GetPortList method returns a list of all of the ETS instances that are connected to phones. The AttachPort method allows applications to access specific ETS instances. This is all done through the ViaHub.exe.

5.7.3 Command Line Options

ViaHub.exe /Delay=<seconds>, /ClearUsbPorts

5.7.3.1 /ClearUsbPorts

This option causes all of the previous USB port assignments to be removed. USB ports are then assigned starting with the value 1 and incrementing by one for each new port that is used. This only needs to be done when the operator wants to change the port assignments.

5.7.3.2 /Delay

This option is used for devices that have an intermittent initial connection. It causes the ViaHub to wait for the device to stay connected for <seconds> before it is considered to be connected. If this option is omitted then there is no connection delay.

5.8 ETS Message Commands

This section describes messages that can be sent to EtsMain.exe or ViaIf.dll that are not sent to the phone. Instead they are interactions with EtsMain.exe or ViaIf.dll themselves.

5.8.1 BINLOG Messages

BINLOG Messages start with “BINLOG” for both commands and responses.

The followings are the formats for BINLOG commands and responses. The messages are described as the command (Cmd) to be sent to ETS or ViaIf.dll and the response (Rsp) that is returned.

5.8.1.1 Create Binary Log File

This message pair is used to create a binary log file.

Cmd: "BINLOG, Cmd=Create, Path=<pathname>"

Rsp: "BINLOG, Cmd=Create, Status={Ok|Fail}, Cause=<cause>"

5.8.1.1.1 Command Parameters

Path=<pathname>	The pathname of the file to be created.

5.8.1.1.2 Response Parameters

Status={Ok|Fail}	If it is Ok then the binary file has been created and is being written to. If it is Fail then the file did not open for some reason.
Cause=<cause>	Specify the failure reason if Status = {Fail}

5.8.1.2 Close Binary Log File

This message pair is used to close a binary log file.

Cmd: "BINLOG, Cmd=Close"

Rsp: "BINLOG, Cmd=Close, Status={Ok|Fail}, Cause=<cause>"

5.8.1.2.1 Command Parameters

None.

5.8.1.2.2 Response Parameters

Status={Ok|Fail}	If it is Ok then the binary file has been closed. If it is Fail then the file did not close for some reason.
Cause=<cause>	Specify the failure reason if Status = {Fail}

5.8.1.3 Get Status of Binary Log File

This message pair is used to get the status of a binary log file.

Cmd: "BINLOG, Cmd=GetStatus"

Rsp: "BINLOG, Cmd=GetStatus, Status={Idle|Logging|Playback|Fail}, Path=<pathname>"

5.8.1.3.1 Command Parameters

None.

5.8.1.3.2 Response Parameters

Status={ Idle|Logging|Playback|Fail }	If it is Idle then no binary file has been created. If it is Logging then a binary file is created and being written to. If it is Playback then a binary file is being playback. If it is Fail then a failure status occurs for some reason.
Cause=<cause>	Specify the failure reason if Status = {Fail}
Path=<pathname>	The pathname of the binary log file if there's one.

5.8.1.4 Playback Binary Log File

This message pair is used to playback a binary log file

Cmd: "BINLOG, Cmd=Playback, Path=<pathname>"

Rsp: "BINLOG, Cmd=Playback, Status={Ok|Fail}, Cause=<cause>"

5.8.1.4.1 Command Parameters

Path=<pathname>	The pathname of the file to be playback.

5.8.1.4.2 Response Parameters

Status={Ok|Fail}	If it is Ok then the binary file has been playback. If it is Fail then the file can not be playback for some reason
Cause=<cause>	Specify the failure reason if Status = {Fail}

5.8.2 GETDEF

Add support to access Message Definition for EtsMain.exe and ViaIf.dll. Both Command and Response string start with “GETDEF”. The following are the examples for GETDEF commands and responses. The messages are described as the command (Cmd) to be sent to ETS or ViaIf.dll and the response (Rsp) that is returned.

5.8.2.1 Get Definition Entry Count

This message pair is used to get Definition Entry count

Cmd: "GETDEF, {Record|Message|Enum|…} = <definition name>

Rsp: "GETDEF, Entries=<number of entries>, <parameters>"

5.8.2.1.1 Command Parameters

{Record|Message|Enum|...} = <definition name>	The name of the Definition[11].

5.8.2.1.2 Response Parameters

Entries=<number of entries>	The number of entry in this Definition file
<parameters>	Returned parameters if there’re any

5.8.2.1.3 Example

Example of a Record Definition[12]:

Begin Record, ETS Definition Version

Project ,CBP4.05

Ets Version ,5.4.0

End

In this example,

Cmd: "GETDEF, Record = ETS Definition Version"

Rsp: "GETDEF, Entries = 2”

5.8.2.2 Get Definition Entry Value by Entry Number

This message pair is used to get Definition Entry Value by Entry number

Cmd: "GETDEF, {Record|Message|Enum|...} = <definition name>, Entry = <entry number>

Rsp: "GETDEF, <entry>"

5.8.2.2.1 Command Parameters

{Record|Message|Enum|...} = <definition name>, Entry = <entry number>

The Entry number that is used to retrieve entry value.

5.8.2.2.2 Response Parameters

<entry> The Entry value.

5.8.2.2.3 Example

In the above example:

Cmd: "GETDEF, Record = ETS Definition Version, Entry = 0"

Rsp: "GETDEF, Project, CBP4.05"

5.8.2.3 Get Definition Entry Value by Entry Name

This message pair is used to get Definition Entry value by Entry name.

Cmd: "GETDEF, {Record|Message|Enum|...} = <definition name>, Entry = <entry name>

Rsp: "GETDEF, <entry parameters>

5.8.2.3.1 Command Parameters

{Record|Message|Enum|...} = <definition name>, Entry = <entry name>

The Entry name that is used to retrieve entry value.

5.8.2.3.2 Response Parameters

<entry parameters> The Entry value.

5.8.2.3.3 Example

In the above example:

Cmd: "GETDEF, Record = ETS Definition Version, Entry = Project"

Rsp: "GETDEF, CBP4.05"

5.9 Download DLL

The download.dll is a COM object that encapsulates the methods needed to do the bulk data transfers to and from a CBP processor. Bulk transfers include the following.

- Flash Download
- Block Db Download
- Block Db Upload
- Data Download
- Data Upload

The download.dll uses the ViaIf interface to communicate with the CBP. The download.dll is used by an application that also uses the ViaIf. The application that creates an instance of the download.dll will provide the download.dll with a reference to a usable ViaIf object using the Connect method.

5.9.1 IDownloader interface

5.9.1.1 Methods

5.9.1.1.1 Connect

This call provides the instance of the downloader with a reference to the ViaIf interface. Note that this needs to be one of the first calls to the download.dll.

BOOL Connect(LPDISPATCH pViaIf);

pViaIf: A dispatch pointer to the ViaIf object to be used. The downloader requires that this object is already connected to a port and has the definitions loaded.

Return True if successful otherwise false. When false is returned the GetLastError function provides a description of the error.

5.9.1.1.2 Disconnect

Releases the reference to the ViaIf.dll as well as any other resources.

BOOL Disconnect();

Return: True if successful otherwise false. When false is returned the GetLastError function provides a description of the error.

5.9.1.1.3 GetLastError

Retrieves the string representing the last error that occurred.

BSTR GetLastError();

Return: The string representing the last error that occurred.

5.9.1.1.4 FlashDownload

Downloads the selected processor with the specified file.

BOOL FlashDownload(BSTR Processor, BSTR Path);

Processor: The processor name as specified in the ETS definition file under the section “Begin Enum, Flash Program Sections”. Note that additional processors or sections can be added to this list.

Path: The full pathname of the file to be downloaded.

Return: True if successful otherwise false. When false is returned the GetLastError function provides a description of the error.

5.9.1.1.5 WaitForBootToLoader

Waits for the Boot To Loader message to be sent from the CBP and then responds with a Boot To Loader message.

BOOL WaitForBootToLoader();

Return: True if successful otherwise false. When false is returned the GetLastError function provides a description of the error.

5.9.1.1.6 GetState

Returns the string representing the current state of the downloader.

BSTR GetState();

Return: The state of the downloader.

5.9.1.1.7 GetProgress

long GetProgress();

Return: Returns a number representing the progress of the transfer. This can be compared to the value returned by the GetTotal call to determine the percent completion.

5.9.1.1.8 GetTotal

long GetTotal();

Return: Returns the total number of bytes that will be transferred in the current operation.

5.9.1.1.9 Stop

Causes the active transfer to terminate.

BOOL Stop();

Return: True if successful otherwise false. When false is returned the GetLastError function provides a description of the error.

5.9.1.1.10 IsConnected

BOOL IsConnected();

Return: True if successful otherwise false. When false is returned the GetLastError function provides a description of the error.

5.9.1.1.11 DataDownload

BOOL DataDownload(BSTR Path, long Address, long Words);

Path: The full pathname of the file to be downloaded.

Address: The address in the CBP memory that the downloaded data should be sent to.

Words: The number of 16 bit words that will be downloaded.

Return: True if successful otherwise false. When false is returned the GetLastError function provides a description of the error.

5.9.1.1.12 DataUpload

BOOL DataUpload(BSTR Path, long Address, long Words);

Path: The full pathname of the file to be uploaded.

Address: The address in the CBP memory that the uploaded data should be read from.

Words: The number of 16 bit words that will be uploaded.

Return: True if successful otherwise false. When false is returned the GetLastError function provides a description of the error.

5.9.1.1.13 BlockDbDownload

The most common use of this function will be to download the PRL. In addition any of the block databases can be downloaded that are listed in the ETS definition for “Begin Enum, Block Database List Write”.

BOOL BlockDbDownload(BSTR Path, BSTR Database);

Path: The full pathname of the file to be downloaded.

Database: The block database to be downloaded.

Return: True if successful otherwise false. When false is returned the GetLastError function provides a description of the error.

5.9.1.1.14 BlockDbUpload

The most common use of this function will be to download the PRL. In addition any of the block databases can be downloaded that are listed in the ETS definition for “Begin Enum, Block Database List Read”.

BOOL BlockDbUpload(BSTR Path, BSTR Database);

Path: The full pathname of the file to be uploaded.

Database: The block database to be uploaded.

Return: True if successful otherwise false. When false is returned the GetLastError function provides a description of the error.

5.9.1.2 Properties

5.9.1.2.1 TransferWindow

The TransferWindow refers to the number of transfer messages that can be sent before an acknowledgement is received. For the highest data transfer rate this should be set to the largest value supported by the CBP for the transfer being made.

short TransferWindow;

5.9.1.2.2 EraseTimeoutMs

This parameter defines how long the downloader will wait for the CBP to respond to the initial flash download command that causes a flash memory section to be erased.

long EraseTimeoutMs;

5.9.1.2.3 StatusPeriodMs

This parameter defines how often the Status event will be send to the calling program. Set this value before the transfer actually starts.

long StatusPeriodMs();

5.9.1.3 IDownloaderEvents Interface

A single method in this interface provides the status information needed to update a progress display.

HRESULT Status([in] BOOL Active, [in] long TotalBytes, [in] long TransferredBytes, [in] BSTR State);

Active:	This is True when a Transfer is in progress or if the downloader is otherwise in the process of doing something.
TotalBytes:	The total number of bytes that will be transferred.
TransferredBytes:	The number of bytes that have been transferred.
State:	The State of the Transfer.

6 Adding Commands, Traces, Spies, Faults

The purpose of this section is to provide instructions on how to add new ETS commands, traces, spies, and fault messages to ETS text files and CP software. All examples herein pertain to the MMI sub-unit. These instructions, however, can also be used to add new ETS commands, traces, spies and fault messages for other sub-units by replacing MMI with the other unit's name wherever appropriate.

The approach taken is to examine specific commands, traces, spies, faults have been integrated into ETS text files and CBP CP (.c, .h) files. Note that all numeric I.D.s occurring within square brackets, such as [1109] in section 6.1.1.1.1 can change between versions of software. However, these numeric I.D.s must always be consistent between the ETS text files and the CBP CP (.c, .h) files.

6.1 ETS Commands

ETS commands are used to pass data to the CBP software.

6.1.1 Adding an ETS Command

6.1.1.1 Modify ETS Message Id and Message Body Definition Text Files

On the ETS side, the file that defines all ETS commands is **'msg_id.txt'.** The command message body definitions are defined in each sub-unit **'<unit>_msg.txt**' file and are included by **'etsmsg.txt'.** We will now describe the steps for modifying both of these ETS text files.

6.1.1.1.1 Declare the New Command in 'msg_id.txt'

Open the file '/**msg_id.txt**'. This file contains several sub-unit menu declarations, including MON, HWD, etc. Scroll down to the 'MMI Sub Menu' section in 'msg_id.txt' and add a new declaration for the new command.

Example:

```
CP Test Message, [1199], CO, CP/MMI
```

The hard-coded value in square bracket [1199] is the value of the command ID and is optional. The logic works like a C language enumeration. If a value is not present, the command ID will be that of the previous command plus one.

6.1.1.1.2 Add the Command Message Body to '<unit>_msg.txt'

Open the file **'/cp/mmi/mmi_msg.txt'.** Go to the 'Start MMI Definitions' and add the command message body.

Example:

```
Begin Message, ETS CP Test Message Cmd
   Count,     uint8,  base=10
End
```

6.1.1.2 Modify CBP CP Software Files

In the CBP CP software, adding an ETS command involves declaring the new ETS command ID in a global header file, routing the new command ID to the CP task which is expected to process the message, prototyping the new message structure in that task's api file, and adding a message handler in its main message switch. These steps are described below.

6.1.1.2.1 Declare New Command ID in <inc/iopets.h>

Add the new command id to the 'IopEtsMsgIdT' enumeration. Note that the same number, [1199] is used here as in the ETS **'msg_id.txt'** file.

Example:

```
IOP_CP_TEST_MESSAGE_ETS=1199,
```

6.1.1.2.2 Add Routing for the New Command Message in <iop/ioprtabl.c>

Add the routing for the new message id to the 'IopRoutingTable'.

1. The value in the right-most column in the table indicates whether an ACK to ETS is needed. If it is TRUE, an ACK is needed, and the IOP task will perform the ACK.
2. The value in the (right-most – 1) column in the table indicates whether a response to ETS is needed. If it is TRUE, a response is needed and the IOP task will add the response information (the SysRspMsgT structure) to the message that is routed to the MMI task. If it is FALSE, a response is not needed.

   Example:

```
IOP_CP_TEST_MESSAGE_ETS,  EXE_MMI_ID,  MMI_MAILBOX,  MMI_TEST_MESSAGE_MSG, FALSE,
FALSE,
```

6.1.1.2.3 Add the New Message Structure to <inc/mmiapi.h>

A message structure that matches exactly to the one defined by the ETS command message body (see section 6.1.1.1.2) must be defined in mmiapi.h. The message handler that carries out this new ETS command will operate on this message structure. The C preprocessor directive ‘PACKED’ should always be used in defining ETS message structures because in ETS, message bodies are always packed.

Example:

```
typedef PACKED struct
{
   uint8       Count;
} MmiTestMessageMsgT;
```

6.1.1.2.4 Add Message Handler in Mmitask.c

In the body of the main task switch statement add a case to handle the new message id.

Example:

```
      case MMI_TEST_MESSAGE_MSG:
         MmiTestMessage (MsgDataP, MsgSize, MsgId);
         break;
```

6.1.2 Executing an ETS Command

From the ETS Main application click on CP\MMI\’Test Message’ to open up the new command window. Whenever the new command needs to be sent click on the 'Send' button on its window. The main log window will log the new message that has been just sent to the CP.

Note: Whenever any Ets *.txt file is modified, the ETS application has to be restarted before the changes will take effect.

6.2 ETS Traces

6.2.1 Adding an ETS Trace

ETS traces are typically used to track code execution.

6.2.1.1 Modify ETS Message Id and Message Body Definition Text files

On the ETS side, the file that defines all ETS commands is **'trace_id.txt'.** The command message body definitions are defined in each sub-unit **'<unit>_trace.txt**' file and are included by **'etsmsg.txt'.** We will now describe the steps for modifying both of these ETS text files.

6.2.1.1.1 Declare the New Trace in trace_id.txt

Open the file '/**trace_id.txt**'. This file contains several sub-unit menu declarations, including MON, HWD, etc. Scroll down to the 'MMI Sub Menu' section in 'trace_id.txt' and add a new declaration for the new trace.

Example:

```
CP MMI Test Msg, [0x263], CT, CP/Trace/MMI
```

Again, the trace message id in the square brackets is optional.

6.2.1.1.2 Add the Trace Message Body to <mmi/mmi_trace.txt>

Open the file **'/cp/mmi/mmi_trace.txt'.** Go to the 'Start MMI Definitions' and add the trace message body. Note that the MON task in the CP software that handles traces automatically adds a 32-bit system time variable to the top of the trace data structure. Therefore, the trace message body will have to include the variable "Sys Time, uint32" as its first argument.

Example:

```
Begin Message, CP Trace CP MMI Test Msg Rsp
   PrintFormat, 1, 1, 0x%08X CP - MMI VRec Msg: MsgId=%s, NewState=%s
   Sys Time,  uint32
   MsgId,     uint32
End
```

6.2.1.2 Modify CBP CP Software Files

In the CBP CP software, adding an ETS trace involves declaring the new ETS trace ID in a global header file and adding calls to MonTrace() to pass the desired information to ETS.

6.2.1.2.1 Declare New Trace ID in <inc/monids.h>

Add the new trace id to the 'MonTraceIdT' enumeration.

Example:

```
MON_CP_MMI_TEST_MSG_TRACE_ID,
```

6.2.1.2.2 Add MonTrace() Calls to Source Code

In the C source files, a 'MonTrace()' call can be added to send trace information to ETS wherever such information is desired.

The prototype for 'MonTrace' is defined as follows:

```
extern void MonTrace(uint16 TraceId, uint32 NumArgs, ...);
```

Example:

```
/* display the incoming message id and the Test Message */
MonTrace (MON_CP_MMI_TEST_MSG_TRACE_ID, 2, (uint32) MsgId);
```

6.2.2 Enabling an ETS Trace

From the ETS Main application click on CP\TRACE\MMI\'trace name' to turn on or off a trace where 'trace name' is the name of the trace. In the above example, 'trace name' is 'CP MMI Test Msg'. The traces received from the CP will be displayed in the Trace window, only if the Trace is enabled through ETS; ETS passes a message to the CP MON task, when a trace is enabled by the user. By default all traces are turned OFF.

Note: Whenever any ETS *.txt file is modified, the ETS application has to be restarted before the changes will take effect.

6.3 ETS Spies

6.3.1 Adding an ETS Spy

ETS spies are typically used to display changes to data.

6.3.1.1 Modify ETS Message Id and Message Body Definition Text Files

On the ETS side, the file that defines all ETS commands is **'spy_id.txt'.** The command message body definitions are defined in each sub-unit **'<unit>_spy.txt**' file and are included by **'etsmsg.txt'.** We will now describe the steps for modifying both of these ETS text files.

6.3.1.1.1 Declare the New Spy in 'spy_id.txt'

Open the file '/**spy_id.txt**'. This file contains several sub-unit menu declarations, including MON, HWD, etc. Scroll down to the 'MMI Sub Menu' section in 'spy_id.txt' and add a new declaration for the new spy.

Example:

```
CP Mmi Test Spy, [0x1C1], CT, CP/Spy/MMI
```

Again, the spy id in the square brackets is optional.

6.3.1.1.2 Add the Spy Message Body to <mmi/mmi_spy.txt>

Open the file **'/cp/mmi/mmi_spy.txt'.** Go to the 'Start MMI Definitions' and add the spy message body. Note that the MON task in the CP software that handles spies automatically adds a 32-bit system time variable to the top of the spy data structure. Therefore, the spy message body will have to include the variable "Sys Time, uint32" as its first argument.

Example:

```
Begin Message, CP Spy CP Mmi Test Spy Rsp
   Sys Time,   uint32
   Test Data, array, type=int16, maxsize=64, size=end, columns=60
End
```

6.3.1.2 Modify CBP CP Software Files

In the CBP CP software, adding an ETS spy involves declaring the new ETS spy ID in a global header file and adding calls to MonSpy() to pass the desired information to ETS.

6.3.1.2.1 Declare New Spy ID in <inc/monids.h>

Add the new command id to the 'MonSpyIdT' enumeration.

Example:

```
MON_CP_MMI_TEST_SPY_ID = 0x1C1,
```

6.3.1.2.2 Add MonSpy() Calls to Source Code

In the CBP CP source files, a 'MonSpy()' call can be added to send spy information to ETS whenever such information is desired.

The prototype for 'MonSpy' is defined as follows:

```
extern void MonSpy(uint16 SpyId, uint8 *DataP, uint32 NumBytes);
```

Example:

```
/* display the first 8 bytes of the template data */
MonSpy (MON_CP_MMI_TEST_SPY_ID, (uint8 *) Test.Template.ptr, 8);
```

6.3.2 Enabling an ETS Spy

From the ETS Main application click on CP\SPY\MMI\'spy name' to turn on or off a spy, where 'spy name' is the name of the spy. In the above example, 'spy name' is ‘CP Mmi Test Spy’. Each spy will have its own display window. The spies received from the CP will be displayed in the relevant Spy window, only if the Spy is enabled through ETS; ETS passes a message to the CP MON task, when a Spy is enabled by the user. By default all spies are turned OFF.

Note: Whenever any ETS *.txt file is modified, the ETS application has to be restarted before the changes will take effect.

6.4 ETS Faults

6.4.1 Adding an ETS Fault

ETS Faults are used to signal error conditions, which are typically NOT expected to occur.

6.4.1.1 Modify ETS File mmi/mmi_errs.txt

On the ETS side, the file that defines all ETS fault messages for the MMI is **‘mmi/mmi_errs.txt’.**

6.4.1.1.1 Declare the New Fault Code

Open the file **‘mmi/mmi_errs.txt’**. Go to the 'CP Mmi Err Codes' enum and add the new fault code.

Example:

```
Test Error, [13]
```

Again, the fault code in the square brackets is optional.

6.4.1.1.2 Add New Fault Message Body

Below the 'CP Mmi Err Codes' enum add the fault message body.

Example:

```
Begin Message, Mmi Vrs Vocabulary Empty Rsp
   Code 2,     uint32
End
```

6.4.1.2 Modify CBP CP Files

In the CBP CP software, adding an ETS fault message involves adding the new fault code to the task’s error code enum and adding calls to MonFault() to pass the fault to ETS.

6.4.1.2.1 Add New Fault Code to <mmi/mmi_errs.h>

Add the new fault code to the ‘MmiErrsT’ enum in mmi_errs.h

Example:

```
MMI_ERR_TEST,  /* 13 */
```

6.4.1.2.2 Add 'MonFault () Calls to Source Code

In the CBP CP source files, a 'MonFault()' call can be added to send fault information to ETS whenever such information is desired.

The prototype for 'MonFault' is defined as follows:

```
extern void MonFault(MonFaultUnitT UnitNum, uint32 FaultCode1,
                     uint32 FaultCode2, MonFaultTypeT FaultType);
```

and MonFaultTypeT is defined as follows:

```
typedef enum
{
   MON_CONTINUE  = 0x00,   /* non-catastrophic */
   MON_HALT      = 0x01    /* catastrophic */
} MonFaultTypeT;
```

Example:

```
MonFault (MON_MMI_FAULT_UNIT, MMI_ERR_TEST, 0, MON_CONTINUE);
```

6.4.2 Results of ETS Faults

The fault codes from all Faults will be displayed in the ETS Main Fault log window. Meanwhile, if MonFault() is called with FaultType equal to MON_CONTINUE, the CP continues to operate as normal. If MonFault() is called with FaultType equal to MON_ HALT, the CP will go into a ‘while(TRUE)’ loop and stops all processing.

Note: Whenever any ETS *.txt file is modified, the ETS application has to be restarted for the changes to take effect.

7 Rev 27_D Review Information

Topic/Section(s) Reviewed: Sections 5.1.1 and 5.6; reviewed via email
Date: Sept 14,2005
Related Documents:
Meeting Moderator: N/A
Meeting Scribe: Darren Orloff

Invitees	Attended (Y/N)	Invitees	Attended (Y/N)
Roy Davis			
Bryan Yang			

Is approval required from anyone who didn't attend?
If so, from whom?

Agenda/Issues discussed:

1.
2.
3.
4.

Outcome/Summary: .

Action items

ID	Item	Owner	Date Due	Status	Closed?
1	Cells will expand to size of text				
2					
3					
4					
5					
6					
7					
8					

Were these action items transferred to the CRDB?
If so, who will update the CRDB? **If so, when were items transferred?**

8 Rev 25_D Review Information

Topic/Section(s) Reviewed: Sections 3 and 4.6; reviewed via email
Date: May 3, 2005
Related Documents:
Meeting Moderator: Darren Orloff
Meeting Scribe: Darren Orloff

Invitees	Attended (Y/N)	Invitees	Attended (Y/N)
Chuck Lewis		Laurent Ronc	
Hai Nguyen		Andy Monteiro	
Mike Shaver			
Ray Smith			

Is approval required from anyone who didn't attend?
If so, from whom?

Agenda/Issues discussed:

1.
2.
3.
4.

Outcome/Summary: No comments. Document approved as is.

Action items

ID	Item	Owner	Date Due	Status	Closed?
1	Cells will expand to size of text				
2					
3					
4					
5					
6					
7					
8					

Were these action items transferred to the CRDB?
If so, who will update the CRDB? **If so, when were items transferred?**

9 Ver 23_P Design Review Information

Topic of Review: ETS Overview document **Date**: 01/07/05
Related Documents:
Meeting Moderator: Charles Lewis
Meeting Scribe: Charles Lewis

Invitees	Attended (Y/N)	Invitees	Attended (Y/N)
John Miller	Y	Ray Smith	N
Mike Shaver	Y	Laurent Ronc	N
Henry Hsin	N	Chuck Lewis	Y
Amy Poon	N	Hai Nguyen	Y
Tiffany Lam	N	Maggie Liu	Y
Darren Orloff	Y	Andy Monteiro	Y

Is approval required from anyone who didn't attend? No
If so, from whom?

Agenda/Issues discussed:

1.

Outcome/Summary:

Action items

ID	Item	Owner	Date Due	Status	Closed?
1	Remove Win95 and NT, add Win98	Darren	1/7/05	Completed	Y
2	Remove Greenleaf reference	Darren	1/7/05	Completed	Y
3	Add 5.1.14 Auto Config Loading	Darren	1/7/05	Completed	Y
4					

Were these action items transferred to the CRDB? No
If so, who will update the CRDB?
If so, when were items transferred?

10 Ver 14_D Design Review Information

Topic of Review: ETS Overview document **Date**: 5/22/2003
Related Documents:
Meeting Moderator: Charles Lewis
Meeting Scribe: Charles Lewis

Invitees	Attended (Y/N)	Invitees	Attended (Y/N)
Henry Hsin	Y	Ray Smith	N
Sean Oleary	Y	Laurent Ronc	N
Mike Shaver	Y	Amy Poon	N

Is approval required from anyone who didn't attend? No
If so, from whom?

Agenda/Issues discussed:

1. Formatting of tables
2. Content of document

Outcome/Summary:

Action items

ID	Item	Owner	Date Due	Status	Closed?
1	Added descriptions of the following new options. Disabled commands, hidden fields, read only fields and startup command options.	Lewis	5/28/03	Completed	Yes
2	Updated with changes as recommended in review. Added Fault Log section description.	Lewis	5/28/03	Completed	Yes
3	Moved section 5 to the end.	Lewis	5/28/03	Completed	Yes
4					

Were these action items transferred to the CRDB? No
If so, who will update the CRDB?
If so, when were items transferred?

Page: 64
[l1]Is it really the name of the Message Definition file or the name of the definition.

Page: 64
[l2]This is a record definition