MCS®-96 Object Module Format

Procedural Interface

External Product Specification

27 April 1990

Version 3.1

DISCLAIMER

Intel makes no representation or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Intel reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Intel to notify any person of such revision or changes. The publication of this specification should not be construed as a commitment on Intel's part to implement any product.

TABLE OF CONTENTS

The Document Control Center Number for this MCS-96 OMF Procedural Interface EPS is 483419.

CHAPTER 1

PREFACE

1.1. Abstract

The MCS-96 OMF Procedural Interface (abbreviated OMF96P) is an object library which provides a convenient way of creating and reading Object Module Format files for the MCS-96 family [1]. OMF96P will be used by translators (ASM96, PL/M-96), linkers/locators (RL96) and loaders. OMF96P main features are:

Automatic "bookkeeping"

This feature includes:

- Automatic record length and checksum computation.
- Automatic record building/splitting.

In other words - the user needs only to specify relevant data, and not to be concerned with structural information.

OMF96 internal structure independence

The user sends and accepts data through procedures. The user needs to know how to interact with these procedures but does not need to know the internal representation of items in the OMF96 file. Thus, changes in the OMF96 file format can be implemented with a minor effect on the user.

OMF96 consistency checks

The interface will make some general checks about the consistency of the object file. *E.g*. - valid checksums, valid record types, etc. The content of the records (i.e., the data beyond the Record Length and before the checksum), will not be checked for consistency; OMF96P will assume that the translator output need not be rechecked again and again.

OMF96P is **not** supposed to check its "host" program, *i.e.*, OMF96P assumes that the "host" program calls it with the legal information in the required order.

1.2. Use Environment

OMF96P uses the UDI.C subsystem (which is based on the IOBUFF [4] library routines and the UDI [2] system calls) for its input/output operations. The UDI.C subsystem in turn requires an operating system environment which supports the standard C-language I/O services (stdio), such as DOS, System V UNIX, or HP-UX.

The OMF96P library will be supplied in a SMALL version (compiled under and used for the SMALL model of segmentation) and in a COMPACT version compiled under and used for the COMPACT model of segmentation).

The SMALL version will be called OMF96S.LIB; the COMPACT version will be called OMF96C.LIB.

The term OMF96P in the rest of the document stands for either of these libraries - the SMALL and the COMPACT.

CHAPTER 2

USER INTERFACE

2.1. Introduction

This section describes assumptions, conventions and notation used throughout this document.

The reader is assumed to know the terminology associated with OMF96 (as described in [1]). However, the reader need not be familiar with the internals of the OMF96 files.

The term "user" always applies to the programmer of the "host" program (i.e. the program which calls OMF96P procedures).

The term "any" when it comes within a procedure/variable name, stands for any possible string which generates a defined name. *E.g.* O96_Open_any stands for O96_Open_In and for O96_Open_Out.

2.2. Overview

The OMF96P is divided into three classes of procedures:

1) The OUT family, which creates OMF96 files, used mostly by translators but also by linkers.
2) The IN family, which reads and **checks** OMF96 files, used by linkers and loaders.
3) Common procedures. This class consists currently of only theO96_Set_Error_Handler.

The procedures are described in details later. A list of all procedures is given in Appendix A.

In addition to the OMF96P library, a header file is supplied, called O96.H. This file contains declarations for the public OMF96 functions, in ANSI prototype format. The file also contains declarations for public variables, definitions for the structures returned by O96_Get_Info, and #defines for common constants (such as Seg_Id values, error codes, etc.). O96.H is given in Appendix C.

Publics and constants in OMF96P are prefixed by O96.

The user is encouraged not to generate public symbols with the O96 prefix in the program in order to avoid errors.

The description of procedures consists of the procedure declaration, using PLM syntax, followed by a short discussion. There is only one deviation: the type of the formal parameters is given inside the procedure heading (Pascal-style) rather than specifying it in a separate declaration. PLM syntax is used also for the definition of the data structures.

Description of parameters is usually skipped if the parameter stands for OMF96 self explanatory element (such as Seg_Type, Seg_Id, etc.).

The procedures in each class are divided according to subjects.

2.3. Common Elements

2.3.1. Input/Output

OMF96P uses IOBUFF [4] for its input and output operations. At any given time, no more than one input and one output OMF96 file are handled by OMF96P.

If the user program consists of overlays, and the user needs one of the above files to be opened for more than one overlay life time, then the appropriate O96_Open_any procedure must reside in the root (this ensures that every procedure that must be in the root will be there).

OMF96P uses two fixed size internal buffers for its operation, one for input and one for output. Defining the buffers internally, will ensure compatibility between the size of the records created by the translator and the size of the records expected by the linker. The size of each buffer will be about 1K bytes.

2.3.2. Parameter Types

STRING

A string is a sequence of bytes preceded by a length byte:

```
DCL STRING IS '(1) BYTE';
```

The length specifies how many bytes are in the rest of the string. Any variable which contains NAME or STRING as part of its name is built in such manner. The maximum string length is as follows:

1) Name (module-, public-, external-, local-symbol): 40 characters.
2) Date_Time: 64 characters.

Excessive length will be reported by OMF96P as an error.

File Position

File position is represented as a single DWORD variable.

2.3.3. OMF96P Input File Status

The status of OMF96 input file can be determined by the values of the following public variables:

1) O96_Rec_Type_In (byte) - the record type for the last read **input** item. Valid only after calling O96_Get_Info.
2) O96_Next_Type_In (byte) - the type byte of the next **input** item to be read. Always valid (except after processing end-of-file record).
3) O96_Rec_Num_In (word) - the sequential number of the current **input** record. Record counting starts from one. Valid only after calling O96_Get_Info.
4) O96_Position_In (DWORD) - The file position at the beginning of the current record. Valid only after calling O96_Get_Info.
5) O96_Next_Position_In (DWORD) - If the current record is fully processed, (*i.e.* O96_Get_Info was called for the last item) then this is the file position at the beginning of the next record. Else it is equivalent to O96_Position_In.

O96_Next_Type_In is always defined (provided O96_Open_In was called).
O96_Rec_Type_In, O96_Rec_Num and O96_Position_In are valid only after a call to O96_Get_Info, and until the next call to O96_any.

CHAPTER 3

OUT PROCEDURES

3.1. Overview

The following is common for all OUT procedures:

1) Named O96_any_Out or O96_Put_Any.
2) Accepts information via parameters.
3) Untyped procedures.

If one of the above is not true for a procedure, it is explicitly stated.

Some OMF96 records consist of repetitions of items (e.g. segment definition record which may contain several segment definitions). The OUT procedures which refer to such records, group successive items of the same type into one record as long as the buffer can contain the record. The user is encouraged to call the same procedure sequentially as many times as possible in order to reduce the size of the output OMF96 file.

The OUT services provide one public variable, O96_Position_Out:

```
DCL O96_Position_Out DWORD;
```

O96_Position_Out may be used by the MCS-96 library manager. As such, it should be used before a call is made to O96_Put_Mod_Hdr. It gives the current file position of the OUT file. This DWORD, see File Position above, can be later used as an input to the O96_Seek_In. **(Not implemented)**

3.2. File Handling

```
O96_Open_Out: PROC
(Fname_P PTR) PTR;
```

O96_Open_Out opens the specified OMF96 file for writing. Fname_P is a pointer to a counted string containing the file name. O96_Open_Out returns a pointer to a FILE structure (as defined in stdio.h) for the opened file.

```
O96_Append_Out: PROC
(Fname_P PTR) PTR;
```

O96_Append_Out operates similarly to O96_Open_Out. The only difference is that a seek is performed to the eofile before returning to the user. Thus this service can be used when a new overlay is loaded where the previous overlay has closed the file (via O96_Close_Out). **(Not implemented)**

```
O96_Close_Out: PROC;
```

O96_Close_Out writes the current content of the file buffer to the OMF96 file and closes the OMF96 file.

3.3. Module Records

Includes two self explanatory procedures, each generates its respective record:

```
O96_Put_Mod_Hdr: PROC
  (Mod_Name_P PTR,
   Trn_Id BYTE,
   Date_Time_P PTR);

O96_Put_Mod_End: PROC
  (Module_Type BYTE,
   Validity BYTE);
```

3.4. Type Definition Records

The user calls the appropriate O96_Put_any_Leaf procedure for each single leaf in a definition of a type branch.

```
O96_Put_Numeric_Leaf: PROC
    (Value DWORD,
     Nice BOOL);

O96_Put_Index_Leaf: PROC
    (Type_Index WORD,
     Nice BOOL);

O96_Put_Nil_Leaf: PROC
      (Nice BOOL);

O96_Put_String_Leaf: PROC
    (String_P PTR,
     Nice BOOL);

O96_Put_Repeat_Leaf: PROC
      (Nice BOOL);

O96_Put_End_Leaf: PROC WORD;
```

Generating a definition of a type branch is done using the above routines. Each call generates one leaf. The O96_Put_End_Leaf is a function that returns the type index of the branch that has been terminated. The first type index to be returned is 32, due to the predefined type indices.

3.5. Definition Records

The user calls the appropriate O96_Put_any_Def procedure for each single definition.

```
O96_Put_Abs_Seg_Def: PROC
    (Seg_Type BYTE,
     Base_Address DWORD,
     Seg_Size DWORD);

O96_Put_Rel_Seg_Def: PROC
    (Seg_Type BYTE,
     Align_Type BYTE,
     Seg_Size DWORD);
```

Generating segment definitions is done via the above procedures. Each call generates one definition. The separation into two procedures is done because of the different parameters involved.

```
  O96_Put_Pub_Def: PROC
    (Seg_Id BYTE,
     Offset WORD,
     Name_P PTR,
     Type_Index WORD);

  O96_Put_Ext_Def: PROC
    (Seg_Id BYTE,
     Name_P PTR,
     Type_Index WORD);
```

Generating public/external definitions is done via the above procedures. Each call generates one definition.

3.6. Debug Records

```
O96_Put_Mod_Ancestor: PROC
    (Name_P PTR);

 O96_Put_Dbg_Seg_Def: PROC
    (Seg_Type BYTE,
     Base_Address DWORD,
     Seg_Size DWORD);
```

O96_Put_Mod_Ancestor is called by RL96 for each translator-generated module header record or module ancestor record scanned at the input. An O96_Put_Mod_Ancestor call must be followed by zero to seven calls to O96_Put_Dbg_Seg_Def, with each call defining another segment. O96_Put_Dbg_Seg_Def may not be called in any other context.

```
O96_Put_Block_Def: PROC
   (Name_P PTR,
    Seg_Id BYTE,
    Offset DWORD,
    Size DWORD,
    Type_Index WORD);
```

O96_Put_Block_Def generates a definition of one DO-block for each call. When the length of the name is zero, the 'Type_Index' parameter is ignored.

```
O96_Put_Proc_Def: PROC
   (Name_P PTR,
    Seg_Id BYTE,
    Offset DWORD,
    Size DWORD,
    Type_Index WORD,
    Scope BYTE,
    Frame_Id WORD,
    Frame_Offset DWORD,
    Return_Offset WORD,
    Prologue_Size BYTE);
```

O96_Put_Proc_Def generates a definition of one procedure block for each call. Frame_Id is either the Seg_Id or Ext_Id of the frame register, as specified by the scope parameter.

```
O96_Put_Block_End: PROC;
```

O96_Put_Block_End generates one block-end record for each call.

```
O96_Put_Loc_Sym: PROC
  (Seg_Id BYTE,
   Offset DWORD,
   Name_P PTR,
   Type_Index WORD);
```

O96_Put_Loc_Sym generates one local symbol definition for each call.

```
O96_Put_Lin_Num: PROC
  (Seg_Id BYTE,
   Offset DWORD,
   Number WORD);
```

O96_Put_Lin_Num generates one line number definition for each call.

3.7. Content And Fixup Records

Throughout this document, the word "Cont" is used (usually in names) as a shorthand for "content".

A user can specify either a pure content block (or byte, word, 3-byte, or dword field) or a fixed-up item. A fixed-up item contains the 1-to-4 byte field to be fixed and the type of fixup to perform.

OMF96P will accumulate all the data bytes in one buffer, the fixups in another buffer, and will decide what will be the actual form of the fixups: whether to put the reference offset in the fixup unit or in the fixed-up field. One rule must be observed: the object code between calls to O96_Put_Cont_Hdr must be sent in the order of its placement in memory. *E.g.* if the fourth and the sixteenth bytes, after an O96_Put_Cont_Hdr call, are to be fixed-up then the sequence of calls should be:

Send first three bytes (e.g. via O96_Put_Cont_Str).
Fixup a byte item (via O96_Put_Fix_Cont).
Send next 11 bytes.
Fixup a byte item.
Send the rest of the bytes.

In other words, there is no way to fixup a data which has already been sent to the buffer.

The procedures provided for generation of content and fixup records are:

```
O96_Put_Cont_Hdr: PROC
   (Seg_Id BYTE,
    Offset DWORD);
```

O96_Put_Cont_Hdr initiates a **new** content record and closes the previous content and fixup records.

The following three procedures fill content records:

```
O96_Put_Cont_Str: PROC
   (Cont_Ptr PTR,
    Cont_Length WORD);
```

O96_Put_Cont_Str appends a string of bytes of a given (unlimited) length to the content record. For four frequent cases, the following procedures are cheaper and more convenient:

```
O96_Put_Cont_Byte: PROC
   (Cont_Byte BYTE);
```

O96_Put_Cont_BYTE appends one byte to the current content data.

```
O96_Put_Cont_Word: PROC
   (Cont_Word WORD);
```

O96_Put_Cont_Word appends one word to the current content data.

```
O96_Put_Cont_24Bits: PROC
  (Cont_24Bits DWORD);
```

O96_Put_Cont_24Bits appends the least significant 3 bytes of Cont_24Bits to the current content data.

```
O96_Put_Cont_Dword: PROC
  (Cont_Dword DWORD);
```

O96_Put_Cont_Dword appends one dword to the current content data.

```
O96_Put_Fix_Cont: PROC
  (Fixup_Type BYTE,
   Id WORD,
   Ref_Offset DWORD);
```

O96_Put_Fix_Cont defines an object item which has to be fixed up by RL96.

Fixup_Type has the same format as the respective field in the OMF specification; with the Scope, Ref_Type and Align_Type. The only difference is that Ref_Offset_Flag, the bit which specifies whether Ref_Offset should be placed in the fixup unit or in the fixed-up field, is not specified here. Rather, this bit is determined by OMF96P by considering Fixup_Type and Ref_Offset.

Id is interpreted as either the Seg_Id (in the low order byte of the word), or the Ext_Id (a full word), depending on the Scope field in Fixup_Type.

Ref_Offset has the same meaning as in the OMF spec. This is usually the Value field of the evaluated relocatable/external expression.

3.8. Library Control Records

```
O96_Put_Lib_Hdr: PROC
  (Mod_Count WORD,
   File_Position DWORD);

O96_Put_Lib_Mod_Name: PROC
    (Mod_Name_P PTR);

O96_Put_Lib_Mod_Loc: PROC
  (File_Position DWORD);

O96_Put_Lib_Pub_Name: PROC
    (Pub_Name_P PTR);
```

O96_Put_Lib_Hdr creates a Library Header record. The rest create one item of the corresponding library control records (O96_Put_Lib_Pub_Name relates to the Library Dictionary record). **(Not implemented)**

CHAPTER 4

IN PROCEDURES

4.1. Overview

The IN family is divided into two classes:

1) Buffer and record handling (e.g. skip to next record).
2) Transfer of information (e.g. content of segment definition). This transfer is accomplished by a single procedure, O96_Get_Info, whose result is placed in a user supplied buffer.

The IN procedures and variables are named either O96_any_In or O96_Get_Any.

The public variables O96_Rec_Type_In, O96_Next_Type_In, O96_Rec_Num_In, O96_Position_In, and O96_Next_Position_In (see 2.2.3) contain the status about the current and the next input record type, number and position.

4.2. File And Buffer Handling

4.2.1. O96_Open_In

```
O96_Open_In: PROC
(In_File_Ptr PTR,
 Fname_P PTR) PTR;
```

O96_Open_In opens the specified OMF96 file for input, initializes the internal buffer for the following O96_Get_Info calls, and returns a pointer to this buffer. The buffer will contain the item received by each O96_Get_Info.

Fname_P is a pointer to a counted string specifying the name of the file to be opened.
In_File_Ptr is a pointer to a pointer which will receive the address of the FILE structure (as specified in stdio.h) for the opened file.

4.2.2. O96_Close_In

```
O96_Close_In: PROC;
```

O96_Close_In closes the OMF96 input file.

4.2.3. O96_Get_Next_Rec

```
O96_Get_Next_Rec: PROC;
```

Skip the current OMF96 input record. All subsequent calls will refer to the next record. This is the way in which the linker can skip all debug records etc.

The type of the new record resides in the O96_Next_Type_In variable and can be examined.

4.2.4. O96_Get_Next_Mod

```
O96_Get_Next_Mod: PROC;
```

Skip the current OMF96 module. All subsequent calls will refer to the next module in the file (if it exists). This is the way in which the linker can skip an entire module.

4.2.5. O96_Seek_In

```
O96_Seek_In: PROC
(File_Position DWORD,
 New_Rec_Type BYTE);
```

The procedure seeks the current file to the position specified by File_Position. The position should point to the beginning of a record whose type is given by New_Rec_Type.

4.3. O96_Get_Info

```
O96_Get_Info: PROC;
```

O96_Get_Info is a parameter-less procedure which gets the next item from the input file and puts it in the internal buffer whose address is returned by the O96_Open_In call. When this procedure ends, the public variable O96_Next_Type_In is set to the type of the next item.

The item is placed in the buffer, decomposed into its fields. *E.g.* a public symbol definition will be decomposed into a Seg_Id, Offset, Name and Type index. The definition of the structures for each of the possible items is given in the following sections.

The meaning of each field, its range of values, etc., are defined in the OMF [1].

Note that since O96_Next_Type_In is changed by O96_Get_Info, the actual structure in the buffer to be used **after** calling O96_Get_Info, is O96_Rec_Type_In.

The special notations used in the following sections are:

STR1 and STR2

In the following declarations, STR1 and STR2 stand for the following string definitions (the first byte in each string is the length byte):

```
DCL
   STR1 IS '(41) BYTE',
   STR2 IS '(65) BYTE';
```

ITEM

Another literal used is ITEM. It is used to define structure received by O96_Get_Info calls, and is defined as follows:

```
DCL
   ITEM IS 'BASED Buff_Start STRUCTURE';
```

Where Buff_Start is the pointer returned by O96_Open_In.

O96_Rec_Type_In

The value of the current record type, which determines the kind of the structure to be used, is given as a comment to the right of the ITEM declaration.

4.3.1. Module Records

```
/* O96_Rec_Type_In = O96_Mod_Hdr */
        DCL Mod_Hdr Item
            (Mod_Name Str1,
             Trn_Id Byte,
             Time_Date Str2);

/* O96_Rec_Type_In = O96_Mod_End */
        DCL Mod_End Item
            (Mod_Type Byte,
             Validity Byte);
```

No Structure Is Provided For The EOF Record (O96_Rec_Type_In = O96_Eof).

4.3.2. Definition Records

```
/* O96_Rec_Type_In = O96_Seg_Def */
        DCL Seg_Def ITEM
          (Seg_Id BYTE, /* the MSB of Seg_Id is the
              Relocatability bit */
           Base_Address_Or_Align_Type DWORD, /*
              depends on the Relocatability bit */
           Size DWORD);
```

The following structure is used for Pub_Def, Ext_def and Loc_Sym_Def. However, since the external name has no offset attribute, the Offset field in Sym_Def is ignored here.

```
/* O96_Rec_Type_In = O96_Pub_Def, O96_Ext_Def or O96_Loc_Sym_Def */
                    DCL Sym_Def ITEM
                        (Seg_Id BYTE,
                         Offset DWORD,
```

```
                    Sym_Name STR1,
                    Type_Index WORD);
```

4.3.3. Object Code Records

The structure for content record contains also a place for the possible following fixups (the content record must be valid when the fixups arrive).

```
/* O96_Rec_Type_In = O96_Content or O96_Fixup */
          DCL Content ITEM
          (Seg_Id BYTE,
           Offset DWORD,
           Size DWORD,
           Data_Part (MAX_DATA_SIZE) BYTE,
           Fixup_Type BYTE, /* MSB contains the
              Scope bit */
           PCR_Offset WORD,
           Ext_Or_Seg_Id WORD,
           Ref_Offset DWORD /* unused if
              Ref_Offset_Flag in Fixup_Type is
              IN_PCR */);
```

O96_Get_Info will place the content information in the Seg_Id, Offset, Size and the Data_Part fields. Each of the following fixups will be placed in the Fixup_Type, PCR_Offset, EXT_Or_Seg_Id and the Ref_Offset fields. MAX_DATA_SIZE is currently defined as 1024.

4.3.4. Debug Records

```
/* O96_Rec_Type_In = O96_Mod_Ancestor */
        DCL Mod_Ancestor ITEM
           (Mod_Name STR1,
            Seg_Count BYTE,
            Abs_Seg_Def(63) BYTE);
```

Mod_Name is the module name.

Seg_Count is the number (up to seven) of absolute segment definitions in Abs_Seg_Def.

Abs_Seg_Def (specified above as a simple byte array because of PLM limitations) is an array of absolute segment definitions, corresponding to the located relocatable segments of the module Mod_Name. The Abs_Seg_Def entries have the Seg_Def structure, described previously.

The structure used for local symbol definition is the same as the one used for public and external symbols, Sym_Def (see above).

```
/* O96_Rec_Type_In = O96_Lin_Num */
        DCL Lin_Num ITEM
```

```
                (Seg_Id BYTE,
                 Offset DWORD,
                 Number WORD);

          /* O96_Rec_Type_In = O96_Block_Def */
                  DCL Block_Def ITEM
                    (Block_Name  STR1,
                     Seg_Id BYTE,
                     Offset DWORD,
                     Size DWORD,
                     Block_Flags BYTE,
                     Type_Index WORD,
                     Frame_Ext_or_Seg_Id WORD,
                     Frame_Offset DWORD,
                     Ret_Offset WORD,
                     Prologue_Size BYTE );
```

No structure is provided for the block end record (O96_Rec_Type_In = O96_Block_End).

4.3.5. Type Definition Records

```
          /* O96_Rec_Type_In = O96_Type_Def */
                  DCL Leaf_Def ITEM
                    (Kind BYTE,
                     Lngth BYTE, /* the number of content
                        bytes */
                     Cntnt (1) BYTE );
```

4.3.6. Library Control Records

```
           /* O96_Rec_Type_In = O96_Lib_Hdr */
                   DCL Lib_Hdr ITEM
                     (Mod_Count WORD,
                      Offset DWORD /* File offset */);
```

The items from the three library tail record are returned one for each call to O96_Get_Info.

```
          /* O96_Rec_Type_In = O96_Mod_Name */
                 DCL Lib_Mod_Name ITEM
                     (Mod_Name STR1);

           /* O96_Rec_Type_In = O96_Mod_Loc */
                  DCL Lib_Mod_Loc ITEM
                     (Offset DWORD /* 1'st word = low part,
                         2'nd = high part */);

        /* O96_Rec_Type_In = O96_Lib_Pub_Name */
                 DCL Lib_Pub_Name ITEM
                     (Pub_Name STR1);
```

Notes:

1) A Pub_Name of zero length signifies the end of the module public list.

CHAPTER 5

ERROR HANDLING

Before calling OMF96P services, the user should initialize the error handler procedure by calling O96_Set_Error_Handler. The last procedure is defined as follows:

```
O96_Set_Error_Handler: PROC
       (Proc_P PTR);
```

where Proc_P points to a user supplied error handler defined as follows:

```
User_Supplied_Error_Handler: PROC
   (Err_No  BYTE /* OMF96P error code (see APPENDIX
     B) */,
    Value DWORD /* meaning depends upon Err_No
     (usually the erroneous Value) */);
```

When an error is detected omf96p calls indirectly the user supplied procedure.

OMF96P can detect invalid OMF96 input file and I/O errors. Errors detected are categorized into the following classes:

1) Invalid string. If a field of an item is a string and the string is longer than allowed, the rest of the characters are ignored and an error is reported.
2) Invalid object file. This class includes invalid record type, checksum error, invalid sequence of records.
3) I/O errors. Since the IN family performs I/O via the IOBUFF package, IN file errors must be handled by the IOBUFF error handler [4].

The user can use the O96_Rec_Num_In and O96_Rec_Type_In public variables of OMF96P in order to create his error messages.

OMF96P supplies a standard error handler, O96_Std_Error_Handler, which may be attached via a O96_Set_Error_Handler call. This procedure prints the OMF96P errors in the following format:

```
OMF96 ERROR NUMBER: number.
   ERROR TEXT: error message.
   RECORD NUMBER: number.
   RECORD TYPE: number.
   PARAM: number.
```

After the message is printed the program is aborted.

Note - if O96_Set_Error_Handler is not called, all errors detected by OMF96P are ignored.

APPENDIX A

LIST OF PUBLIC VARIABLES AND PROCEDURES

The first section of this appendix contains the list of all the public procedures of OMF96P. The second section contains the definitions of the structure types used in the procedure definitions. The third section contains the list of the public variables provided by the IN and OUT services.

A.1 Procedures

```
O96_Open_Out: PROC
 (Fname_P PTR) PTR;

O96_Append_Out: PROC
 (Fname_P PTR) PTR;

O96_Close_Out: PROC;

O96_Put_Mod_Hdr: PROC
 (Mod_Name_P PTR,
  Trn_Id BYTE,
  Date_Time_P PTR);

O96_Put_Mod_End: PROC
 (Module_Type BYTE,
  Validity BYTE);

O96_Put_Eof: PROC;

O96_Put_Abs_Seg_Def: PROC
 (Seg_Type BYTE,
  Base_Address DWORD,
  Seg_Size DWORD);

O96_Put_Rel_Seg_Def: PROC
 (Seg_Type BYTE,
  Align_Type BYTE,
  Seg_Size DWORD);

O96_Put_Pub_Def: PROC
 (Seg_Id BYTE,
  Offset DWORD,
  Name_P PTR,
  Type_Index WORD);

O96_Put_Ext_Def: PROC
 (Seg_Id BYTE,
  Name_P PTR,
  Type_Index WORD);
```

```
O96_Put_Mod_Ancestor: PROC
 (Name_P PTR);

O96_Put_Dbg_Seg_Def: PROC
 (Seg_Type BYTE,
  Base_Address DWORD,
  Seg_Size DWORD);

O96_Put_Block_Def: PROC
 (Name_P PTR,
  Seg_Id BYTE,
  Offset DWORD,
  Size DWORD,
  Type_Index WORD);

O96_Put_Proc_Def: PROC
 (Name_P PTR,
  Seg_Id BYTE,
  Offset DWORD,
  Size DWORD,
  Type_Index WORD,
  Scope BYTE,
  Frame_Id WORD,
  Frame_Offset DWORD,
  Return_Offset,
  Prologue_Size BYTE);

O96_Put_Block_End: PROC;

O96_Put_Loc_Sym: PROC
 (Seg_Id BYTE,
  Offset DWORD,
  Name_P PTR,
  Type_Index WORD);

O96_Put_Lin_Num: PROC
 (Seg_Id BYTE,
  Offset DWORD,
  Number WORD);

O96_Put_Cont_Hdr: PROC
 (Seg_Id BYTE,
  Offset DWORD);

O96_Put_Cont_Str: PROC
 (Cont_Ptr PTR,
  Cont_Length WORD);

O96_Put_Cont_Byte: PROC
```

```
 (Cont_Byte BYTE);

O96_Put_Cont_Word: PROC
 (Cont_Word WORD);

O96_Put_Cont_24Bits: PROC
 (Cont_24Bits DWORD);

O96_Put_Cont_Dword: PROC
 (Cont_Dword DWORD);

O96_Put_Fix_Cont: PROC
 (Fixup_Type BYTE,
  Id WORD,
  Ref_Offset DWORD);

O96_Put_Lib_Hdr: PROC
 (Mod_Count WORD,
  File_Position DWORD);

O96_Put_Lib_Mod_Name: PROC
 (Mod_Name_P PTR);

O96_Put_Lib_Mod_Loc: PROC
 (File_Position DWORD);

O96_Put_Lib_Pub_Name: PROC
 (Pub_Name_P PTR);

O96_Open_In: PROC
 (In_File_Ptr PTR,
  Fname_P PTR) PTR;

O96_Close_In: PROC;

O96_Get_Next_Rec: PROC;

O96_Get_Next_Mod: PROC;

O96_Seek_In: PROC
 (File_Position DWORD,
  New_Rec_Type BYTE);

O96_Get_Info: PROC;

O96_Set_Error_Handler: PROC
 (Proc_A ADDRESS);

O96_Std_Error_Handler: PROC
```

```
 (Err_No BYTE,
  Param DWORD);
```

A.2 Structure Types

```
/* O96_Rec_Type_In = O96_Mod_Hdr */
DCL Mod_Hdr ITEM
 (Mod_Name STR1,
  Trn_Id BYTE,
  Time_Date STR2);

/* O96_Rec_Type_In = O96_Mod_End */
DCL Mod_End ITEM
 (Mod_Type BYTE,
  Validity BYTE);

/* O96_Rec_Type_In = O96_Seg_Def */
DCL Seg_Def ITEM
 (Seg_Id BYTE /* the MSB of Seg_Id is the Relocatability bit */,
  Base_Address_Or_Align_Type DWORD /* depends on the
   Relocatability bit */,
  Size DWORD);

/* O96_Rec_Type_In = O96_Pub_Def, O96_Ext_Def or O96_Loc_Sym_Def
*/
DCL Sym_Def ITEM
 (Seg_Id BYTE,
  Offset DWORD,
  Sym_Name STR1,
  Type_Index WORD);

/* O96_Rec_Type_In = O96_Content or O96_Fixup */
DCL Content ITEM
 (Seg_Id BYTE,
  Offset DWORD,
  Size DWORD,
  Data_Part (1024) BYTE,
  Fixup_Type BYTE /* MSB contains the Scope bit */,
  PCR_Offset WORD,
  Ext_Or_Seg_Id WORD,
  Ref_Offset DWORD /* unused if Ref_Offset_Flag in
  Fixup_Type is IN_PCR */);

/* O96_Rec_Type_In = O96_Mod_Ancestor */
DCL Mod_Ancestor ITEM
 (Mod_Name STR1,
  Seg_Count BYTE,
  Abs_Seg_Def(63) BYTE);
```

```
/* O96_Rec_Type_In = O96_Lin_Num */
DCL Lin_Num ITEM
 (Seg_Id BYTE,
  Offset DWORD,
  Number WORD);

/* O96_Rec_Type_In = O96_Lin_Num */
DCL Block_Def ITEM
 (Block_Name STR1,
  Seg_Id BYTE,
  Offset DWORD,
  Size DWORD,
  Block_Flags BYTE,
  Type_Index WORD,
  Frame_Ext_or_Seg_Id WORD,
  Frame_Offset DWORD,
  Ret_Offset WORD,
  Prologue_Size BYTE);

/* O96_Rec_Type_In = O96_Type_Def */
DCL Leaf_Def ITEM
 (Kind BYTE,
  Lngth BYTE, /* the number of content bytes */
  Cntnt (1) BYTE);

/* O96_Rec_Type_In = O96_Lib_Hdr */
DCL Lib_Hdr ITEM
 (Mod_Count WORD,
  Offset DWORD);

/* O96_Rec_Type_In = O96_Mod_Name */
DCL Lib_Mod_Name ITEM
 (Mod_Name STR1);

/* O96_Rec_Type_In = O96_Mod_Loc */
DCL Lib_Mod_Loc ITEM
 (Offset DWORD);

/* O96_Rec_Type_In = O96_Lib_Pub_Name */
DCL Lib_Pub_Name ITEM
 (Pub_Name STR1);
```

A.3 Scalars

```
DCL O96_Rec_Type_In BYTE;

DCL O96_Next_Type_In BYTE;

DCL O96_Rec_Num_In WORD;
```

```
DCL O96_Position_In DWORD;

DCL O96_Next_Position_In DWORD;

DCL O96_Position_Out DWORD;
```

APPENDIX B

ERROR MESSAGES

Errors are numbered from 1 and above.

List of all errors follows. The value associated with the error is also described.

01 - CONTENT RECORD TOO LONG.
Input record length.

02 - Reserved.

03 - CHECKSUM ERROR.
The record checksum value.

04 - ILLEGAL SEQUENCE OF INPUT RECORDS.
The next record type.

05 - ILLEGAL RECORD TYPE.
The record type.

06 - ITEM TOO LONG.
Record size.

07 - PREMATURE END OF FILE.
None.

08 - STRING TOO LONG.
String length.

09 - Reserved.

10 - INVALID TRANSLATOR ID.
Erroneous field.

11 - INVALID SEGMENT TYPE.
Erroneous field.

12 - INVALID SYMBOL TYPE.
Erroneous field.

13 - INVALID ALIGN TYPE.
Erroneous field.

14 - INVALID REFERENCE TYPE.

Erroneous field.

15 - INVALID EXTERNAL ID.
Erroneous field.

16 - INVALID PCR OFFSET.
Erroneous field.

17 - INVALID TYPE DEFINITION.
Erroneous field.

18 - INVALID SEGMENT SIZE
Segment size.

19 - INVALID SEGMENT OFFSET
Segment offset.

APPENDIX C

O96.H

This appendix lists the contents of the OMF96P header file, O96.H.

```
/* OMF96P - Constants */

#define MAX_CONT_DATA_SIZE  1024
#define MAX_NAME_LEN        40
#define MAX_DATE_TIME_LEN   64

/* OMF96P - Type definitions */

typedef unsigned char BYTE;
typedef unsigned short WORD;
typedef unsigned long DWORD;
typedef char *PTR;

typedef BYTE STR1[MAX_NAME_LEN+1];
typedef BYTE STR2[MAX_DATE_TIME_LEN+1];
typedef WORD CONNECTION;
typedef BYTE BOOL;

typedef void ERROR_HANDLER( BYTE Err_No, DWORD Param );


/* OMF96P - Structures returned by O96_Get_Info */

struct struc_Mod_Hdr
{
   STR1 Mod_Name;
   BYTE Trn_Id;
   STR2 Date_Time;
};

struct struc_Mod_End
{
   BYTE Mod_Type;
   BYTE Validity;
};

struct struc_Seg_Def
{
   BYTE Seg_Id;
   DWORD Base_Address_Or_Align_Type;
   DWORD Size;
};
```

```
struct struc_Sym_Def
{
   BYTE Seg_Id;
   DWORD Offset;
   STR1 Symbol_Name;
   WORD Type_Index;
};

struct struc_Content
{
   BYTE Seg_Id;
   DWORD Offset;
   DWORD Size;
   BYTE Data_Part[MAX_CONT_DATA_SIZE];
   BYTE Fixup_Type;
   WORD PCR_Offset;
   WORD Ext_Or_Seg_Id;
   DWORD Ref_Offset;
};

struct struc_Mod_Ancestor
{
   STR1 Mod_Name;
   BYTE Seg_Count;
   struct struc_Seg_Def Abs_Seg_Def[7];
};

struct struc_Lin_Num
{
   BYTE Seg_Id;
   DWORD Offset;
   WORD Number;
};

struct struc_Block_Def
{
   STR1 Block_Name;
   BYTE Seg_Id;
   DWORD Offset;
   DWORD Size;
   BYTE Block_Flags;
   WORD Type_Index;
   WORD Frame_Ext_Or_Seg_Id;
   DWORD Frame_Offset;
   WORD Ret_Offset;
   BYTE Prologue_Size;
};

struct struc_Leaf_Def
{
```

```
   BYTE Kind;
   BYTE Lngth;
   BYTE Cntnt[1];
};

struct struc_Lib_Hdr
{
   WORD Mod_Count;
   DWORD Offset;
};

struct struc_Lib_Mod_Name
{
   STR1 Mod_Name;
};

struct struc_Lib_Mod_Loc
{
   DWORD Offset;
};

struct struc_Lib_Pub_Name
{
   STR1 Pub_Name;
};


/* OMF96P - Error Codes */

#define O96_Rec_Too_Long         1
#define O96_Reserved_2           2
#define O96_Checksum_Error       3
#define O96_Ill_Sequence         4
#define O96_Ill_Rec_Type         5
#define O96_Item_Too_Long        6
#define O96_Premature_Eof        7
#define O96_String_Too_Long      8
#define O96_Reserved_9           9
#define O96_Invalid_Trn_Id      10
#define O96_Invalid_Seg_Type         11
#define O96_Invalid_Sym_Type         12
#define O96_Invalid_Align_Type       13
#define O96_Invalid_Ref_Type         14
#define O96_Invalid_Ext_Id           15
#define O96_Invalid_PCR_Offset       16
#define O96_Invalid_Type_Def         17
#define O96_Invalid_Seg_Size         18
#define O96_Invalid_Seg_Off          19

/* OMF96P - Record Types  */
```

```
#define O96_Mod_Hdr             0x02
#define O96_Mod_End             0x04
#define O96_Content             0x06
#define O96_Lin_Num             0x08
#define O96_Block_Def           0x0a
#define O96_Block_End           0x0c
#define O96_Eof                 0x0e
#define O96_Mod_Anc             0x10
#define O96_Loc_Sym             0x12
#define O96_Type_Def            0x14
#define O96_Pub_Def             0x16
#define O96_Ext_Def             0x18
#define O96_Res_Type_1A         0x1a
#define O96_Res_Type_1C         0x1c
#define O96_Res_Type_1E         0x1e
#define O96_Seg_Def             0x20
#define O96_Fixup               0x22
#define O96_Res_Type_24         0x24
#define O96_Lib_Mod_Loc         0x26
#define O96_Lib_Mod_Nam         0x28
#define O96_Lib_Dic             0x2a
#define O96_Res_Type_2C         0x2c
#define O96_Lib_Hdr             0x2e
#define O96_Res_Type_30         0x30
#define O96_Res_Type_32         0x32
#define O96_Max_Rec_Type        0x32

/* OMF96P - Translator Id */

#define O96_Trn_Id_Mask         0xe0
#define O96_Trn_ASM96           0x00
#define O96_Trn_PLM96           0x20
#define O96_Trn_C96             0x40
#define O96_Trn_Unspec          0xe0

/* OMF96P - OMF_Ver */

#define O96_OMF_Ver_Mask            0x1e
#define O96_Min_OMF_Ver             0x00
#define O96_OMF96_V1dot0_DOC_V1dot4 0x00
#define O96_OMF96_V2dot0_DOC_V2dot0 0x04
#define O96_OMF96_V3dot0_DOC_V3dot0 0x06
#define O96_Max_OMF_Ver             0x06

/* OMF96P - Generator */

#define O96_Generator_Mask          0x01
#define O96_Translator96            0x00
#define O96_RL96                    0x01
```

```
/* OMF96P - Relocatability types */

#define O96_Rel_Mask            0x80
#define O96_Abs                 0x00
#define O96_Rel                 0x80

/* OMF96P - Based attribute */

#define O96_Based_Mask          0x60
#define O96_Non_Based           0x00
#define O96_Based_16            0x40
#define O96_Based_24            0x60

/* OMF96P - Segment types */

#define O96_Seg_Type_Mask       0x1f
#define O96_Code                0x00
#define O96_Data                0x01
#define O96_Reg                 0x02
#define O96_Overlay             0x03
#define O96_Dynamic             0x04
#define O96_Stack               0x05
#define O96_No_Seg              0x06
#define O96_Constant            0x07
#define O96_Far_Code            0x08
#define O96_Far_Data            0x09
#define O96_High_Code           0x0a
#define O96_Far_Constant        0x0f
#define O96_Max_Seg_Type        0x0f
#define O96_Num_Of_Seg_Types    0x0c

/* OMF96P - Align Types */

#define O96_Align_Mask          0x03
#define O96_Byte_Align          0x00
#define O96_Word_Align          0x01
#define O96_Long_Align          0x02
#define O96_Max_Align_Type      0x02

/* OMF96P - Module Types */

#define O96_Other_Mod           0
#define O96_Main_Mod            1

/* OMF96P - Validity */

#define O96_Valid_Mod           0
#define O96_Err_Mod             1
```

```
/* OMF96P - Nice and Easy */

#define O96_Nice_Mask      0x80
#define O96_Easy           0x00
#define O96_Nice           0x80

/* OMF96P - Types of Leaves */

#define O96_Leaf_Mask               0x7f
#define O96_Max_One_Byte_Sgnint     99
#define O96_Nil_Leaf                100
#define O96_Two_Byte_SgnInt         101
#define O96_Four_Byte_SgnInt        102
#define O96_String_Leaf             103
#define O96_Index_Leaf              104
#define O96_Repeat_Leaf             105
#define O96_End_Of_Branch_Leaf      106
#define O96_Max_Leaf_Type           106

/* OMF96P - Predefined Values for Numeric Leaves */

#define O96_Numeric_Pointer     110
#define O96_Numeric_Bit         109
#define O96_Numeric_Enum        108
#define O96_Numeric_Union       107
#define O96_Numeric_Scalar       99
#define O96_Numeric_Real         98
#define O96_Numeric_Entry        97
#define O96_Numeric_UnsInt       96
#define O96_Numeric_SgnInt       95
#define O96_Numeric_Array        94
#define O96_Numeric_Structure    93
#define O96_Numeric_List         92
#define O96_Numeric_Procedure    91
#define O96_Numeric_Label        90
#define O96_Numeric_Whole        89

/* OMF96P - Predefined Symbol Types (= type indices) */

#define O96_Type_Mask           0x7f
#define O96_No_Type             0
#define O96_Byte_Type           1
#define O96_Word_Type           2
#define O96_Long_Type           3
#define O96_Entry_Type          4
#define O96_Shortint_Type       5
#define O96_Integer_Type        6
#define O96_Longint_Type        7
#define O96_Real_Type           8
#define O96_Hll_Byte_Type       9
```

```
#define O96_Hll_Word_Type        10
#define O96_Hll_Long_Type        11
#define O96_Label_Type           12
#define O96_Max_Predef_Type      31
#define O96_Max_Used_Predef_Type      12
#define O96_Num_Of_Used_Predef_Type   13

/* OMF96P - Scope */

#define O96_Scope_Mask                0x80
#define O96_Local_Scope               0x00
#define O96_Extrn_Scope               0x80

/* OMF96P - Block_Type */

#define O96_Block_Mask           0x40
#define O96_Do_Block             0x00
#define O96_Proc_Block           0x40
```

```
/* OMF96P - Reference types */

#define O96_Ref_Type_Mask       0x3c
#define O96_Reg_Simple          0x00
#define O96_Reg_Auto_Incr       0x04
#define O96_Sh_Count_Imm        0x08
#define O96_Sh_Count_Reg        0x0c
#define O96_DCB_Const           0x10
#define O96_Short_Direct        0x14
/* O96_Short_Direct is not used */
#define O96_Short_Jmp           0x18
#define O96_Medium_Jmp          0x1c
#define O96_Medium_Call         0x20
#define O96_Long_Jmp_Call       0x24
#define O96_Long_Direct         0x28
#define O96_Far_Jmp_Call        0x2c
#define O96_Far_Direct          0x30
#define O96_DCL_Const           0x34
#define O96_Max_Byte_Ref_Type   0x18
#define O96_Max_Word_Ref_Type   0x28
#define O96_Max_24Bit_Ref_Type  0x30
#define O96_Max_Ref_Type        0x34

/* OMF96P - Immediate offset flag */

#define O96_Imm_Offset_Mask     0x40
#define O96_Imm_Offset          0x00
#define O96_Offset_In_PCR       0x40

/* o96out.c - OMF96P OUT services */

extern DWORD O96_Position_Out;

FILE *O96_Open_OUT( char *Fname );
void O96_Close_OUT( void );
void O96_Put_Mod_Hdr( STR1 Mod_Name, BYTE Trn_Id, STR2 Date_Time );
void O96_Put_Mod_End( BYTE Module_Type, BYTE Validity );
void O96_Put_Abs_Seg_Def( BYTE Seg_Type, DWORD Base_Address, DWORD
Seg_Size );
void O96_Put_Rel_Seg_Def( BYTE Seg_Id, BYTE Align_Type,
  DWORD Seg_Size );
void O96_Put_Numeric_Leaf( DWORD Value, BOOL Nice );
void O96_Put_Index_Leaf( WORD Type_Index, BOOL Nice );
void O96_Put_Nil_Leaf( BOOL Nice );
void O96_Put_String_Leaf( STR1 String, BOOL Nice );
void O96_Put_Repeat_Leaf( BOOL Nice );
WORD O96_Put_End_Leaf( void );
void O96_Put_Pub_Def( BYTE Seg_Id, DWORD Offset, STR1 Name,
  WORD Type_Index );
```

```
void O96_Put_Ext_Def( BYTE Seg_Id, STR1 Name, WORD Type_Index );
void O96_Put_Mod_Ancestor( STR1 Name );
void O96_Put_Dbg_Seg_Def( BYTE Seg_Type, DWORD Base_Address,
  DWORD Seg_Size );
void O96_Put_Block_Def( STR1 Name, BYTE Seg_Id, DWORD Offset,
  DWORD Size, WORD Type_Index );
void O96_Put_Proc_Def( STR1 Name, BYTE Seg_Id, DWORD Offset,
  DWORD Size, WORD Type_Index, BYTE Scope, WORD Frame_Id,
  DWORD Frame_Offset, WORD Return_Offset,
  BYTE Prologue_Size );
void O96_Put_Block_End( void );
void O96_Put_Loc_Sym( BYTE Seg_Id, DWORD Offset, STR1 Name,
  WORD Type_Index );
void O96_Put_Lin_Num( BYTE Seg_Id, DWORD Offset, WORD Number );
void O96_Put_Cont_Hdr( BYTE Seg_Id, DWORD Offset );
void O96_Put_Cont_Str( PTR Cont_Ptr, WORD Cont_Length );
void O96_Put_Cont_Byte( BYTE Cont_Byte );
void O96_Put_Cont_Word( WORD Cont_Word );
void O96_Put_Cont_24Bits( DWORD Cont_24Bits );
void O96_Put_Cont_Dword( DWORD Cont_Dword );
void O96_Put_Fix_Cont( BYTE Fixup_Type, WORD Id,
  DWORD Ref_Offset );

/* o96in.c - OMF96P IN services */

extern BYTE O96_Rec_Type_In;
extern BYTE O96_Next_Type_In;
extern WORD O96_Rec_Num_In;
extern DWORD O96_Position_In;
extern DWORD O96_Next_Position_In;

PTR  O96_Open_IN( FILE **In_File_Ptr, char *Fname );
void O96_Close_IN( void );
void O96_Get_Next_Rec( void );
void O96_Get_Next_Mod( void );
void O96_Seek_In( DWORD Pos, BYTE New_Rec_Type );
void O96_Get_Info( void );
void O96_Set_Error_Handler( ERROR_HANDLER Error_Handler );

/* o96err.c - OMF96P standard error handler */

ERROR_HANDLER O96_Std_Error_Handler;
```

APPENDIX D

REFERENCES

1) MCS-96 Object Module Format EPS, Rev V3.0, Feb. 22, 1990.
2) Series III Microcomputer Development System Programmer's Reference Manual.
3) PEXN Interfacing Tool, Jan. 10, 1982.
4) IOBUFF, Buffered I/O Cluster, Mar. 16, 1982.

APPENDIX E

REVISION HISTORY

This appendix was added only when version X150 had been published.

- X150, March 1, 1983, by J. Yaari

1) OMF96P can use for its I/O operations either IOBUFF [4] or the I/O services of PL/M-96 Compiler.
2) In Chapter 3 and Appendix A, the entire section on type definition records is added.
3) In various places, 'Sym_Type BYTE' is replaced by 'Type_Index WORD'.
4) In Chapter 3 and Appendix A, added are: 'O96_Put_Proc_Def', 'O96_Put_Block_Def, 'O96_Put_Block_End'.
5) In Appendix C, the following sections are added: 'Based Attribute', 'Nice and Easy, 'Types of Leaves', 'Predefined Values for Numeric Leaves', 'Predefined Symbol Types', 'Scope', 'Block_Type'.
6) In Appendix C, the new procedures mentioned above are also added to this appendix.

- V2.0, Sept. 15, 1983, by A. Ziv

1) All the occurrences of 'Sym_Type BYTE' are replaced by 'Type_Index WORD'. Had been left by the previous version by mistake.
2) All the occurrences of 'Abs_Seg_Def(15) BYTE' are replaced by 'Abs_Seg_Def(20) BYTE'. Had been left by the previous version by mistake.
3) The definition of Block_Def is added to Chapter 4, Appendix A, and Appendix C.
4) The definition of Leaf_Def is added to Chapter 4, Appendix A, and Appendix C.
5) In Appendix A, all the occurrences of 'Type_P PTR' are replaced by 'Type_Index WORD'.
6) In Appendix A, in 'O96_Set_Error_Handler', the parameter 'Ptr_P PTR' is replaced by 'Proc_A ADDRESS'.
7) Code 17 is added to Appendix B and Appendix C.
8) In Appendix C, the section on 'Trn_Id' is replaced by the sections 'OMF_Ver' and 'Generator'.
9) Also added to Appendix C are: O96_Invalid_Type_Def IS '17', O96_Num_Of_Seg_Types IS '07H', O96_Max_Leaf_Type IS '106', O96_Numeric_Whole '89', O96_Max_Used_Predef_Type IS '0CH', O96_Num_Of_Used_Predef_Type IS '0DH'.

- V3.0, March 15, 1990, by A. Hunter

1) Updated to reflect 24-bit addressing extensions in OMF96 V3.1.